FUSION Design Specification

Locus Computing Corporation
IBM Corporation

CONTENTS

LIST OF FIGURES

FUSION Design Specification

Locus Computing Corporation
IBM Corporation

1. Introduction

This **FUSION** Design Specification provides the detailed Internal Architecture Specification for **FUSION**. It will describe the internal workings of each part of the system at an algorithmic level and will describe key data flows between components. Since **FUSION** intended to operate on a variety of underlying hardware and software platforms, this document does not provide highly detailed module level designs descriptions. Such designs would likely be highly specific to a particular base. However, this document should serve as an excellent starting point for producing such detail for a particular base.

The **FUSION** Functional Specification provides a rationale and an overview of the **FUSION** system, as well as a specification for the functionality provided and the interfaces to that functionality. Also provided is a description of error handling, performance goals, and impacts on the customer. That document also contains high level descriptions of the internal working of the system when such description is necessary to completely explain the desired functionality.

This document logically follows the **FUSION** Functional Specification in sequence. It provides a detailed specification for how to build each of the components of the **FUSION** system. This includes detailed algorithms used within each module, the key data structures used within each module, and descriptions of how data flows through the system.

One goal of the **FUSION** Design Specification is to not repeat the material presented in the **FUSION** Functional Specification. Consequently, this document presumes that the reader has already read and understood the **FUSION** Functional Specification. Without that background the reader of this document may become overwhelmed.

This components of **FUSION** are presented in this **FUSION** Design Specification in the same order as they were presented in the **FUSION** Functional Specification. This is done to aid the reader who is consulting both documents concurrently, as it allows the reader switch between the two documents without having to search extensively to find material on the same components.

Each section of the component specifications is structured in a similar way. First a high level description of the design is presented to give the reader an overall understanding of how the component functions. Then detailed specifications of the algorithms and data are presented. This is specified using a combination of code, pseudo-code, figures, and text. When appropriate, NIDL declarations for the new

RPCs used by that component are provided. One of the principal design criteria was to minimize changes required of the base system and of DCE. Since some such changes are inevitable, those components that require such changes will provide descriptions of those changes.

2. The Distributed Execution Environment

2.1 File System Enhancements

2.1.1 Remote Devices

2.1.1.1 Introduction

This document describes the design of remote devices in **FUSION**. It describes how the device inodes will be handled though AFS, how they are opened, and their operation after they are opened. The major issues to be addressed are:

open — It must be possible to associate a particular device name with the correct node that the device is physically attached to. This node will be called "the device node", to distinguish it from the node that the program is running on (which will be called "the execution node") or the node that stores the inode that represents this device (which will be called the "inode node"). In addition to associating the device name with the appropriate device node, this design must address some book keeping issues at open to maintain the UNIX semantics of only calling the device close routine on last close. One last issue for open is the design must be able to work with AIX multiplexed devices where the device open routine resolves that last part of the path naming the device.

read/write — The design must allow for read and write operation as if the device was stored locally. This is fairly straight forward except for cases where the user request very large single transfer operations. This design will specify the behavior of remote devices, but it is doubtful that a perfect solution can be found.

ioctl — Ioctl is a very tricky operation to implement remotely due to the free form nature of this operation. This design will treat ioctls in two classes. The first class of ioctl that will be considered are ioctl commands that are listed in the **FUSION** Functional Specification as supported between **FUSION** machines. These ioctls will be fully supported without any special consideration that the device may be remote. The other class of ioctl commands are "user defined" ioctls. These ioctl commands are not specifically recognized by the **FUSION** remote device support code. A mechanism will be provided so that a user can register an ioctl command with the **FUSION** remote device code. After being registered, user provided ioctl commands will be supported.

inode–only–operations — The remote device inode must be able to be referenced as well to support system calls such as stat, chown, chmod, etc. The design must be able to support these operations, regardless of whether the device node is the same as the inode node.

2.1.1.2 Lookup

In the current design, the lookup vnode op is handled by afs in the remote case and the afs "glue" vnode op would handle the local case. The logical file system lookup code will have to be modified to check to see if the vnode returned by lookup is a remote device vnode. If it is, it will call a remote device support routine that will allocate a new vnode that is initialized with remote device vnode ops. The vn_data of this new vnode points to a structure that includes a pointer to the vnode returned by lookup. The logical file system will return the new vnode in place of the vnode returned from the lookup vnode operation. For AIX 3.1 this will have to be done before lookuppn() checks for multiplexed device. This is so that the remote device lookup vnode op is in place before the last lookup is performed for the multiplexed device.

The remote device lookup vnode op will be a RPC call to a multiplex device lookup on the device node. The device node routine that resolves the last element of the path needs the vnode of the device inode, so the remote device lookup routine will use vn_prep_export() to get a handle which the server part of remote lookup can use vnode_reopen() to get the needed vnode. Then the reverse of this will be used to send the vnode returned from mpx_lookup back to the client node.

Pseudo code for rdev_lookup() is as follows:

```
rdev_lookup (device_vnode, channel_vnode, pathname, flags)

Algorithm:
        Determine the node that stores the device, lookup the rpc handle
        for the remote device service for that node.

        device_vnode_handle = vn_prep_export(device_vnode);
        rdev_lookup_server(rpc_handle, device_vnode_handle, pathname, flags,
                &channel_vnode_handle);
        channel_vnode = vnode_reopen(channel_vnode_handle);
        return;

rdev_lookup_server(rpc_handle, device_vnode_handle, pathname, flags,
        channel_vnode_handle_ptr)
        [in] rpc_handle;
        [in] device_vnode_handle;
        [in] pathname;
        [in] flags;
        [out] channel_vnode_handle_ptr;

Algorithm:
        dvp = vnode_reopen(device_vnode_handle);
        VNOP_LOOKUP(dvp, &channel_vnode_ptr, pathname, flags);
```

```
channel_vnode_handle_ptr = vp_prep_export(channel_vnode_ptr);
return;
```

This design has several implications for AFS. It means that AFS lookup will be able to return a special file vnode (even though afs cannot handle other operations on special files). It also means that AFS must be able to pass file types for device vnodes and multiplexed device vnodes.

2.1.1.3 Open

Remote open will be implemented through a remote open vnode op. This code will issue a RPC call to a server routine on the device storage node which will issue the device open call. In order to do this the code must determine which node it must contact. **NEEDSWORK:** The exact method of figuring out the device node is not yet determined. This information will be retrieved from the afsFid that afs returned in the vnode. When the remote server is called, an in–core vnode must be found (or created if this is a first open) in order to process the open call. This will be done using the primitives provided for open file export. The client side will call vn_prep_export() with the vnode from lookup to get a "remote vnode handle", which the server side can use to get the vnode using xvf_vreopen(). This handle is not the same as the "remote device handle" that remote open will be returning, so care must be exercised to not get them confused. In addition, the open count must be maintained. This will be done at the device node in the device vnode/gnode. When a remote open RPC comes in to the device node, the counts will be incremented as appropriate. In addition, a list of remote nodes that have this device open must be maintained. This list will be hung off a data structure that is keyed from the device vnode. A count of opens from that node is stored in the list. If a new node is added to the list, the "keep alive" service is notified the remote device server code cleanup routine must be called if the node goes down.

NEEDSWORK: If this is the first open of a remote tty, and we don't already have a controlling terminal, we need to establish controling terminal or a new session as needed. This looks as though it will require a change to the base operating system tty line discipline code to understand that there may be a kproc acting as a proxy for a remote open, otherwise it may set the kproc to be the process group for this tty.

The end result of the remote open operation is to actually open the remote device and to return a remote device handle that will be used by the rest of the remote device client routines to properly connect to the remote device server routines. A pointer to the handle will be stored in the vnode data part of the client vnode. Pseudo code for rdev_open:

```
rdev_open(vp, open_flags, extension, vinfop)

Algorithm:
#ifdef AIX_V3
        rc = VNOP_ACCESS(vp, (mode from flags), ACC_SELF);
```

```
        if (rc != 0)
                return(rc);
#endif

        Determine the node that stores the device, lookup the rpc handle
        for the remote device service for that node.

        device_vnode_handle = vn_prep_export(vp);
        rdev_open_server(rpc_handle, device_vnode_handle, flags, ext, vinfo,
                nodeid, &remote_io_handle, &uerror, &specflags);
        if (uerror != 0) {
                return(stderr2errno(uerror));
        }
        if (specflags == DEVNULL) {
                set vnode ops for data ops to local dev null ops.
        } else if (specflags == DEVTTY) {
                if (u.u_ttynode == curnode) {
                        NEEDSWORK: code to hack remote vnode to local
                        controlling tty vnode + extra open.
                } else {
                        lookup the rpc handle for u.u_ttynode.
                        rdev_open_ctty(rpc_handle, u.u_ttyd, u.u_ttympx,
                                flags, ext, vinfo, curnode, &remote_io_handle,
                                &uerror);
                        if (uerror != 0) {
                                return(stderr2errno(uerror);
                        }
                        vp->v_data = remote_io_handle;
        } else {
                vp->v_data = remote_io_handle;
        }
        return;

rdev_open_server(rpc_handle, device_vnode_handle, flags, ext, vinfop,
        nodeid, remote_io_handlep, uerror, specflags)
        [in] rpc_handle;
        [in] device_vnode_handle;
        [in] flags;
        [in] ext;
        [in] vinfop;
        [in] nodeid;
        [out] remote_io_handlep;
        [out] uerror;
```

```
        [out] specflags;

Algorithm:
        vp = vnode_reopen(device_vnode_handle);
        if (isdevnull(vp)) {
                *specflags = DEVNULL;
                return;
        }
        if (isdevtty(vp)) {
                *specflags = DEVTTY;
                return;
        }
        rc = VNOP_OPEN(vp, flags, ext, vinfop);
        if (rc != 0) {
                uerror = errno2stderr(rc);
                return;
        }

        /* keep track of which nodes have this dev open */
        remote_open_node_struct = find_remote(vp, nodeid);
        if (remote_open_node_struct == NULL) {
                remote_open_node_struct = add_remote(vp, nodeid);
                remote_open_node_struct->opencount = 1;

                /* register with the "keep alive service" */

                monitor_nodedown(node, rdevclenup_functp);
        } else {
                remote_open_node_struct->opencount++;
        }
        remote_io_handlep = make_remote_handle(vp);
        return;
```

2.1.1.4 Migration

Reopen operation must set up remote vnodes that are similar in function to a remote open. Reopen is implemented in two stages. The first stage is the "vno_prep_export()" vnode op that is used by the export file operation that is called as part of the process migration operation. This operation will create a "reopen handle" that will be shipped with the migrating process for use with the "fo_reopen()" VFS operation at the new node. The reopen VFS operation will find or create an appropriate vnode at the new node and call its reopen vnode operation. If the new node is remote from the device node, the reopen vnode operation will make an RPC that will call a remote reopen function at the device node. This operation will increment the open counts for the device vnode/gnode as well as the count that is in

the remote node extension entry. The remote node will be added to "keep alive" service if it is not already registered there. This RPC will return a device handle that can be used for the rest of the remote device operations and a pointer to this handle will be stored in the vnode data part of the new node device vnode. If this operation is the result of a migration, the old (local) process will exit when the migration is complete. This will result in a close of the local instance of the device, finishing the transition from a local device to a remote device. If this was a remote to remote migration, this procedure will be modified by having the "vno_prep_export()" operation being set up to give a reopen handle that points at the device node. Then the reopen operation would happen at the device node just as in the migrate from local case.

2.1.1.5 Read/write

Remote read/write vnode op will be set to routines that will call RPC routines to "function ship" the operation to the device node. NEEDSWORK: how are very large user buffers handled?

2.1.1.6 inode–only–operations

This design assumes that inode only operations will be able to be supported by the AFS operations just as if device inodes were file inodes. The remote device vnode operations will call the coresponding AFS vnode operation that is in the saved vnode from the original lookup though AFS. All of the arguments to the remote device operation will be passed through to the AFS operation except that a pointer to the AFS vnode will be passed instead of the pointer to the remote device vnode.

2.1.1.7 Ioctl

2.1.1.7.1 Introduction

One of the operations that a remote special file server must support is the ioctl() call. This presents several problems which are outlined below, together with some proposed solutions.

The cause of these problems is threefold. First of all, the ioctl() interface is untyped, and therefore is difficult to express in a strongly typed language such as NIDL.

Secondly, the ioctl() interface was designed under the assumption that the target of the call (i.e. a device driver) has unlimited and efficient access to the address space of the caller. This is obviously not the case in the RPC-based system being built.

Thirdly, current Unix standards such as POSIX are aimed at source-code compatibility, and not binary compatibility. But, our remote special file facility will be a case of one kernel calling the services of another, possibly quite different kernel, for example AIX3.1 calling System V.4. This kind of inter-operability has more of the flavor of binary compatibility, and raises problems 2, 3, 4 and 5 below. It is assumed such inter-operability is a requirement for FUSION software.

The good news is that the code to solve most of these problems is fairly trivial to write.

Section 2 outlines problems with remote ioctl()'s, and details solutions. Section 3 contains sketches of what the code for handling ioctl()'s does at the client and server. Section 4 contains a list of well known ioctl()'s that need to be support.

2.1.1.7.2 Problems with ioctl()'s

2.1.1.7.2.1 Problem 1: ioctl() is untypable in NIDL

The ioctl() interface looks like this:

ioctl(fd, CMD, ARG);

The type (and existence) of ARG depends on the value of CMD. For example, here are some of the types assumed by ARG, for various ioctl()'s:

none
char
short
int
int *
long
struct termio *
struct sgtty *
struct ltchars *
struct winsize *
struct ttypagestat *
struct rtentry *
struct ifreq *
struct ifconf *
struct arpreq *

It is not possible to write down a single NIDL operation declaration which covers all these cases, and which would actually work.

2.1.1.7.2.2 Problem 2: CMD encoding's are not uniform

The encodings of CMD are not identical for all kernels. For example, in AIX 3.1 TCGETA is defined as:

```
#define TCGETA  (TIOC | 5)
```

whereas in System V Release 3.2 TCGETA is defined as:

```
#define TCGETA  (TIOC | 1)
```

Our solution to both of these problems is to provide a separate NIDL-defined operation for each CMD symbol. So the interface definition includes:

```
void ioctl_tcgeta(
     [in] handle_t    device,
     [out] termio_t  *arg,
     [out] errno_t   *copy_of_u_error
```

```
);

[idempotent]
void ioctl_tcseta(
     [in] handle_t    device,
     [in] termio_t   *arg,
     [out] errno_t   *copy_of_u_error
);

void ioctl_tcflsh(
     [in] handle_t    device,
     [in] int         queue_selector,
     [out] errno_t   *copy_of_u_error
);

void ioctl_tcsbrk(
     [in] handle_t    device,
     [in] int         whether_to_send_break,
     [out] errno_t   *copy_of_u_error
);
```

In order to support user written ioctls, the user is allowed to register a "callout" for his command. Then his callout would have to use his NIDL interface to support his ioctl.

2.1.1.7.2.3 Problem 3: bit field encodings are not uniform

The RPC layer can encode C structs, such as struct termio, in NDR form. However, the layout of bit fields within integer fields in the struct could, in general, vary from system to system. For example, POSIX only defines the NAMES of the bits inside the fields of the struct termio. Each implementation is free to "#define IGNBRK" to be any value it likes. This means the c_iflag field in the struct which is passed through the RPC cannot have its normal type, "unsigned short", and the function cannot rely on the local pre-processor-defined masks for the definition of the individual bits.

The solution is to make NIDL type definitions for the bit fields using the "bitset enum" and "enum" type constructors, and to provide mapping functions, between the NIDL datatype, and the native Unix datatype. The mapping functions would need to be called in the client and in the server.

For example, the "struct termio" contains a field of type "unsigned short" called "c_iflag". To provide a representation-independent version of this flag, one would define, in NIDL, this datatype:

```
typedef short bitset enum
```

```
{
    /* These are POSIX defined. */

    brkint,
    icrnl,
    ignbrk,
    igncr,
    ignpar,
    inlcr,
    inpck,
    istrip,
    ixoff,
    ixon,
    parmrk,

    /* These ones are defined in AIX, inter alia. */

    iuclc,
    ixany,
    ascedit
} iflag_nt;
```

The client code would call a function like the following to convert from "native iflag" to "NIDL iflag":

```
iflag_nt                                                    |
native_to_iflag_nt(unsigned short native)                   |
{
    iflag_nt return_value;

    return_value = 0;

    #ifdef BRKINT
    if (native & BRKINT)
        return_value |= brkint;
    #endif
     ...

    #ifdef ASCEDIT
    if (native & ASCEDIT)
        return_value |= brkint;
    #endif

    return return_value;
}
```

There would be a similar inverse function, iflag_nt_to_native(), to map the other way.

2.1.1.7.2.4 Problem 4: some struct definitions must be NIDLized

As a consequence of the above, new type definitions for those structs which appear in ioctl() RPC's and which have bit-fields defined in them are required.

Thus, the NIDL declaration for ioctl(TCSETA, ...) would look like:

```
[idempotent]
void ioctl_tcseta(
     [in] handle_t    device,
     [in] termio_nt   *arg,
     [out] errno_t   *copy_of_u_error
);
```

where termio_nt is declared something like:

```
typedef struct termio
{
    iflag_nt          iflag;
    oflag_nt          oflag;
    cflag_nt          cflag;
    char_size_nt      char_size;
    baud_rate_nt      baud_rate;
    lflag_nt          lflag;
    char              line;          /* line discipline */

    /* POSIX defined names of special characters. */

    char              veof;
    char              veol;
    char              verase;
    char              vintr;
    char              vkill;
    char              vquit;
    char              vsusp;
    char              vstart;
    char              vstop;
    char              vmin;
    char              vtime;

    /* AIX defines these. */

    char              veol2;
    char              vswtch;
} termio_nt;
```

This now means that a mapping functions to map between termio_nt format and "native struct termio" format is required.

2.1.1.7.2.5 Problem 5: errno encodings are not uniform

In coding the remote special file server (and possibly other servers), it will sometimes be necessary to reflect back to the client, error conditions that originated in the server's kernel. These conditions, coded as numbers, must be converted to and from a NIDL-defined (representation independent) datatype, because there is no guarantee that all kernels will encode the same error symbol with the same integer value. For example, POSIX defines a set of error values, but only by their symbolic name. Two kernels can claim POSIX compliance, even though one is compiled with "#define ENOMEM 22" and the other is compiled with "#define ENOMEM 44".

The solution here is to define, in NIDL, a type thus:

```
typedef enum
{
    /* POSIX defined error conditions */
    e2big,
    eacces,
    ebadf,
    ebusy,
     ...

    /* System V error conditions */
    eadv,
    ebade,
    ebadfd,
    ebadmsg,
    ebadr,

     ...
} errno_nt;
```

The server would contain a function like this:

```
errno_nt errno_to_errno_nt (int errno)
{
    switch (errno)
    {
     ...
    case ENOMEM:
         return enomem;

     ...
 }
```

And the client would contain a function like this:

```
int errno_nt_to_errno (errno_nt nidl_errno)
{
    switch (nidl_errno)
    {
     ...
    case enomem:
         return ENOMEM;

     ...
 }
```

(Of course, both functions would probably be table driven.)

2.1.1.7.3 Code sketches for handling remote ioctl()'s

Here is roughly what would happen in the client and server when performing a remote ioctl().

2.1.1.7.3.1 Client side (Special File VFS code):

```
switch (ioctl_cmd) {
#ifdef SAMPLE_IOCTL_CMD
case SAMPLE_IOCTL_CMD:
      Get a well-typed pointer to the argument (if it's a struct);
            OR
      Get a well-typed scalar containing the argument (if it's an int, char, long);

      If (the argument is a struct which needs to be NIDLized)
            allocate an instance of the NIDL type;
            convert the local struct to NIDL form;

      Call strongly-typed client-stub code;

      If (the ioctl() returns a struct which needs to be de-NIDLized)
            convert the NIDL struct to local form;

      If (returned nidl_errno indicates an error)
            convert nidl_errno to local_errno;
            store local_errno in u.u_error;

      If (appropriate)
            return the value returned by the RPC;
      break;
#endif SAMPLE_IOCTL_CMD

default:
```

call user registered ioctl routines.

}

2.1.1.7.3.2 Server side (Remote device server):

(entered at the manager procedure)

```
int manager_proc_for_ioctl_sample(
        handle_t device_handle,
        sample_arg_t *arg,
        errno_nt *nidl_errno)
{
#ifndef SAMPLE_IOCTL_CMD:
        *nidl_errno = errno_to_errno_nt (EINVAL);
        return -1;
#else
        Ascertain actual device (fd, or major/minor) from device_handle;

        If (the argument is a struct which needs to be de-NIDLized)
                allocate an instance of the local struct type;
                convert the NIDL struct to local form;

        Call driver (ioctl system call, or through devsw[]), with cmd =
        SAMPLE_IOCTL_CMD, and arg = address of local struct;

        If (returned errno or u.u_error indicates an error)
                convert it to NIDL form and store in *nidl_errno;

        If (appropriate)
                return the value returned by the driver's ioctl function;
#endif }
```

2.1.1.7.4 A list of most (?) well-known ioctl() CMD's

The sources for the following were the AIX PS2 Gen1 source code, and include files on System V, and BSD. The ioctl()'s have been grouped according to origin and function. The groups are listed in approximate order of importance. (i.e. if some of these groups are not going to be supported, they should be deleted from the end of the list, rather than the start.)

```
/* Origin:    ATT Unix.
 * Purpose:   Terminal control.
 * Arg types: struct * termio
 */
```

```
TCFLSH,
TCGETA,
TCSBRK,
TCSETA,
TCSETAF,
TCSETAW,
TCXONC,

            /* Origin:    Version 7.
             * Purpose:   Terminal control.
             * Arg types: struct * sgttyb
             */
TIOCGETP,
TIOCSETP,
TIOCSETN,

TIOCEXCL,
TIOCNXCL,

TIOCHPCL,

TIOCGETD,
TIOCSETD,

TIOCFLUSH,

            /* Origin:    BSD.
             * Purpose:   Terminal control.
             * Arg types: scalars only, no structs.
             */
TIOCSTI,

TIOCCBRK,
TIOCSBRK,

TIOCCDTR,
TIOCSDTR,

TIOCSTART,
TIOCSTOP

TIOCSPGRP,
TIOCGPGRP,
```

```
TIOCOUTQ,

            /* Origin:    BSD (?)
             * Purpose:   Terminal control.
             * Arg types: struct * tchars,
             *            struct * ltchars,
             *            struct * ttypagestat,
             *            struct * modem_control,
             *            struct * tiocpkt.
             */
TIOCBIC,
TIOCBIS,

TIOCCGET,
TIOCCSET,

TIOCGET,
TIOCSET,

TIOCGETC,
TIOCSETC,

TIOCGLTC,
TIOCSLTC,

TIOCGPAGE,
TIOCSPAGE,

TIOCLBIC,
TIOCLBIS,

TIOCLGET,
TIOCLSET,

TIOCMODG,
TIOCMODS,
TIOCMBIC,
TIOCMBIS,
TIOCMGET,
TIOCMSET,

TIOCNOTTY,
TIOCPKT,
```

```
TIOCREMOTE,

            /* Origin:    BSD (?)
             * Purpose:   Various control functions.
             * Arg types: scalars only, no structs.
             * Notes:     Some apply to ANY file, not just devices.
             */
FIOCLEX,    /* set close on exec on fd */
FIONCLEX,   /* remove close on exec */
FIONREAD,   /* get # bytes to read */
FIONBIO,    /* set/clear non-blocking i/o */
FIOASYNC,   /* set/clear async i/o */
FIOSETOWN,  /* set owner */
FIOGETOWN,  /* get owner */

            /* Origin:    AIX (?)
             * Purpose:   ??
             * Arg types: struct * termcb
             * Notes:     Found in <sys/termio.h>.
             *            ldopen, ldclose, ldchg interpreted by lpld.c
             *            (the line printer device-independent part).
             *            Also tt1.c interprets some.
             */
LDOPEN,
LDCLOSE,
LDCHG,
LDGETT,     /* ? */
LDSETT,     /* ? */
LDGETDT,    /* ? */
LDSETDT,    /* ? */

            /* Origin:    AIX.
             * Purpose:   Terminal control.
             * Arg types: struct * tty_page
             */
TCGLEN,
TCSLEN,

            /* Origin:    AIX Gen 1 (?).
             * Purpose:   Get and set partition info on a raw disk .
             * Arg types: struct * dkpart
             */
DKGETPART,
```

```
DKSETPART,

            /* Origin:    AIX.
             * Purpose:   Gets device-specific information.
             * Arg types: struct * devinfo
             * Notes:     See "man devinfo" on AIX.
             */
IOCINFO,    /* Gets device info */
IOCTYPE,    /* Return device type, left shifted 8 bits */

            /* Origin:    BSD networking code.
             * Purpose:   Manipulate network layers below a socket.
             * Arg types: struct * rtentry,
             *            struct * ifreq,
             *            struct * ifconf,
             *            struct * arpreq,
             *            struct * ie5_arpreq
             * Notes:     These might not belong in the special
             *            file server interface, but probably
             *            belong in the socket server interface.
             */
SIOCADDRT,
SIOCATMARK,
SIOCDARP,
SIOCDELRT,
SIOCGARP,
SIOCGHIWAT,
SIOCGIFADDR,
SIOCGIFBRDADDR,
SIOCGIFCONF,
SIOCGIFDSTADDR,
SIOCGIFFLAGS,
SIOCGIFMETRIC,
SIOCGIFMTU,
SIOCGIFNETMASK,
SIOCGLOWAT,
SIOCGPGRP,
SIOCSARP,
SIOCSHIWAT,
SIOCSIFADDR,
SIOCSIFBRDADDR,
SIOCSIFDSTADDR,
SIOCSIFFLAGS,
```

SIOCSIFMETRIC,
SIOCSIFMTU,
SIOCSIFNETMASK,
SIOCSLOWAT,
SIOCSPGRP

2.1.2 Remote Pipes

This document describes the design of remote pipe and FIFO in **FUSION**. Emphasis is on the FIFO client/server protocol and the interface between the server's policy and storage modules. In addition to the major goal of providing UNIX semantics for FIFOs in a distributed environment, two other considerations drive the **FUSION** FIFO implementation: separation of data storage from distributed FIFO management, and optimization to allow local data storage when all processes using a FIFO are running on the same host.

Throughout, both unnamed pipes and named pipes are referred to as "FIFOs", except where an explicit distinction is made between the two.

If all processes using a FIFO reside on the same host, no significant performance penalty should be imposed for using the **FUSION** distributed FIFO code. In particular, we envision an HP-style scheme for supporting named FIFOs. If all processes that access a named FIFO reside on host A, then host A should do the work of storing the FIFO data even though the named FIFO special file is stored on host B, the controlling host for the FIFO. If an additional process on host C opens the FIFO, host B should then reclaim the FIFO and resume its responsibility as the storage host. This process of transferring responsibility for the FIFO to a using host is called FIFO lending (or borrowing). When a client host is acting as surrogate storage host, the FIFO is said to be "on loan" to the client host.

2.1.2.1 The FIFO Virtual File System

The **FUSION** FIFO implementation uses the client/server paradigm, but the client and server are really parts of the same virtual file system code. If all processes using a particular FIFO are on one host, the client VFS at that host will be given responsibility for managing and storing data for the FIFO. Since the client may need to perform both storage host and using host functions (*e.g.* when the FIFO is on loan; see Section 2.1.2.3.4), the code for both client and server functionality are placed in a single FIFO VFS. Whether the FIFO VFS acts as server or client depends on the storage host of the individual FIFO being accessed.

When we speak of the "FIFO client" or "FIFO server", we really mean "the FIFO VFS acting as client" or "the FIFO VFS acting as server."

Figure 1 shows the **FUSION** FIFO VFS. It is multi-threaded and comprises three functional units: a VFS+ interface, an RPC interface, and a policy module (PM). When **FUSION** code is added to a new vendor kernel, the VFS+ and RPC interface code and the policy module are ported directly, and the vendor supplies their own storage module implementation.

The VFS+ interface provides access to FIFOs for processes running on the local host. The RPC interface provides access to locally stored FIFOs for processes running remotely. The RPC calls used by this interface closely resemble the vnode operations used by the VFS interface, but there are also some additional RPC calls to manage

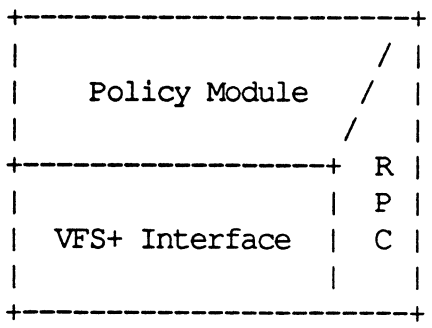


Figure 1. The FUSION FIFO Virtual File System

distributed FIFOs. For example, one such additional RPC informs a FIFO client that it should take responsibility for managing a particular FIFO, since all processes using that FIFO reside on the client's host.

The policy module is the heart of distributed FIFO support in **FUSION**. It maintains descriptors for both locally and remotely stored FIFOs. Operations on descriptors for local FIFOs trigger corresponding calls to the storage module to store or retrieve data. Operations on descriptors for remote FIFOs trigger RPC calls to the remote FIFO server.

The storage module presents a first in/first out byte stream abstraction to the policy module. Storage module FIFOs can be created, written to, read from, and destroyed by subroutine calls from the policy module. The policy/storage interface is well defined and allows vendors to easily fit their preferred FIFO implementation into the **FUSION** distributed FIFO model. Figure 2 describes the **FUSION** FIFO server data structures, notice that the local VFS and **FUSION** VFS are connected through the **FUSION** FIFO descriptor, it stores all the necessary information for the **FUSION** FIFO operations. The **FUSION** VFS comes to exist when the pipe system call is made locally or the remote access to a named pipe is requested from a client node. There is a one to one correspondence between the FIFO descriptor and a remote client during the **FUSION** FIFO operations. *

2.1.2.2 Synchronization

In a non-preemptable uniprocessor UNIX kernel, FIFO I/O synchronization is achieved using sleep() and wakeup(). If a process's I/O request can be satisfied, it reads or writes its data and returns to user mode. If not, it sleeps on some appropriate channel and is later reawakened by the occurrence of some event, *e.g.* another process writing data to an empty FIFO, and it can then proceed.

We assume kernel threads are plentiful and cheap, allowing FIFO RPC code to sleep at the server end just as local FIFO operations do. If sleeping were to be done at the client end, an elaborate callback mechanism would be required. We trust that the provided kernel thread implementation will make this scheme unnecessary, all blocking being done at the server's **FUSION** VFS layer and can be interrupted. |

It is possible to have multiple readers and writers for a single FIFO. It is assumed that concurrent access is serialized in some convenient fashion, for instance by a token passing mechanism.

2.1.2.3 End-of-file Condition

The read operation gets a return value of zero if there is no data in the FIFO and no writing processes. Whenever data is removed from the FIFO there is a check for any process that may be suspended, awaiting more space in the FIFO so they may continue writing. If the FIFO storage node crashes during the read operation, the system generates a server node down(SR_DOWN) error condition. The writer processes receive the SIGPIPE signal when attempting to write a FIFO that has no reading processes or the FIFO storage node went down during the write operation.

2.1.2.4 The Policy Module

The policy module maintains a pool of FIFO descriptors. A FIFO descriptor contains all the state that the policy module needs for managing a FIFO. Information in the descriptor indicates whether it represents a locally or remotely stored FIFO. Each descriptor is named by an ordered pair of <hostnbr, FID>, where hostnbr is a 32-bit **FUSION** host number and FID is the AFS file identifier associated with the special file of a named FIFO. For unnamed pipes, FID is not an actual AFS file identifier but a special "cookie" assigned by the pipe's storage host policy module. Cookies are distinguished from real FIDs by having an illegal value in one of the FID subfields. (Since AFS FIDs are 24 bytes long, it may be desirable to use a shorter cookie for *all* FIFOs. This should be no problem as long as the name given to a FIFO descriptor can uniquely identify the storage host FIFO.)

Figures 2 and 3 show the **FUSION** client/server organization in four different layers, the top two layers are the **FUSION** policy module. We describe a named FIFO open operation example, explaining the function of each layer. Suppose that a client application issues an open system call for reading to a named pipe at remote host, this is first handled by physical file system pathname lookup routine which will contact client cache manager for each pathname component. A vnode with an AFS file identifier is returned to the client at the end of pathname lookup, if the client can't find the vnode from the local FIFO descriptor list then it will send a message to the server to open the named FIFO and send back the information of the named FIFO so that this named FIFO can be opened at client. Upon receiving the RPC request from the client, the **FUSION** FIFO server will search the virtual file system for this vnode. If the search failed, the server will create a new **FUSION** vnode and FIFO descriptor for this named FIFO. Otherwise, the named FIFO is opened at server node by using base system vnode open routine, the **FUSION** vnode pointer is returned to the client so that the named FIFO can be opened at client node. If a writing process exists before the open or if the client opens the named FIFO with the no delay option, the open returns immediately, even if there is no writer. But if neither condition is true, the process sleeps at server end at the **FUSION** layer until a writer process opens the named FIFO.

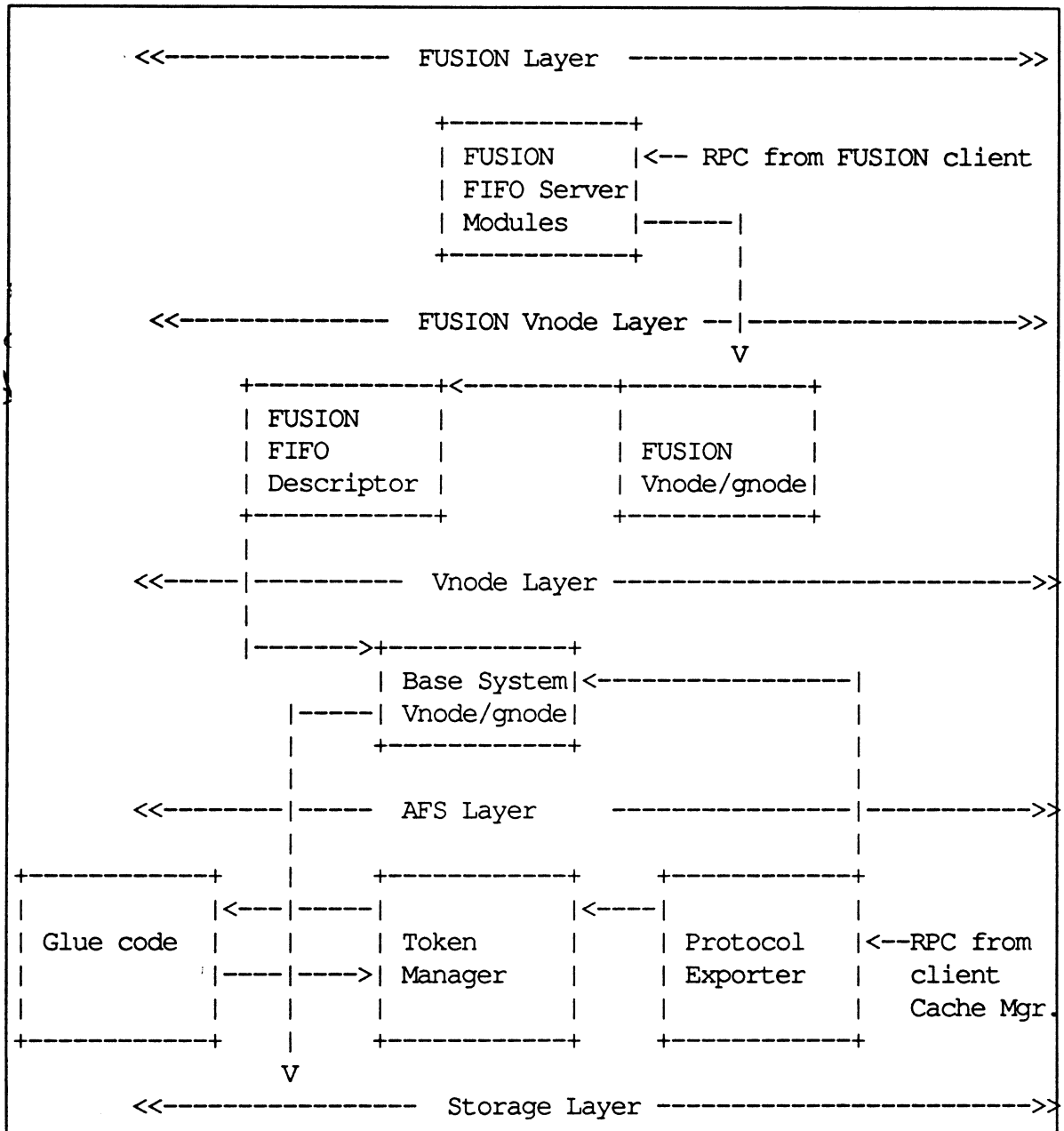


Figure 2. FUSION FIFO Server data structure

The data structure of the **FUSION** single node unnamed pipe operation is similar to Figure 2 except that there is no AFS layer involvement at all.

2.1.2.4.1 FIFO Descriptors Are Objects

For purposes of the policy module to policy module RPC protocols, FIFO descriptors are given a particular state (see Section 2.1.2.3.4, **FIFO Lending Protocol**). Since

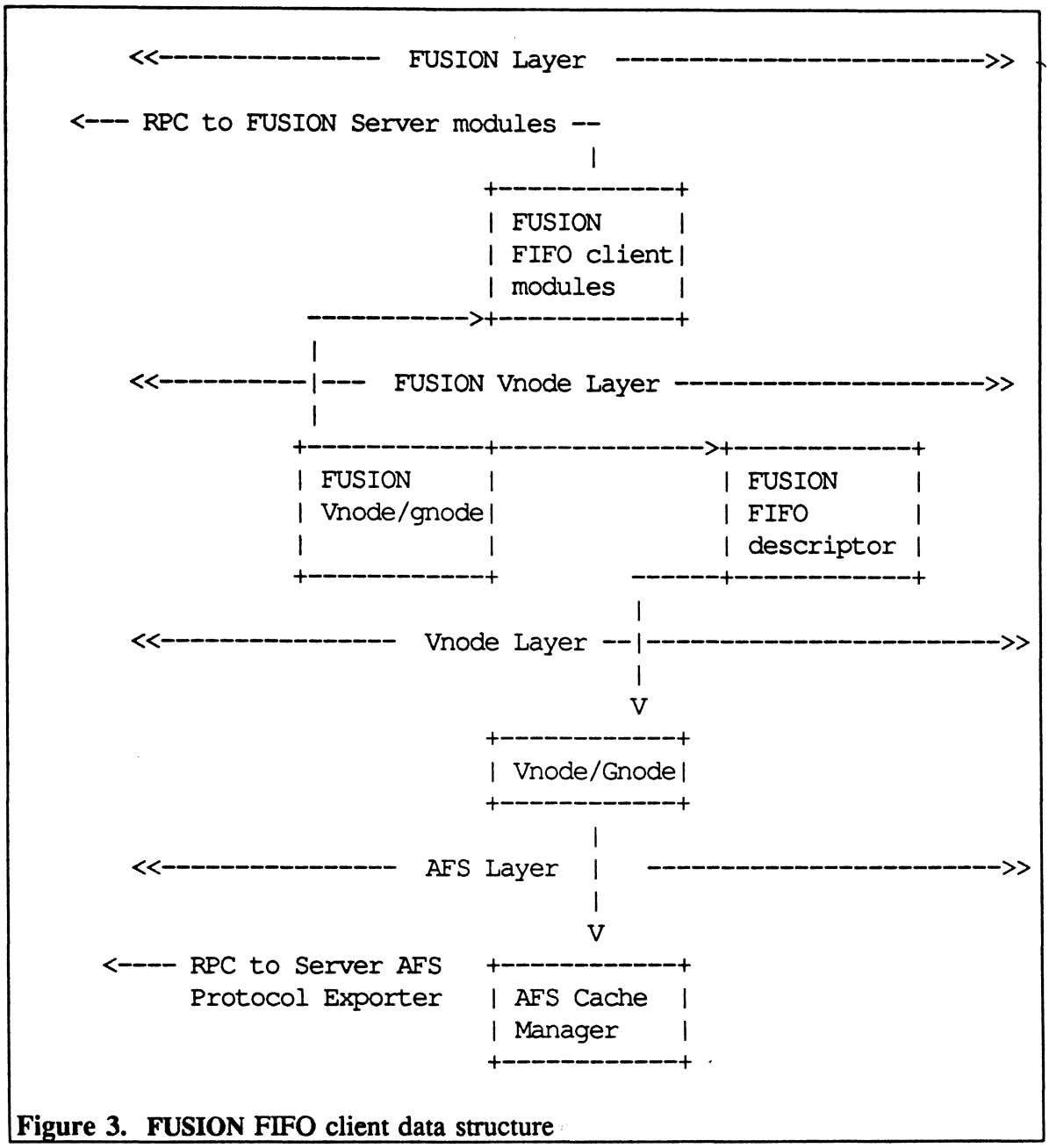


Figure 3. FUSION FIFO client data structure

the behavior of the policy module will be different depending on the state of particular FIFO descriptors, we envision an object-oriented approach. Read, write, open, and close operations will be received from either the VFS+ or RPC interface, and will be applied to particular FIFO descriptors in the same way regardless of their state. The descriptor state can be used as an index into a "method table" of function addresses. Thus the low level code executed for a READ operation depends on the descriptor state. For example, the READ routine for a descriptor in CLIENT state issues an RPC call, while the READ routine for a descriptor in SERVER state actually calls the

storage module to read some data.

2.1.2.4.2 Preserving PIPE_BUF Semantics In A Heterogeneous Environment

The POSIX standard guarantees that WRITE requests of less than or equal to PIPE_BUF bytes will be written atomically into a pipe or FIFO, that is, without being interleaved with data from other WRITE requests. The value of PIPE_BUF varies from host to host in a heterogeneous environment. To comply with the standard, the policy module must choose the largest PIPE_BUF value from among all using hosts that have a FIFO open for writing. Thus the client host's PIPE_BUF value must be passed along with an server_rfifo_reopen() RPC, and the operative value of PIPE_BUF may be different from descriptor to descriptor on the same server host.

If a FIFO's server host storage module implements FIFOs using a pinned memory page of 4K bytes and the FIFO is opened for writing by a remote host whose native PIPE_BUF is 8K, atomic writes of greater than 4K cannot be guaranteed at the server host. In order to perform the FIFO operations efficiently, we have to find a node which has the largest PIPE_BUF value to act as the storage node.

2.1.2.4.3 Notes on Policy Module RPC Protocol

1. Each RPC must pass a *flags* parameter, so that appropriate blocking behavior can be done (i.e. O_NDELAY). The FIFO descriptor should not have to keep track of the usage modes of all using hosts.
2. The client machine's PIPE_BUF value must be passed with the server_rfifo_reopen() RPC,

2.1.2.4.4 FIFO Lending Protocol

When all processes using a FIFO reside on the same host, RPC overhead can be avoided by having that host manage the FIFO. A state transition protocol is used to allow the controlling host to temporarily lend a FIFO to another host when all using processes reside on the second host. If a process from a third host opens the FIFO, the controlling host can reclaim responsibility for managing the FIFO.

Figures 4 and 5 show the state transition diagrams for controlling and non-controlling hosts respectively.

This scheme allows diskless hosts to access named FIFOs without network overhead. Further, it allows for smooth transition back to the controlling host when a reclaim operation is done.

Section 2.1.2.3.4.1 presents a typical FIFO lending scenario. Section 2.1.2.3.4.2 gives a brief description of each possible FIFO descriptor state.

2.1.2.4.4.1 FIFO Lending Example

Note: Currently we plan to have remote hosts that open a named FIFO be placed immediately into BORROWER state, that is, a FIFO will initially be on loan whenever possible.

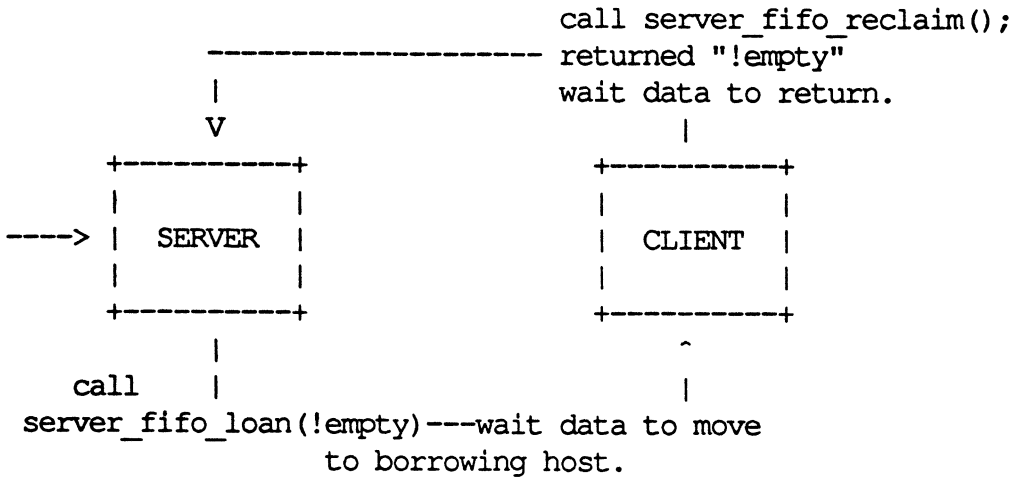


Figure 4. State transitions at the controlling host

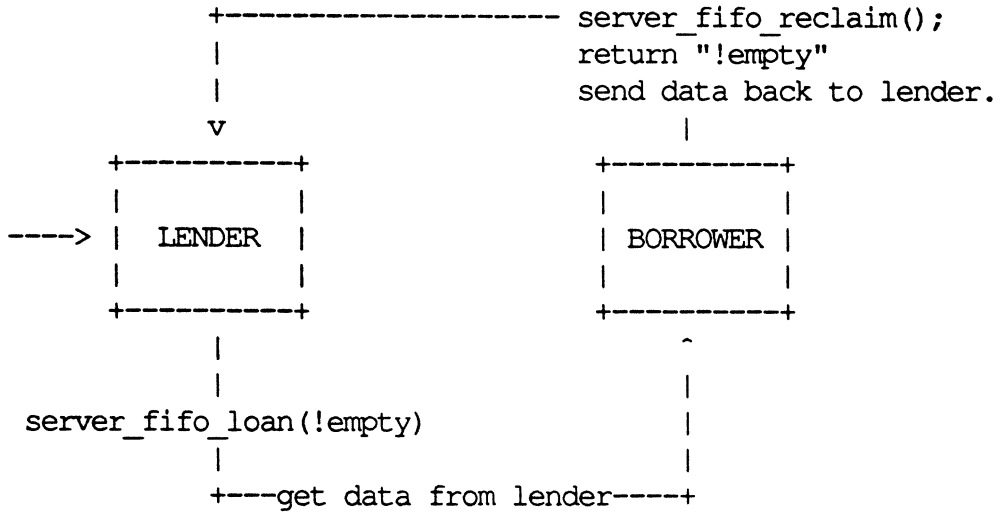


Figure 5. State transitions at the non-controlling host

The primary states are CLIENT and SERVER. A FIFO descriptor on a host is created in SERVER state when a named FIFO special file is opened on that host, or when a process on that host makes a pipe(2) system call. A FIFO descriptor is created in CLIENT state when a process opens a named FIFO special file on another host, or when a process with an open FIFO migrates to this host. Note that the names CLIENT and SERVER are not strictly correct, in that SERVER really denotes the controlling host. I/O requests made by processes on the controlling host are fulfilled without the FIFO descriptor ever leaving SERVER state, and without the involvement of any CLIENT state FIFO descriptor.

Suppose that processes on host A open a named FIFO on host B. FIFO descriptors are set up at both hosts. Initially the descriptor at A is in CLIENT state and the

descriptor at B is in SERVER state.

The host B policy module may choose to lend the FIFO to host A, provided that the only processes using the FIFO are on A. To begin the lending protocol, host B sends a fifo_to_client() RPC call to the host A policy module. The FIFO descriptor at host A can be identified by the **FUSION** vnode at host B and its node number.

The fifo_to_client() call takes an input parameter, isempty, which is true if and only if there is no data in the FIFO. If the FIFO was empty, host B goes into LENDER state. Upon receiving a fifo_to_client() RPC with isempty set to true, Host A becomes responsible for managing the FIFO and is placed into BORROWER state. If there was data in host B FIFO, then all the READ and WRITE requests are blocked until the host B FIFO data is moved to the host A. The host A FIFO descriptor is now in LENDER state and responsible for managing the FIFO.

When a third host, C, attempts to open the FIFO, the LENDER descriptor on host B sends an server_rfifo_reclaim() RPC to host A. The protocol for reclaiming the FIFO from the BORROWER host is much as described above.

2.1.2.4.4.2 FIFO Descriptor States

2.1.2.4.4.2.1 SERVER State.

Generally, a FIFO descriptor in SERVER state indicates that this is the controlling host for the FIFO, and the local storage module is used to maintain the FIFO data unless the FIFO has been loaned out to another host. When a descriptor object in SERVER state detects that all its users are on a single host, it loans the FIFO to that host using a fifo_to_client() RPC call. If the FIFO was empty at the time, it goes immediately to LENDER state; otherwise, it has to wait for the FIFO data to copy to the borrowing host.

2.1.2.4.4.2.2 LENDER State.

In this state, the controlling host FIFO descriptor is quiescent. All I/O is being performed at the borrowing host. If a third host attempts to open the FIFO, the LENDER descriptor must issue a server_rfifo_reclaim() RPC call to get the FIFO back from the borrowing host. It then either reverts to SERVER state if the FIFO was empty or wait until the FIFO data has been returned to the lending host.

2.1.2.4.4.2.3 CLIENT State.

A FIFO descriptor in CLIENT state represents a true remote FIFO client. All operations are passed via RPC to the SERVER descriptor at the FIFO's controlling host. Upon receiving a fifo_to_client() RPC call, CLIENT descriptors prepare to manage the FIFO locally by moving to BORROWER state, depending on whether there is some data left in the FIFO on the lender host.

2.1.2.4.4.2.4 BORROWER State.

In this state the FIFO descriptor object has temporary control of the FIFO, and all I/O is done using the local storage module. Receipt of an server_rfifo_reclaim() RPC call

from the lending host causes the FIFO descriptor to return to CLIENT state if the FIFO was empty, or return data to the lender.

2.1.2.5 Policy Module/Storage Module Interface

The policy module uses a vendor-specific storage module to store and retrieve FIFO data. The storage module presents a simple FIFO abstraction, "pipe objects" or pobj's. Routines are provided to open, close, read, write and select pipe objects, test for blocking conditions (empty FIFO, full FIFO), count the number of unread bytes in the FIFO. To minimize data copying, storage module routines pass data to and from the policy module using the uio structure to indicate the amount of data to transfer and the buffer location to store the data. The storage module has no knowledge of the FIFO lending protocol or any other aspect of the distributed environment; it simply acts as the local repository for FIFO data.

The storage module may sleep waiting for resources (such as paged out data structures or buffers), but it should never sleep because of a full or empty FIFO, or for synchronization reasons. These conditions should be checked and handled by the policy module.

2.1.2.6 Preliminary NIDL specification

```
/*
 *  Remote FIFO Interface Definition
 */

interface rfifo {

/*
 * FIFO states
 */
typedef long enum { FIFO_NULL,
                    FIFO_SERVER,
                    FIFO_CLIENT,
                    FIFO_LENDER,
                    FIFO_BORROWER } fifo_state_t;

/*
 * The server send this structure to the client node in
 * order to establish the server state in the client node.
 */
typedef struct lend_info {
        unsigned long li_trcnt;         /* total readers count */
        unsigned long li_twcnt;         /* total writer count */
        unsigned short li_rcnt;         /* client read count */
        unsigned short li_wcnt;         /* client write count */
```

```
        fifo_state_t li_state;          /* server state */
} lend_info_t;

/*
 *  Return status for RPC calls
 */
typedef enum {
        OK,
        WOULDBLOCK,
        ERROR,
        DRAINED,
        AGAIN
} rf_status_t;

/*
 *  Remote FIFO syscall-like RPC routines
 */

void server_rfifo_reopen(
[ in ]          handle_t        h,
[ in ]          vnode_addr_t    vp,
[ in ]          long int        open_flag,
[ in ]          nodeno_t        reopening_node,
[ in ]          long int        ext,
[ out ]         pt_uerror_t     *rc,
[ out ]         unsigned32      *st
);
{
        struct vfs *rfifo_vfs;

        do locking
        rc = check_fifo_state(vp, FIFO_SERVER);
        /*
         * return to caller, try again
         */
        if (*rc == EBUSY) {
                unlock;
                exit;
        }
        *rc = local_fifo_reopen(vp,
                                reopening_node,
                                open_flag);
}
```

```
void                                                                    |
server_rfifo_read(
        [ in ]          handle_t        h,                              |
        [ in ]          vnode_addr_t    fusion_vp,                      |
        [ in ]          int             flags,                          |
        [ in ]          char            *data_buf,                      |
        [ in ]          int             *cnt,                           |
        [ in ]          long            offset,                         |
        [ in ]          pt_uerror_t     *uerrorp,                       |
        [ in ]          error_status_t  *st)                            |
{
        get kernel lock                                                 |
        *uerrorp = check_fifo_state(fusion_vp, FIFO_SERVER);            |
        if (*uerrorp == EBUSY) {                                        |
           unlock;                                                      |
           return;                                                      |
        }                                                               |
        setup iovec for reading                                         |
        initialize uio structure
        find the FIFO descriptor associated with the remote client      *
        get base system vnode pointer from FIFO descriptor              |
        VNOP_RDWR(base_vp, rw, flags, &uio, ext, vinfo)
        release kernel lock                                             |
}

void
server_rfifo_write(
        [ in ]          handle_t        h,                              |
        [ in ]          vnode_addr_t    fusion_vp,                      |
        [ in ]          int             flags,                          |
        [ in ]          char            *data_buf,                      |
        [ in ]          int             *cnt,                           |
        [ in ]          long            offset,                         |
        [ out ]         pt_uerror_t     *uerrorp,                       |
        [ out ]         error_status_t  *st)                            |
{
        get kernel lock                                                 |
        *uerrorp = check_fifo_state(fusion_vp, FIFO_SERVER);            |
        if (*uerrorp == EBUSY) {                                        |
           unlock;                                                      |
           return;                                                      |
        }                                                               |
        setup iovec for writing                                         |
```

```
        initialize uio structure
        find the FIFO descriptor associated with the remote client     *
        get base system(storage) vnode pointer from FIFO descriptor
        VNOP_RDWR(base_vp, rw, flags, &uio, ext, vinfo)
        release kernel lock                                            |
}

void
server_rfifo_close(
        [ in ]          handle_t        h,                             |
        [ in ]          vnode_addr_t    fusion_vp,                     |
        [ in ]          int             flag,                          |
        [ in ]          nodeno_t        closing_node,                  |
        [ out ]         pt_uerror_t     *rc,                           |
        [ out ]         error_status_t  *st                            |
{
        get kernel lock                                                |
        *uerrorp = check_fifo_state(fusion_vp, FIFO_SERVER);           |
        if (*uerrorp == EBUSY) {                                       |
           unlock;                                                     |
           return;                                                     |
        }                                                              |
        find the FIFO descriptor associated with the remote client
        *rc = local_fifo_close(fusion_vp, flags, closing_node);        |
        release kernel lock                                            |
}                                                                      |

[ idempotent ] void server_rfifo_getattr(                              |
        [ in ]          handle_t        h,                             |
        [ in ]          vnode_addr_t    vp,                            |
        [ out ]         pt_vattr_t      *vattrp,                       |
        [ out ]         pt_uerror_t     *uerrorp,                      |
        [ out ]         unsigned32      *st                            |
);                                                                     |
{
        get kernel lock                                                |
        VNOP_GETATTR(vp, vattrp)                                       |
        release kernel lock                                            |
}

/*
 * This RPC call is used to effect protocol state transition
 * in the Policy Module lending scheme at client node
```

```
 */
[ idempotent ] fifo_to_client(                                          |
        [ in ]          handle_t        h,                              |
        [ in ]          nodeno_t        curserver,                      |
        [ in ]          vnode_addr_t    fvp,                            |
        [ in, length_is(*cnt) ]char     data_buf[REMOBJ_MAXBUF],        |
        [ in, out ]     long            *cnt,                           |
        [ in ]          lend_info_t     *lend_info_ptr,                 |
        [ out ]         pt_uerror_t     *rc,                            |
        [ out ]         unsigned32      *st                             |
);                                                                      |
{

        search the FUSION vfs to find the vnode that                    |
        has the server fusion vnode                                     |
        if (CLIENT state) {
                allocate an inode from client pipedev
                get a base system vnode from the inode
                set the FIFO state in BORROWER state
                update the read/write counts in the FIFO descriptor     |
        }
        if (BORROWER state) {
                set up the local uio structure
                flags |= FNODELAY
                VNOP_RDWR(base_vp, rw, flags, &uio, ext, vinfo)
        }
        if (the server is in CLIENT state) {                            |
                change the client to be the SERVER state                |
                if (total read == 0)                                    |
                        VNOP_CLOSE(base_vp, FREAD, vinfop);             |
                if (total write == 0)                                   |
                        VNOP_CLOSE(base_vp, FWRITE, vinfop);            |
        }                                                               |
        wakeup any reader/writer for the FIFO lending to finish.        |
}

} /* end of interface rfifo */
```

2.1.2.7 Preliminary header file describing storage module interface

```
/*
 *
 *  pobj.h - Pipe OBJect interface
 *
```

```
 *      These routines define the interface between the FUSION FIFO
 *      VFS's policy module and the vendor's FIFO storage module.
 */


/*
 *  The structure of the FIFO descriptor which is allocated
 *  for each FUSION FIFO, when they get accessed by remote hosts.
 */
struct rfifo_hdr {
        int fi_node;                /* client of the FIFO */            |
        caddr_t *fi_vnode_addr;     /* point to the base system vnode */|
        enum vtype fi_type;         /* type of object */                |
        int fi_clients;             /* total clients of the FIFO */     |
        int fi_trcnt;               /* total readers */                 |
        int fi_twcnt;               /* total writers */                 |
        unsigned long fi_state;     /* FIFO state */                    |
        int fi_wait;                /* event list of processes waiting on */|
                                    /* FIFO data lending */             |
        unsigned long fi_flag;      /* flag accessible to FIFO */       |
        struct rfifonode *rfifo;    /* point to FIFO descriptor list */ |
}

struct rfifo_data {
        struct rfifo_data *next_rfifo;  /* next descriptor */
        struct rfifo_data *prev_rfifo;  /* previous descriptor */
        int node;                       /* client node number */
        short rfifo_rcnt;               /* read count */                *
        short rfifo_wcnt;               /* write count */
};                                                                      *

/*
 * the value of fi_flag                                                 |
 */
#define O_RDWR      0001                                                |
#define O_NONBLOCK  0002                                                |
#define FINOREAD    0003            /* unread pipe data */              |

/*                                                                      *
 *     The FUSION FIFO reopen routine either does the reopen of         |
 *     the FUSION vnode lcoally or create a local vnode for the remote  |
 *     FIFO and send RPC request to the server to do the reopen.        |
 *                                                                      |
```

```
 *      Input parameters:                                                    |
 *            *vfsp          pointer to the FUSION vfs                      |
 *            *reopen_data   contains the FUSION vnode and the node number  |
 *            **fvp          pointer to a pointer to the FUSION vnode       |
 *            open_flag      FREAD/FWRITE                                   |
 *                                                                          |
 *      Output parameter:                                                   |
 *            **fvp          the FUSION vnode                               |
 *                                                                          |
 *      Return value:                                                       |
 *            0              success                                        |
 *                           EBUSY                                          |
 *                           ENOMEM                                         |
 *                                                                          |
 */
fusion_fifo_reopen(struct vfs  *vfsp,                                       |
                   caddr_t *reopen_data,                                    |
                   struct  vnode **fvp,                                     |
                   int     open_flag,                                       |
                   caddr_t *vinfo)                                          |
{

loop:                                                                       |
        get the server vnode and server node number from reopen_data        |
        /*
         * if the vnode has returned to its originating
         * site, the reopen will be local
         */
        if (server node == cursite) {
              fifo_reopen(server_vp,                                        |
                          server_node,                                      |
                          open_flag,                                        |
                          vinfop);                                          |
              *fvp = server_vp;                                             |
              return rc;                                                    |
        }

        /*
         * build a local FUSION vnode for the remote FIFO
         */
        request = CLIENT;                                                   |
        vp = fusion_vnode(fvfsp,
                          server_vp,
```

```
                        server_node,
                        FIFO_CLIENT,
                        request,
                        obj_type)
        if (vp == NULL)
                rc = ENOMEM;
                goto out;
        }
        /*
         * performance the server node reopen
         */
        rc = ffifo_remote_reopen(reopen_data,
                        open_flag,
                        vinfop)
        if (rc == EBUSY)
                goto loop;
out:
        if (!remote open ok) {
                VNOP_RELE(vp)
                vp = NULL;
        }
        *fvp = vp;
        return rc;
}

/*
 *      This is the FUSION read/write routine,
 *      if the FUISON vnode is remote then RPC to the server to
 *      do the read/write.
 *
 *      Input parameters:
 *            *vp         pointer to the FUSION vnode
 *            rw          UIO_READ or UIO_WRITE
 *            *uiop       pointer to uio struct
 *
 *      Output parameters:
 *            none
 *
 *      return value:
 *             0          success
 *                        EBUSY
 *
 */
```

```
fusion_fifo_rdwr(struct vnode *vp,
                 enum uio_rw rw,
                 int flags,
                 struct uio *uiop)
{
loop
         get the pointer to the FIFO descriptor
         check_fifo_state(vp, 0)
         if (FIFO is local) {
               get the base vnode pointer
               VNOP_RDWR(base_vp, rw, flags, uiop, vinfop)
         }
         else {
                rc = ffifo_remote_rdwr(vp, rw, flags, uiop)
                if (rc == EBUSY)
                    goto loop;
         }
         return rc;
}

/*
 *     This routine does the close of the FUSION vnode.
 *     If the FUISON vnode is remote then RPC to the server to
 *     do the close.
 *
 *     Input parameters:
 *           *vp        pointer to the FUSION vnode
 *           flags      FREAD/FWRITE
 *
 *     Output parameters:
 *           none
 *
 *     return value:
 *            0         success
 *                      EBUSY
 *
 */
fusion_fifo_close(struct vnode *vp,
                  int flags)
{

loop:
        get the pointer to the FIFO descriptor
```

```
        if (FIFO is local)
              local_fifo_close(vp, flags, cursite)
        else {
               rc = ffifo_remote_close(vp, flags, cursite)
               if (rc == EBUSY)
                   goto loop;
        }
        return rc
}

/*
 *     This is the client node FIFO read/write routine
 *
 *     Input parameters:
 *           *vp        pointer to the FUSION vnode
 *           rw         UIO_READ or UIO_WRITE
 *           *uiop      pointer to uio struct
 *
 *     Output parameters:
 *           none
 *
 *     return value:
 *            0         success
 *                      EBUSY
 *
 */
ffifo_remote_rdwr(struct vnode *vp,
                  enum uio_rw,
                  int flag,
                  struct uio      *uiop)
{
        get the node number and the FUSION vnode of the server
        calculate the total fifo data in uio structure
        if (rw == UIO_READ) {
           do {
                num_read = MIN(total_data, MAX_BLK_SZ)
                size = num_read
                /*
                 * RPC call to read remote fifo data into local buffer
                 */
                server_rfifo_read(h,
                                  server vnode,
                                  flags,
```

```
                        loc_buf,
                        num_read,
                        offset,
                        uerror,
                        st)
        /*
         * check return status
         */
        if (st != rpc_s_ok || uerror == EBUSY)
           return uerror;
        rcnt = uiomove(loc_buf, num_read, rw, uiop)
        total_data -= num_read
  } while (num_read == size && total_data > 0)
else if (rw == UIO_WRITE) {
        do {
        /*
         * save offset before uiomove change it
         */
        save_offset = uiop->uio_offset
        num_written = MIN(total_data, MAX_BLK_SZ)
        size = num_written
        /*
         * We need to restore the uio struct, if the
         * writer blocked or RPC failed
         */
        uiosave(uiop, uiosave_buf);
        wcnt = uiomove(loc_buf, num_written, rw, uiop)
        if (wcnt < 0)
                return(wcnt)
        /*
         * RPC call to write data into remote open fifo
         */
        server_rfifo_write(h,
                        server vnode,
                        flags,
                        loc_buf,
                        num_written,
                        write_offset,
                        uerror,
                        st)
        if (st != rpc_s_ok || uerror == EBUSY) {
               uiorestore(uiop, uiosave_buf)
               return uerror
```

```
                }
                total -= num_written
              } while(num_written == size && total_data > 0)
          }
}

/*
 *      This is the client node FIFO close routine
 *
 *      Input parameters:
 *            *vp         pointer to the FUSION vnode
 *            int         flag
 *            int         closing_node
 *
 *      Output parameters:
 *            none
 *
 *      return value:
 *             0          success
 *
 */
ffifo_remote_close(
         vnode_addr_t vp,
         int flag,
         caddr_t vinfo)
{
         find the FUSION vnode pointer
         get the node number of the server
         server_rfifo_close(h,
                            server_vp,
                            open_flag,
                            cursite,
                            &rc,
                            &st)
         return rc
}

/*
 *      This is the client node FIFO reopen routine
 *
 *      Input parameters:
 *            *reopen_data        pointer to the FUSION vnode
 *            flag                FREAD/FWRITE
```

```
 *                                                                    |
 *     Output parameters:                                             |
 *           none                                                     |
 *                                                                    |
 *     return value:                                                  |
 *            0          success                                      |
 *                       EBUSY                                        |
 *                                                                    |
 */                                                                   |
ffifo_remote_reopen(caddr_t *reopen_data,                             |
                    int flag)                                         |
{                                                                     |
      get the server vnode and server node number from reopen_data    |
      server_rfifo_reopen(h,                                          |
                          server_vp,                                  |
                          flag,                                       |
                          cursite,                                    |
                          &rc,                                        |
                          &st)                                        |
      if (st != rpc_s_ok) {                                           |
           if (rc != EBUSY)                                           |
             rc = EIO                                                 |
      }                                                               |
      return rc                                                       |
}                                                                     |

/*                                                                    |
 *     This is the FUSION close routine at server node                |
 *                                                                    |
 *     Input parameters:                                              |
 *           *vp          pointer to the FUSION vnode                 |
 *           flag         FREAD/FWRITE                                |
 *           closing_node the node that request the close operation   |
 *                                                                    |
 *     Output parameters:                                             |
 *           none                                                     |
 *                                                                    |
 *     return value:                                                  |
 *            0          success                                     |
 *                                                                    |
 */                                                                   |
local_fifo_close(struct vnode vp,                                     |
                 int flag,                                            |
```

```
                closing_node)
{
      get the FIFO descriptor pointer
      get the base vnode pointer
      search the client list to find the client that requests the close ops
      if (not found)
         return ENXIO
      if (FREAD) {
         decrement the total read count of this client
         decrement the total read count of the FUSION FIFO vnode
         if (total read count of the vnode == 0)
            VNOP_CLOSE(base_vp, FREAD, vinfop)
      }
      if (FWRITE) {
         decrement the total write count of this client
         decrement the total write count of the FUSION FIFO vnode
         if (total write count of the vnode == 0)
            VNOP_CLOSE(base_vp, FWRITE, vinfop)
      }
      if (total read count and write count of this client == 0) {
         delete this client from the list
         decrement the total count of the client
         free up the base vnode if this is the last client on the list
      }
      if (all the readers/writers are on the same client node)
         rfifo_lend(base_vp, vp, client_node)
      return rc
}

/*
 *     This is the FUSION reopen routine at server node
 *
 *     Input parameters:
 *           *server_vp          pointer to the FUSION vnode
 *           node                the node requesting the reopen
 *           flag                FREAD/FWRITE
 *
 *     Output parameters:
 *           none
 *
 *     return value:
 *            0         success
 *
```

```
 */
local_fifo_reopen(struct vnode *server_vp,
                  int node,
                  int flag)
{
      get the FIFO descriptor pointer
      get the base vnode pointer
      search the client on the client list
      if (not found)
         add the new client to the list
      if (FREAD) {
         increment the total read count of this client
         increment the total read count of the FUSION FIFO vnode
      }
      if (FWRITE) {
         increment the total write count of this client
         increment the total write count of the FUSION FIFO vnode
      }
      return rc
}


/*
 * This routine called from local_fifo_close() at server node
 * to invoke the FUSION FIFO lending protocol.
 *
 * Input parameters:
 *       *vp        base system vnode
 *       *fvp       the FUSION vnode at server node
 *       node       the node that will be the FIFO new storage node
 *
 * Output parameters:
 *       rc         0  success.
 *                  -1 fail.
 *
 */
rfifo_lend(vp, fvp, node)
        struct vnode   *vp;          /* base system vnode */
        vnode_addr_t   *fvp;         /* FUSION vnode at server node */
        int            node;
{
        get kernel lock
        set the FIFO state to FIFO_CLIENT
```

```
        initialize the struct of lend_info which will get send to
        new storage node
        do {
               setup the uio struct
               VNOP_RDWR(base_vp, rw, flags, &uio, ext, vinfo)
               num_read -= uio.uio_resid;
              /*
               * Push any data to the borrower, along with an
               * indication as to whether we're done or not.
               */
              err = client_push_fifo(node,
                             fvp,
                             pipe_buf,
                             &num_read,
                             &lend_info)
              if (err) {
                 rc = -1;
                 unlock
              }
              else rc = 0;
         } while(fifo_state == FIFO_LENDER)
         fifo_state = FIFO_CLIENT
         wake up all the readers/writers that are waiting for
         FIFO lending to finish.
         return rc
}

/*
 * This routine called from rfifo_lend()
 * to push the FUSION FIFO data to new storage node.
 *
 * Input parameters:
 *       node           the node to which the FIFO data get push
 *       *vp            base system vnode
 *       *data_buf      contains the FIFO data
 *       *cnt           amount of the data get pushed
 *       *lend_info_ptr pointer to a struct that has the server FIFO info
 *
 * Output parameters:
 *       *cnt
 *
 * return value:
 *       0              success
```

```
 *                      EIO                                                |
 *                                                                         |
 */
client_push_fifo(int node,                                                 |
                 struct vnode *vp,                                         |
                 char *data_buf,                                           |
                 int *cnt,                                                 |
                 lend_inf_t *lend_info_ptr)                                |
{
       get the remote node handle                                          |
       fifo_to_client(h,                                                   |
                      cursite,                                             |
                      vp,                                                  |
                      data_buf,                                            |
                      cnt,                                                 |
                      lend_info_ptr,                                       |
                      &err,                                                |
                      &st)                                                 |
       if (st != rpc_s_ok) {                                               |
           err = EIO                                                       |
       }                                                                   |
       return err                                                          |
}                                                                          |

/*                                                                         |
 * This routine allocates a FUSION vnode and the FUSION FIFO descriptor.|
 * The FIFO descriptor contains all the necessary information about    |
 * the remote FIFO if there is one exists, and two pointers which      |
 * point to the base system vnode and a list of remote clients of the FIFO.|
 *                                                                         |
 * Input parameters:                                                       |
 *       *fvfsp     pointer to the FUSION vfs.                            |
 *       *vp        pointer to the FUIOSN vnode if remote,                |
 *                  or the base vnode if local.                           |
 *       node       node number of the remote FIFO, or cursite if local.|
 *       fifo_state FIFO_SERVER/FIFO_CLIENT                               |
 *       request    either from the client node or the server node.       |
 *       obj_type   type of the remote object of the vnode.               |
 *                                                                         |
 * Output parameters:                                                      |
 *       none                                                              |
 *                                                                         |
 * Return value:                                                           |
```

```
 *       a pointer to the FUSION vnode/gnode.                                |
 *                                                                           |
 */                                                                          |
fusion_vnode(struct vfs *fvfsp,                                              |
             vnode_addr_t *basevp,                                           |
             int node,                                                       |
             unsigned long fifo_state,                                       |
             char request,                                                   |
             enum vtype obj_type)                                            |
{                                                                            |
        if (request from client) {                                           |
            find the FUSION vnode from the FUSION VFS
            if (found) {                                                     |
                  return fusion vp;                                          |
            }
        } else {                                                             |
                  allocate memory space for gnode and the FIFO descriptor|
                  if (state == FIFO_SERVER) {                                |
                        initialize read/write count in the FIFO descriptor|
                  }                                                          |
                  if (state == FIFO_CLIENT) {                                |
                        initialize read/write count in the FIFO descriptor|
                  }                                                          |
                  connect the FUSION vnode to base vnode                     |
                  thru the FIFO descriptor                                   |
                  install the FUSION vnode ops                               |
                  /*                                                         |
                   * allocate the FUSION vnode                              |
                   */                                                        |
                  rc = vn_get(fvfsp, gnp, &vp);                              |
                  if (rc != 0) {                                             |
                        xmfree(space allocated for gnode)                    |
                        return base vnode;                                   |
                  }                                                          |
                  else return (FUSION vnode)                                 |
        }                                                                    |
}
```

2.1.3 Remote Sockets

This section of the **FUSION** Design Specification describes the operation of remote sockets in the **FUSION** environment. It explains the data structures needed, the routines that must be changed and the code that must be added. It explains the design of the remote socket calls and how the remote socket client/server interaction takes place.

The Berkeley Socket abstraction defines an interface to interprocess communication, with a socket as an endpoint of communication. There are two socket domains commonly provided. The UNIX domain sockets is an IPC mechanism similar to a bidirectional pipe. The Internet domain provides an interface to the networking services of TCP/IP. Other domains to support other networking services such as OSI could realistically be expected in the near future.

With **FUSION**'s remote processing capabilities, a mechanism is needed to allow processes to transparently use sockets controlled on a remote node. This is important, because the underlying network protocols operate below the RPC layer, and in general do not provide mechanisms to redirect communication to another node. The **FUSION** remote socket support will allow a process with open sockets to move to or be created on another node, without any support from the underlying network protocols (e.g., TCP/IP).

2.1.3.1 Overall Design

The **FUSION** remote socket service is available for both datagram (SOCK_DGRAM) and connection oriented services (SOCK_STREAM) for both UNIX domain (AF_UNIX) and Internet domain (AF_INET) protocol families.

An additional flag bit SS_ISREMOTE is defined for the socket state field. If this bit is set to one, the socket is an instantiation of a socket being used on this node that is controlled at a remote node. A user can only create sockets on the local node. A remote socket is only created if a process is created on or moved to another node while a socket is open.

When a remote socket is created, the associated file operations table of the file structure is filled in with remote socket client routines. These bundle the requests and ship them to the node where the socket is controlled, using the DCE RPC mechanisms. On the controlling node these RPCs are handled by a server thread. The requests are serviced on that node, and any result or error indication is returned to the client code. If a particular kernel implementation does not support the file operations implementation, very small hooks into the base kernel will be added within each system call to intercept the system call as soon as it determines the file descriptor is a socket.

Since it is possible to access sockets through special system calls in addition to regular file operations, the **FUSION** design requires minor modifications to these calls. These changes detect if an access is to a socket controlled on a remote node. If so,

the corresponding request is shipped to the controlling node via an RPC mechanism. There a server routine handles the access, and ships any result or error indication to the accessing node.

The **FUSION** remote socket service also provides support for the extended file operations "fo_prepare_for_export" and "fo_reopen". These operations are part of the general file reopen mechanisms discussed in Section 2.1.6. They are used when a process is created on or migrates to another node while a socket is open. These routines support the cases of when the process has a local socket open, when a process has a remote socket open and goes to another remote node, and when a process has a remote socket open and goes to the original node controlling the socket.

The remote socket layer does not require special support for some of the standard file access functions. These include select(), fcntl(), and tracking usage counts of local and remote sockets. They are handled by the file operations above the socket layer.

2.1.3.2 Remote Socket Data Structures

The data structures needed to support remote sockets in **FUSION** are defined in this section.

2.1.3.2.1 Support for SS_REMOTE Flag

The SS_ISREMOTE bit is added to the defines for the so_state field of the socket structure. This is defined in the header file socketvar.h in most socket implementations.

```
#define SS_REMOTE 0x800 /* socket is controlled at a remote node */
```

The value chosen here was 0x800 because this value does not conflict with any BSD, AIX, or OSF. This value is only interpreted locally and the only requirement of this design is that it not conflict with other flag bits in the local implementation.

2.1.3.2.2 Support for Storing the Controlling Node of the Socket

When a socket is controlled at a remote node, the socket stores the internet address of the controlling node. This is done by an additional data structure called the remote socket handle. The socket structure is modified to contain a pointer to this structure. This is defined in the header file socketvar.h in most socket implementations.

```
caddr_t so_rsohandle;
```

To minimize impact on the base system, this field should be overlaid with the so_pcb field in the standard socket structure. The so_pcb field is not used when the socket is remote. Overlaying this avoids changing the socket structure and thus avoids changes to kernel diagnostic tools.

2.1.3.2.3 rso_reopendata Structure

The rso_reopendata structure is returned by rso_prep_export. It contains information that can be used to uniquely identify an open file from a remote node. This information consists of an identifier to locate the file table entry of this socket on the controlling node (rso_re_rfileid), and a unique ID (rso_re_uniqueid) that can be used to insure that the file table entry is still valid for this socket. The identifier of the file structure is provided by the fo_prep_for_export file operation and is passed to the fo_reopen fileop. It is usually the kernel virtual address of that file structure, but can be anything significant to the generating host.

```
struct rso_reopendata {
        int rso_re_rfileid;     /* ID of file structure on remote node */
        int rso_re_uniqueid;     /* Unique ID to detect reboots */
}
```

2.1.3.2.4 rso_handle Structure

The rso_handle structure is used to refer to a remote socket on another site. It contains RPC handle information (rsh_rpchandle), file reopen data (rsh_reopendata), and socket information from the controlling node, including the domain (e.g., AF_UNIX or AF_INET) of the socket (rsh_domain), and the type (e.g., SOCK_STREAM or SOCK_DGRAM) of the socket (rsh_type).

```
struct rso_handle {
        handle_t rsh_rpchandle;                 /* RPC handle */
        struct rso_reopendata rsh_reopendata; /* file reopen data */
        int rsh_domain;                         /* domain of socket */
        int rsh_type;                           /* type of socket */
}
```

2.1.3.3 Functions

The functions used in the **FUSION** remote socket design fall into two categories, which are file operations for remote sockets and extensions to socket specific system calls along with the corresponding RPC server routines. In addition, subroutines to support these functions are described.

2.1.3.3.1 File Operations for Remote Sockets

Typical file operation provided are fo_rw(), fo_ioctl(), fo_select(), fo_close(), and fo_fstat(). Some implementations may provide fo_read() and fo_write() rather than fo_rw(), and some implementations may not provide fo_stat. In addition to these standard file operations, **FUSION** has defined two additional file operations, which are fo_prepare_for_export() and fo_reopen() to support remote process operations, as described in Section 2.1.6.

2.1.3.3.1.1 rso_rw

The routine rso_rw() is called for doing reads and writes to a socket. It is implemented by directly calling soreceive() or sosend() depending on whether the

operation is a read or a write. This adapts easily into the FUSION architecture, which provides remote versions of soreceive() and sosend(), called rsoreceive() and rsosend(). Thus rso_rw() is a clone of soo_rw(), except that calls to soreceive() are replaced with rsoreceive(), and calls to sosend() are replaced with rsosend().

Some file operations implementations such as OFS/1 provide separate read and write fileops, called soo_read() and soo_write(). Remote versions of these are created just as easily. The routine rso_read() is identical to soo_read() except that calls to soreceive() are replaced with calls to rsoreceive(), and rso_write() is identical to soo_write() except that calls to sosend() are replaced with calls to rsosend().

The routines rsoreceive() and rsosend() are described later, in sections 2.1.3.3.2.10 and 2.1.3.3.2.11.

2.1.3.3.1.2 rso_ioctl

The rso_ioctl file op is called when an ioctl operation is performed on a remote socket. Ioctls require numerous RPCs because the RPC protocol cannot merely deal with ioctl arguments as a series of bytes to be copied, but must recognize the underlying types being dealt with by the ioctl. This routine rso_ioctl will package up the known ioctls and send them to the remote socket server using the appropriate RPC for that ioctl type. Unknown ioctls will pass the data to the controlling node as a stream of bytes.

The following specifies the rso_ioctl file operation:

```
/*
 * rso_ioctl(fp, cmd, data)
 * struct file *fp;
 * int cmd;
 * caddr_t data;
 *
 * Abstract:
 *      remote socket ioctl file op
 *
 * Parameters
 *      fp: pointer to file structure being closed
 *      cmd: ioctl command being performed
 *      data: user data or pointer to user data
 *
 * Return Values:
 *      error indication if an error occurred.
 *
 * Algorithm:
 *        get socket pointer from file structure
 *        switch(cmd) {
 *        case FIONBIO:
```

```
*       case FIOASYNC:
*       case SIOCSHIWAT:
*       case SIOCSLOWAT:
*             copy in integer data from user space
*             call rso_ioctl_inint RPC with the following parameters:
*                   handle, fh, and f_flags from socket structure
*                   cmd passed as argument to routine
*                   data copied in from user
*                   pointer to status return variable
*             return status to caller
*
*       case SIOCSPGRP:
*             copy in unsigned long data from user space
*             call rso_ioctlinulong RPC with the following parameters:
*                   handle, fh, and f_flags from socket structure
*                   cmd passed as argument to routine
*                   data copied in.
*                   pointer to status return variable
*             return status to caller
*
*       case SIOCGHIWAT:
*       case SIOCGLOWAT:
*       case SIOCGATMARK:
*             call rso_ioctl_outint RPC with the following parameters:
*                   handle, fh, and f_flags from socket structure
*                   cmd passed as argument to routine
*                   pointer to int for return data
*                   pointer to status return variable
*             if successful copy out integer data to user
*             return status to caller
*
*       case FIONREAD:
*             call rso_ioctloutulong RPC with the following parameters:
*                   handle, fh, and f_flags from socket structure
*                   cmd passed as argument to routine
*                   pointer to unsigned long for return data
*                   pointer to status return variable
*             if successful copy out unsigned long data to user
*             return status to caller
*
*       case SIOCSARP:
*       case SIOCSARP_802_5:
*       case SIOCSARP_X_25:
```

```
*       case SIOCDARP:
*       case SIOCDARP_802_5:
*       case SIOCDARP_X_25:
*              copy in ARP data from user space into rpc_arpreq structure
*                           (byte count may be dependent on specific ioctl cmd)
*              call rso_ioctlinARP RPC with the following parameters:
*                     handle, fh, and f_flags from socket structure
*                     cmd passed as argument to routine
*                     data copied in.
*                     pointer to status return variable
*              return status to caller
*
*       case SIOCGARP:
*       case SIOCGARP_802_5:
*       case SIOCGARP_X_25:
*              call rso_ioctloutARP RPC with the following parameters:
*                     handle, fh, and f_flags from socket structure
*                     cmd passed as argument to routine
*                     pointer to rpc_arpreq data structure for return data
*                     pointer to status return variable
*              if successful copy out ARP data to user.  The byte count may
*                              be dependent on the particular ioctl cmd)
*              return status to caller
*
*       case SIOCGIFCONF:
*              call rso_ioctlIFCONF RPC with the following parameters:
*                     handle, fh, and f_flags from socket structure
*                     cmd passed as argument to routine
*                     max_index: maximum size of list based on size of
*                            local internal structure
*                     pointer to out_index value
*                     pointer to buffer for returned interface list
*                     pointer to status return variable
*              if successful copy out interface structure to user.  The
*                     out_index value indicates the size of the list.
*              return status to caller
*
*       case SIOCGIFFLAGS:
*       case SIOCGIFMTU:
*       case SIOCGIFREMMTU:
*              call rso_ioctloutIFshort RPC with the following parameters:
*                     handle, fh, and f_flags from socket structure
*                     cmd passed as argument to routine
```

```
*                       interface name string provided from caller
*                       pointer to short for return value
*                       pointer to status return variable
*               if successful copy out short data to user.
*               return status to caller
*
*       case SIOCGIFMETRIC:
*               call rso_ioctloutIFint RPC with the following parameters:
*                       handle, fh, and f_flags from socket structure
*                       cmd passed as argument to routine
*                       interface name string provided from caller
*                       pointer to int for return value
*                       pointer to status return variable
*               if successful copy out integer data to user.
*               return status to caller
*
*       case SIOCSIFMETRIC:
*               copy in integer data from user space
*               call rso_ioctlinIFint RPC with the following parameters
*                       handle, fh, and f_flags from socket structure
*                       cmd passed as argument to routine
*                       interface name string provided from caller
*                       integer data copied from user space
*                       pointer to status return variable
*               return status to caller
*
*       case SIOCSIFFLAGS:
*       case SIOCSIFMTU:
*       case SIOCSIFREMMTU:
*               copy in short data from user space
*               call rso_ioctlinIFint RPC with the following parameters
*                       handle, fh, and f_flags from socket structure
*                       cmd passed as argument to routine
*                       interface name string provided from caller
*                       short data copied from user space
*                       pointer to status return variable
*               return status to caller
*
*       case SIOCGIFADDR:
*       case SIOCGIFBRDADDR:
*       case SIOCGIFDSTADDR:
*       case SIOCGIFNETMASK:
*               call rso_ioctloutIFaddr RPC with the following parameters:
```

```
*                       handle, fh, and f_flags from socket structure
*                       cmd passed as argument to routine
*                       interface name string provided from caller
*                       pointer to sockaddr_t structure for return value
*                       pointer to status return variable
*               if successful copy out sockaddr_t structure to user.
*               return status to caller
*
*       case SIOCSIFADDR:
*       case SIOCSIFBRDADDR:
*       case SIOCSIFDSTADDR:
*       case SIOCSIFNETMASK:
*               copy in sockaddr_t structure from user.
*               call rso_ioctlinIFaddr RPC with the following parameters:
*                       handle, fh, and f_flags from socket structure
*                       cmd passed as argument to routine
*                       interface name string provided from caller
*                       sockaddr_t structure from user
*                       pointer to status return variable
*               return status to caller
*
*       case SIOCADDRT:
*       case SIOCDELRT:
*               copy in destination sockaddr_t, gateway sockaddr_t,
*                       and route flags short from user.
*               call rso_ioctlSIFRT RPC with the following parameters:
*                       handle, fh, and f_flags from socket structure
*                       cmd passed as argument to routine
*                       interface name string provided from caller
*                       destination sockaddr_t structure from user
*                       gateway sockaddr_t structure from user
*                       route flags from user
*                       pointer to status return variable
*               return status to caller
*
*       default:
*               copy in data in raw form
*               call rso_ioctlUNKOWN RPC with the following parameters:
*                       handle, fh, and f_flags from socket structure
*                       cmd parameter
*                       size of raw data
*                       raw data
*                       pointer to status return variable
```

```
*               return status to caller
*       }
*/
```

The following specifies the server side behavior for the rso_ioctl:

```
/*
* rso_ioctl*_server(h, fh, f_flags, cmd, RPC-specific data, st)
* handle_t h;
* rso_rfh_t fh;
* int f_flags;
* int cmd;
* [declarations for RPC-specific data]
* int *st;
*
* Abstract:
*     server for rso_listen RPC
*
* Parameters
*     h: RPC handle for socket
*     fh: handle for remote file structure (local on server node)
*     f_flags:  f_flags field from client structure
*     cmd: ioctl command
*     RPC-specific data
*
* Return Values:
*     return indication is made via st variable
*
* Local Variables
*
* Algorithm
*     Put parametric data into locally valid form
*     put cmd into locally valid format
*     call soo_ioctl(fp, cmd, data)
*     *st = u.u_error
*     return
*
*/
```

A more general solution for unrecognized ioctls may be appropriate.

2.1.3.3.1.3 rso_select

Support for select on remote special files is specified in section 2.1.4. Special support for select on sockets is not required.

2.1.3.3.1.4 rso_close

The rso_close file op is called when a remote socket is closed for the last time on that node.

```
/*
 * rso_close(fp)
 * struct file *fp;
 *
 * Abstract:
 *      remote socket close file op
 *
 * Parameters
 *      fp: pointer to file structure being closed
 *
 * Return Values:
 *      error indication if an error occurred.
 *
 * Algorithm:
 *        remove remote file handle structure
 *        remove local copy of socket
 *        indicate there is no socket associated with this file block
 *        return error indication if any, or 0 if no error
 *
 */
```

There is no need to provide an rso_remoteClose RPC because calling of file specific close routines is done at the file token layer. See section 2.1.5.

2.1.3.3.1.5 rso_fstat

The rso_fstat file op is called when an fstat operation is performed on an open file descriptor that refers to a socket.

```
/*
 * rso_fstat(fp, ub)
 * struct file *fp;
 * struct stat *ub;
 *
 * Abstract:
 *      remote socket fstat file op
 *
 * Parameters
 *      fp: pointer to file structure being closed
 *      ub: pointer to stat buffer
 *
 * Local Variables
 *      st: return status code
```

```
*
* Return Values:
*     error indication if an error occurred.
*
* Algorithm:
*       get socket pointer from file block (fp->f_data)
*       clear *ub
*       call rso_remoteStat RPC with the following arguments
*              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*              remote file structure handle (so->rso_handler.rsh_reopendata)
*              f_flags field from file structure (fp->f_flags)
*              ub pointer to a stat buffer (ub)
*              pointer to st (&st)
*       return
*
*       return error indication if any, or 0 if no error
*
*/
```

The following specifies the behavior of the server routine for the rso_remoteStat RPC:

```
/*
 * rso_stat_server(h, fh, f_flags, statbp, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * struct stat *statbp;
 * int *st;
 *
 * Abstract:
 *     server for rso_listen RPC
 *
 * Parameters
 *     h: RPC handle for socket
 *     fh: handle for remote file structure (local on server node)
 *     f_flags:  f_flags field from client structure
 *     backlog: maximum length of queue of pending connections
 *     st: pointer to variable in which to store error return value
 *
 * Return Values:
 *     return indication is made via st variable
 *
 * Local Variables
 *       struct file *fp;
 *       struct socket *so;
```

```
 *
 * Algorithm:
 *       fp = pointer file block from caller supplied handle
 *       call soo_stat(fp, statbp)
 *       return value from soo_stat
 *
 */
```

2.1.3.3.1.6 rso_prep_export

The file-op rso_prep_export() is called when a process with an open socket is being moved to another node. This routine is to prepare for the creation of a remote socket. A UNIX domain socket that is unbound, or a socket that is unconnected cannot be created remotely. First, a check is made to see if the socket is eligible for becoming remote. If it cannot, then an error is returned and the operation does not continue. If this check succeeds, rso_prep_export() gathers all info necessary to reopen the socket on the remote node. This includes the socket type, the socket options, the socket state, and an identifier that the controlling node can use to locate the socket structure efficiently.

```
/*
 * rso_prep_export(fp)
 * struct file *fp;
 * struct sock_reopen_handle *srohp;
 *
 * Abstract:
 *     remote socket prepare for export file op
 *
 * Parameters
 *     fp: pointer to file structure being closed
 *     srohp: pointer to a socket reopen handle
 *
 * Return Values:
 *     error indication if an error occurred.
 *
 * Algorithm:
 *       obtain socket pointer from file block
 *       if socket cannot be made remote
 *            return error indication to caller
 *       if there is no socket associated with this file block
 *            return error indication to caller
 *       store the following fields from socket structure
 *            into the socket reopen handle structure:
 *                so_type
 *                so_proto->pr_type
 *                so_proto->pr_domain
```

```
*       store a unique identifier (e.g., the kernel address of the file
*                 structure) and a version ID in the reopen handle
*       return 0 to the caller
*
*/
```

2.1.3.3.1.7 rso_reopen()

The final step in socket migration takes place when the extended file operation fo_reopen is called at the remote node. When a process moves with an open socket, rso_reopen() will be called. First, rso_reopen() checks to see if the socket has migrated back to the original node. If this is the case, rso_reopen() just returns. If this is not the case, the support routine rso_create() is called to create a new socket.

```
/*
 * rso_reopen(fp, srohp)
 * struct file *fp;
 * struct rso_handle *srohp;
 *
 * Abstract:
 *     remote socket reopen file op
 *
 * Parameters
 *     fp: pointer to file structure being closed
 *
 * Return Values:
 *     0 if successful
 *     error indication if an error occurred.
 *
 * Local Variables
 *     so *socket;
 *
 * Algorithm:
 *       if this is the node where the socket is controlled
 *            return 0
 *       call rso_create(srohp->rsh_domain, &fp->f_data, rsh->rsh_type, srohp)
 *       return value from rso_create()
 *
 */
```

The rso_create routine is used by the reopen code to create a socket structure which will be controlled on a remote client. The following specifies the rso_create() routine.

```
/*
 * rso_create(dom, aso, type, proto, rso_handle)
 * int dom;
 * struct socket **aso;
```

```
* int type;
* int proto;
* struct rso_handle *rso_handle;
*
* Parameters
*     dom: domain of socket
*     pointer to socket
*     type: type of socket (stream or datagram)
*     proto: prototype
*     rso_handle: remote socket handle to remote node
*
* Return Values:
*    0 is successful
*    error code otherwise
*
* Abstract:
*     Identical to socreate() with the following exceptions.
*             the so_proto field of the socket is not set.
*             the pr_usrreq() routine for PRU_ATTACH is not called.
*             the so_state field of the socket has the SS_ISREMOTE bit set.
*
*/
```

2.1.3.3.2 Socket System Call Extensions

The socket system calls do not go through a structure similar to the file operations table. Consequently, another approach is required to extend those system calls to support remote socket operations. The **FUSION** remote socket design provides two mechanisms for extending these calls. The approach used for a particular implementation depends on the underlying capabilities of the base system.

Some system such as OSF/1 provide a mechanism to redirect system calls to an alternate entry point. If such a mechanism is available in the target system, this will be used. Each system call to be extended will have its entry point replaced with an extended version. The convention used in this specification for the name of the extended version of a standard socket system call is to prepend an 'x' to the name of the standard call. For example, the extended version of the connect() system call will be called xconnect(). This extended version of the system call detects whether the socket is local or remote. If it is local it calls the standard socket system call entry point. If it is remote, it calls a new remote version of the system call. The convention used in this specification for the name of the remote version of the system call is to prepend an 'r' to the name of the standard system call. Thus, the remote version of the connect() system call will be called rconnect().

For a typical socket system call, the following model is used for the extended system call:

```
/*
 * xsocketcall()
 *
 * Abstract:
 *     extended version of socketcall
 *
 * Parameters
 *      same parameters as that of socketcall
 *
 * Return Values:
 *     same return value as that of socketcall, possibly
 *     enhanced with additional error codes for remote failures
 *
 * Algorithm:
 *       call getsock() to get pointer to file structure
 *       if getsock() failed
 *              return error
 *       get socket pointer from file structure
 *       if the socket is local to this node
 *            do anything necessary so socketcall sees the arguments
 *                        where it expects to find them
 *            call socketcall()
 *            return to caller
 *       call rsocketcall RPC
 *       return any error indication
 *       return to caller
 *
 */
```

The following model is used for the remote version of the system call:

```
/*
 * rsocketcall()
 *
 * Abstract:
 *     remote version of socketcall
 *
 * Parameters
 *      same parameters as that of socketcall
 *
 * Return Values:
 *     same return value as that of socketcall, possibly
 *     enhanced with additional error codes for remote failures
 *
 * Algorithm:
```

```
*       do required local node processing
*       call socketcall specific RPC
*       copy any return values to users
*       return any error indication
*       return to caller
*
*/
```

Other systems that do not support redirection of system calls will require minor hooks into the base system calls. This hook is placed at the beginning of the system call as soon as the socket associated with that system call has been found, before any processing of the call has been performed. The hook detects if the socket is remote, and if so calls the remote version of the system call that was described above.

Aside from the information provided by parameters to the system call, the remote client also provides the state of the f_flags field in the file structure associated with the open file descriptor. The socket server on the controlling node updates the flags field in the local file structure associated with the structure before performing the operation requested by the client. This allows file control operations performed on the client node to be seen by the socket server before the socket operation is performed.

It is a fundamental assumption of this design that it is possible for the kernel RPC server routine to call standard system call entry points at the local node, even though the server routine is executing in a kernel thread. The principal implications of this assumption is that it is possible to put the arguments into a location where the system call will find them without modification to that call, and that data transfers between kernel space and user space will still occur correctly. This assumption greatly simplifies the design as well as the specification of the design, because it means that the RPC server routines can call the standard entry points to provide the requested server. If in fact the kernel RPC does not support this assumption, the functions can still be implemented by cloning system call entry routines, changing them only to get the arguments from a location where they can locate them, and doing all data using kernel-to-kernel techniques.

The following model is used for the RPC server routines:

```
/*
 * rso_socketcall_server()
 *
 * Abstract:
 *     RPC remote server for rsocketcall
 *
 * Parameters
 *      Parameters are passed via the RPC mechanisms.  They
 *      typically consist of RPC control parameters, system
 *      call input parameters, and system call output parameters.
```

```
*
* Return Values:
*     The RPC server routines do not return explicitly values.
*     They return values via the [out] parameters of the RPC.
*
* Algorithm:
*       move parameters from RPC into location suitable in order
*               to call the socketcall system call
*       call the local version of socketcall
*       copy return values to RPC variables
*       indicate RPC status
*       return to RPC, which transfers control back to client node
*
*/
```

The following subsections identify the various socket system calls, and provide details about the remote version of the call and about the server side actions invoked via the RPC.

2.1.3.3.2.1 Socket and Socketpair

Because **FUSION** only creates remote sockets as a result of remote processing operations, the socket system calls socket() and socketpair() are unaffected by the **FUSION** remote socket design. No extended or remote versions of these calls are provided.

2.1.3.3.2.2 Accept

The accept() system call requires special handling on the using node, because it creates a new socket as part of its normal operation. In the socket is controlled remotely, a new remote socket must be created on the using node as well as on the standard socket on the controlling node. This is accomplished using the file reopen mechanisms described above, and an additional nested RPC call.

The following specifies the raccept call:

```
/*
 * raccept(s, name, namelen)
 * int s;
 * caddr_t name;
 * int *anamelen;
 *
 * Abstract:
 *     remote version of accept system call
 *
 * Parameters
 *     NOTE: parameters are passed in an architecture dependent way
 *     s: file descriptor of socket on which to accept the connection
```

```
*     name: domain specific address of connecting entity
*     anamelen: pointer to length of name parameter
*
* Return Values:
*    -1 if connect operation failed.  In this case, errno is set to indicate
*             the error.
*       otherwise, raccept returns the file descriptor of a new socket
*       the address of remote entity is returned via the name parameter
*       the length of the address is returned via the anamelen parameter
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*       int fdes
*       sockaddr_t address
*       int addrlen, addrlensv;
*       int st;
*
* Algorithm:
*       if name != NULL
*              copy in anamelen parameter to addrlen
*              if copy in fails
*                     provide EFAULT error to caller
*                     return
*              addrlensv = addrlen
*       call getsock(s) to get a pointer to the file structure from
*                                                    file descriptor
*       if getsock() failed
*              provide error from getsock() to caller
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       call rso_accept RPC with the following parameters
*              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*              remote file structure handle (so->rso_handler.rsh_reopendata)
*              f_flags field from file structure (fp->f_flags)
*              pointer to fdes (&fdes)
*              pointer to address parameter (&addr)
*              pointer to address length parameter (&addrlen)
*              pointer to error return value (&st)
*       if an error occurred (st != 0)
*              provide error code to caller
*              return
*       if name != NULL
```

```
 *              if addrlen > addrlensv
 *                      addrlen = addrlensv
 *              copy out addrlen bytes from address to name
 *              copy out addrlen to anamelen
 */
```

The following specifies the server routine for the rso_accept RPC.

```
/*
 * rso_accept_server(h, fh, f_flags, fdes, addr, addrlen, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * int *fdes;
 * rpc_sockaddr_t *addr;
 * int *addrlen;
 * int *st;
 *
 * Abstract:
 *     server for rso_accept RPC
 *
 * Parameters
 *     h: RPC handle for socket
 *     fh: handle for remote file structure (local on server node)
 *     f_flags:  f_flags field from client structure
 *     fdes: pointer to file descriptor for new socket
 *     addr: address of connection endpoint returned by accept
 *     addrlen: length of addr parameter
 *     st: pointer to variable in which to store error return value
 *
 * Return Values:
 *     fdes is filled in with the file descriptor of the new socket
 *     addr is filled in with the address of the communication endpoint
 *     addrlen is filled in with the length of the addr parameter
 *     return indication is made via st variable
 *
 * Local Variables
 *       int fd;
 *       int st;
 *
 * Algorithm:
 *       fd = file descriptor for file structure referred to in fh.
 *              This may require some setup.
 *       put fd where accept expects its s argument
 *       put addr where accept expects its name argument
```

```
*       put addrlen where accept expects its anamelen argument
*
*       call accept()
*
*       call f_prep_export(fp, &ofroh)
*       make an rpc handle
*       call rso_doreopen RPC with the following parameters
*              RPC handle
*              remote file structure handle (&ofroh)
*              address of status variable (&st)
*
*       if st == 0
*              put fd where close expects to find it
*              call close fd locally
*       return st
*/
```

The following specifies the server routine for the rso_doreopen RPC.

```
/*
 * rso_doreopen_server(h, ofroh, st)
 *
 * Abstract:
 *     server for rso_doreopen RPC
 *     NOTE:  The rso_doreopen RPC is called from the server
 *            routine of the rso_accept_server RPC and the
 *            rso_socketx RPC.  This server for the rso_doreopen
 *            RPC thus server runs on the client node that issued
 *            the accept() or bind() system call on the remote socket,
 *            but in a different thread
 *
 * Parameters:
 *     h: RPC handle for socket
 *     ofroh: reopen handle for remote file structure
 *     st:  pointer to status return
 *
 * Return Values:
 *     return indication is made via st variable
 *
 * Local Variables:
 *     struct file *fp
 *
 * Algorithm:
 *            find file block pointer from ofroh
 *            call f_reopen(fp, ofrop)
```

```
*            *st = return from f_reopen
*            return
*
*/
```

2.1.3.3.2.3 Bind

The bind system call requires special handling to handle cases where a process moves acquires some node dependent information, moves to another node, creates a new socket, and then tries to bind using that node dependent information (e.g., the TCP/IP address). This is implemented in the extended version of the bind call. If the local bind operation fails with EADDRNOTAVAIL, then a remote socket is created, and an rbind() operation is performed.

The following specifies the xbind() system call interface:

```
/*
 * xbind(s, name, namelen)
 * int s;
 * caddr_t name;
 * int namelen;
 *
 * Abstract:
 *     extended version of the bind system call
 *
 * Parameters
 *     s: file descriptor of socket to be bound
 *     name: domain specific name to bind socket
 *     namelen: length of name parameter
 *
 * Return Values:
 *     0 if bind operation succeeded
 *    -1 if bind operation failed.  In this case, errno is set to indicate
 *              the error.
 *
 * Local Variables
 *       struct rso_handle ofroh;
 *
 * Algorithm:
 *       call getsock(s) to get pointer to file structure
 *       if getsock() failed
 *              return error
 *       get socket pointer from file structure
 *       if the socket is local to this node
 *            bind()
 *            if (u.u_error == EADDRNOTAVAIL)
```

```
*            and (this process was created on another node)
*            and (name != NULL)
*            and (name != address of this node)
*            and (socket domain is AF_INET)
*            and (socket is SOCK_DGRAM or socket is SOCK_STREAM)
*                    set up RPC handle for user-supplied remote address
*                    ofroh.rsh_type = so->so_type
*                    ofroh.rsh_domain = so->so_proto->pr_domain
*                    call rso_socketx RPC with the following parameters
*                            RPC handle
*                            file reopen pointer (&ofroh)
*                            pointer to status variable (&st);
*                    if rso_socketx succeeded
*                            make s refer to new file descriptor
*                            close original socket
*                            rbind()
*                    else
*                            restore error to EADDRNOTAVAIL
*            return
*       rbind()
*       return
*/
```

The following specifies the server routine for the rso_socketx RPC:

```
/*
 * rso_socketx_server(h, fh, st))
 * handle_t h;
 * rso_rfh_t fh;
 * int *st;
 *
 * Abstract:
 *      server for rso_socketx RPC
 *
 * Parameters
 *      h: RPC handle for socket
 *      fh: handle for remote file structure (local on server node)
 *      st: success indication
 *
 * Return Values:
 *      return indication is made via st variable
 *
 * Local Variables
 *       struct file *fp;
 *       struct socket *so;
```

```
*       int stat
*
* Algorithm:
*       get a file block for an open pointer
*       open a local socket with socreate
*       fh->rso_opendata.rso_re_fileid = unique ID for file block (fp)
*       fh->rso_opendata.rso_re_fileid = version number
*       call f_prep_export(fp, fh)
*       call rso_doreopen RPC with the following parameters
*              RPC handle
*              remote file structure handle (fh)
*              address of status variable (&stat)
*       *st = stat
*/
```

The following specifies the rbind system call

```
/*
 * rbind(s, name, namelen)
 * int s;
 * caddr_t name;
 * int namelen;
 *
 * Abstract:
 *     remote version of bind system call
 *
 * Parameters
 *     NOTE: parameters are passed in an architecture dependent way
 *     s: file descriptor of socket to be bound
 *     name: domain specific name to bind socket
 *     namelen: length of name parameter
 *
 * Return Values:
 *     0 if bind operation succeeded
 *    -1 if bind operation failed.  In this case, errno is set to indicate
 *              the error.
 *
 * Local Variables
 *       struct file *fp;
 *       struct socket *so;
 *       rpc_sockaddr_t rpcname;
 *       int st;
 *
 * Algorithm:
 *       call getsock(s) to get pointer file structure from file descriptor
```

```
*       if getsock() failed
*             provide error from getsock() to caller
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       copy in name parameter from user space into rpcname;
*       call rso_bind RPC with the following parameters
*             RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*             remote file structure handle (so->rso_handler.rsh_reopendata)
*             f_flags field from file structure (fp->f_flags)
*             rpcname structure
*             name length parameter from caller
*             pointer to error return value (&st)
*       if an error occurred (st != 0)
*             provide error code to caller
*/
```

The following specifies the server routine for the rso_bind RPC:

```
/*
 * rso_bind_server(h, fh, f_flags, name, namelen, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * rpc_sockaddr_t name;
 * int namelen;
 * int *st;
 *
 * Abstract:
 *     server for rso_bind RPC
 *
 * Parameters
 *     h: RPC handle for socket
 *     fh: handle for remote file structure (local on server node)
 *     f_flags:  f_flags field from client structure
 *     name: domain specific name of remote socket
 *     namelen: length of name parameter
 *     st: pointer to variable in which to store error return value
 *
 * Return Values:
 *     return indication is made via st variable
 *
 * Local Variables
 *       int fd;
 *
 * Algorithm:
```

```
*       fd = file descriptor for file structure referred to in fh.
*            This may require some setup.
*       put fd where bind expects its s argument
*       put name where bind expects its name argument
*       put namelen where bind expects its anamelen argument
*       u.u_error = 0
*
*       call bind()
*
*       *st = u.u_error
*       return
*
```

2.1.3.3.2.4 Connect

The connect system call has no unusual states that require specialized handling on the client node. The following specifies the rconnect call:

```
/*
 * rconnect(s, name, namelen)
 * int s;
 * caddr_t name;
 * int namelen;
 *
 * Abstract:
 *     remote version of connect system call
 *
 * Parameters
 *     NOTE: parameters are passed in an architecture dependent way
 *     s: file descriptor of socket to be connected
 *     name: domain specific name of remote socket
 *     namelen: length of name parameter
 *
 * Return Values:
 *     0 if connect operation succeeded
 *    -1 if connect operation failed.  In this case, errno is set to indicate
 *             the error.
 *
 * Local Variables
 *       struct file *fp;
 *       struct socket *so;
 *       rpc_sockaddr_t rpcname;
 *       int st;
 *
 * Algorithm:
 *       call getsock(s) to get pointer file structure from file descriptor
```

```
*       if getsock() failed
*             provide error from getsock() to caller
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       copy in name parameter from user space into rpcname;
*       call rso_connect RPC with the following parameters
*             RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*             remote file structure handle (so->rso_handler.rsh_reopendata)
*             f_flags field from file structure (fp->f_flags)
*             rpcname structure
*             name length parameter from caller
*             pointer to error return value (&st)
*       if an error occurred (st != 0)
*             provide error code to caller
*/
```

The following specifies the server routine for the rso_connect RPC.

```
/*
 * rso_connect_server(h, fh, f_flags, name, namelen, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * rpc_sockaddr_t name;
 * int namelen;
 * int *st;
 *
 * Abstract:
 *     server for rso_connect RPC
 *
 * Parameters
 *     h: RPC handle for socket
 *     fh: handle for remote file structure (local on server node)
 *     f_flags:  f_flags field from client structure
 *     name: domain specific name of remote socket
 *     namelen: length of name parameter
 *     st: pointer to variable in which to store error return value
 *
 * Return Values:
 *     return indication is made via st variable
 *
 * Local Variables
 *       int fd;
 *
 * Algorithm:
```

```
*       fd = file descriptor for file structure referred to in fh.
*            This may require some setup.
*       put fd where connect expects its s argument
*       put name where connect expects its name argument
*       put namelen where connect expects its anamelen argument
*       u.u_error = 0
*
*       call connect()
*
*       *st = u.u_error
*       return
*
*/
```

2.1.3.3.2.5 Getpeername

The getpeername system call has no unusual states that require specialized handling on the client node. The following specifies the rgetpeername call:

```
/*
 * rgetpeername(fdes, asa, alen)
 * int fdes;
 * caddr_t asa;
 * int *alen;
 *
 * Abstract:
 *     remote version of getpeername system call
 *
 * Parameters
 *     NOTE: parameters are passed in an architecture dependent way
 *     fdes: file descriptor of socket to get peer name
 *     asa: address of user supplied address of buffer to return peer name
 *     alen: length of user supplied address buffer
 *
 * Return Values:
 *     0 if getpeername operation succeeded
 *    -1 if getpeername operation failed.  In this case, errno is set to
 *            indicate the error.
 *       The asa parameter is filled in with the name of the peer
 *       The alen parameter is filled in with the actual length of asa
 *
 * Local Variables
 *       struct file *fp;
 *       struct socket *so;
 *       rpc_sockaddr_t rpcname;
 *       int rpcnamelen;
```

```
*       int st;
*
* Algorithm:
*       call getsock(s) to get pointer file structure from file descriptor
*       if getsock() failed
*              indicate error from getsock()
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       copy in rpcnamelen from user alen parameter
*       if copy in fails
*               provide EFAULT error to caller
*       call rso_getpeername RPC with the following parameters
*              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*              remote file structure handle (so->rso_handler.rsh_reopendata)
*              f_flags field from file structure (fp->f_flags)
*              pointer to rpcname structure
*              pointer to rpcnamelen (&rpcnamelen)
*              pointer to error return value (&st)
*       if an error occurred (st != 0)
*              provide error code to caller
*              return
*       copy out rpcname structure to asa
*       if copy out fails
*              provide EFAULT error to caller
*       copy out rpcnamelen structure to alen
*       return
*/
```

The following specifies the server routine for the rso_getpeername RPC.

```
/*
 * rso_getpeername_server(h, fh, f_flags, name, namelen, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * rpc_sockaddr_t *name;
 * int *namelen;
 * int *st;
 *
 * Abstract:
 *     server for rso_getpeername RPC
 *
 * Parameters
 *     h: RPC handle for socket
 *     fh: handle for remote file structure (local on server node)
```

```
*      f_flags:  f_flags field from client structure
*      name: buffer for returning peer name
*      namelen: pointer to length of name parameter from user
*      st: pointer to variable in which to store error return value
*
* Return Values:
*      peer name is copied into name buffer
*      length of peer name is copied into namelen
*      return indication is made via st variable
*
* Local Variables
*       int fd;
*
* Algorithm:
*       fd = file descriptor for file structure referred to in fh.
*              This may require some setup.
*       put fd where getpeername expects its fdes argument
*       put name where getpeername expects its asa argument
*       put namelen where getpeername expects its asalen argument
*       u.u_error = 0
*
*       call getpeername()
*
*       *st = u.u_error
*       return
*
*/
```

2.1.3.3.2.6 Getsockname

The getsockname system call has no unusual states that require specialized handling on the client node. The following specifies the rgetsockname call:

```
/*
 * rgetsockname(fdes, asa, alen)
 * int fdes;
 * caddr_t asa;
 * int *alen;
 *
 * Abstract:
 *      remote version of getsockname system call
 *
 * Parameters
 *      NOTE: parameters are passed in an architecture dependent way
 *      fdes: file descriptor of socket to get socket name
 *      asa: address of user supplied address of buffer to return socket name
```

```
*     alen: length of user supplied address buffer
*
* Return Values:
*     0 if getsockname operation succeeded
*    -1 if getsockname operation failed.  In this case, errno is set to
*             indicate the error.
*       The asa parameter is filled in with the name of the socket
*       The alen parameter is filled in with the actual length of asa
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*       rpc_sockaddr_t rpcname;
*       int rpcnamelen;
*       int st;
*
* Algorithm:
*       call getsock(s) to get pointer file structure from file descriptor
*       if getsock() failed
*              provide error from getsock() to caller
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       copy in rpcnamelen from user alen parameter
*       if copy in fails
*              provide EFAULT error to caller
*       call rso_getsockname RPC with the following parameters
*              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*              remote file structure handle (so->rso_handler.rsh_reopendata)
*              f_flags field from file structure (fp->f_flags)
*              pointer to rpcname structure
*              pointer to rpcnamelen (&rpcnamelen)
*              pointer to error return value (&st)
*       if an error occurred (st != 0)
*              provide error code to caller
*              return
*       copy out rpcname structure to asa
*       if copy out fails
*              provide EFAULT error to caller
*       copy out rpcnamelen structure to alen
*       return
*/
```

The following specifies the server routine for the rso_getsockname RPC.

```
/*
```

```
* rso_getsockname_server(h, fh, f_flags, name, namelen, st)
* handle_t h;
* rso_rfh_t fh;
* int f_flags;
* rpc_sockaddr_t *name;
* int *namelen;
* int *st;
*
* Abstract:
*     server for rso_getsockname RPC
*
* Parameters
*     h: RPC handle for socket
*     fh: handle for remote file structure (local on server node)
*     f_flags:  f_flags field from client structure
*     name: buffer for returning socket name
*     namelen: pointer to length of name parameter from user
*     st: pointer to variable in which to store error return value
*
* Return Values:
*     socket name is copied into name buffer
*     length of socket name is copied into namelen
*     return indication is made via st variable
*
* Local Variables
*       int fd;
*       struct socket *so;
*       struct mbuf *m;
*
* Algorithm:
*       fd = file descriptor for file structure referred to in fh.
*              This may require some setup.
*       put fd where getsockname expects its fdes argument
*       put name where getsockname expects its asa argument
*       put namelen where getsockname expects its alen argument
*       u.u_error = 0
*
*       call getsockname()
*
*       *st = u.u_error
*       return
*
*/
```

2.1.3.3.2.7 Getsockopt

The getsockopt system call has no unusual states that require specialized handling on the client node. The following specifies the rgetsockopt call:

```
/*
 * rgetsockopt(s, level, name, val, avalsize)
 * int s;
 * int level;
 * int name
 * caddr_t val;
 * int *avalsize;
 *
 * Abstract:
 *      remote version of getsockopt system call
 *
 * Parameters
 *      NOTE: parameters are passed in an architecture dependent way
 *      s: file descriptor of socket to get socket options
 *      level: level at which the option resides
 *      name: name of option to be set
 *      val: address of optional user supplied address of buffer to return option
 *      avalsize: address of length of user supplied address buffer
 *
 * Return Values:
 *      0 if getsockopt operation succeeded
 *     -1 if getsockopt operation failed.  In this case, errno is set to
 *              indicate the error.
 *        The val parameter is filled in with result of the getsockopt() call
 *        The avalsize parameter is filled in with the amount of data in val
 *
 * Local Variables
 *        struct file *fp;
 *        struct socket *so;
 *        rpc_sockaddr_t rpcname;
 *        int rpcvallen;
 *        char tmpval[MLEN];
 *        int st;
 *
 * Algorithm:
 *        call getsock(s) to get pointer file structure from file descriptor
 *        if getsock() failed
 *              provide error from getsock() to caller
 *        fp = return value from getsock()
 *        so = pointer socket structure from file structure (fp->f_data)
```

```
*       copy in rpcvallen from user avalsize parameter
*       if copy in fails
*              provide EFAULT error to caller
*       call rso_getsockopt RPC with the following parameters
*              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*              remote file structure handle (so->rso_handler.rsh_reopendata)
*              f_flags field from file structure (fp->f_flags)
*              level parameter from user
*              name parameter from user
*              rpcvallen value
*              pointer to rpcvallen
*              pointer to tmpval buf
*              pointer to error return value (&st)
*       if an error occurred (st != 0)
*              provide error code to caller
*              return
*       if val != NULL and rpcvallen != 0
*              copy out rpcvallen bytes of tmpval to val
*              if copy out fails
*                     provide EFAULT error to caller
*                     return
*              copy out rpcvallen to avalsize
*       return
*/
```

The following specifies the server routine for the rso_getsockopt RPC.

```
/*
 * rso_getsockopt_server(h, fh, f_flags, level, opt_name, opt_size,
 *                                              opt_len, opt_buf, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * int level;
 * int opt_name;
 * int opt_size;
 * int *opt_len;
 * char opt_buf[];
 * int *st;
 *
 * Abstract:
 *     server for rso_getsockopt RPC
 *
 * Parameters
 *     h: RPC handle for socket
```

```
*      fh: handle for remote file structure (local on server node)
*      f_flags:  f_flags field from client structure
*      opt_name: name of option
*      opt_size: size of data from caller
*      opt_len: pointer to size of data returned to caller
*      opt_buf: buffer for returning socket name
*      st: pointer to variable in which to store error return value
*
* Return Values:
*      socket name is copied into name buffer
*      length of socket name is copied into namelen
*      return indication is made via st variable
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*       struct mbuf *m;
*       int valsize;
*
* Algorithm:
*       fd = file descriptor for file structure referred to in fh.
*              This may require some setup.
*       put fd where getsockopt expects its s argument
*       put level where getsockopt expects its level argument
*       put opt_name where getsockopt expects its name argument
*       put opt_buf where getsockopt expects its val argument
*       *opt_len = opt_size
*       put opt_len where getsockopt expects its avalsize argument
*       u.u_error = 0
*
*       call getsockopt()
*
*       *st = u.u_error
*       return
*
*/
```

2.1.3.3.2.8 Setsockopt

The setsockopt system call has no unusual states that require specialized handling on the client node. The following specifies the rsetsockopt call:

```
/*
 * rsetsockopt(s, level, name, val, avalsize)
 * int s;
 * int level;
```

```
* int name
* caddr_t val;
* int valsize;
*
* Abstract:
*     remote version of setsockopt system call
*
* Parameters
*     NOTE: parameters are passed in an architecture dependent way
*     s: file descriptor of socket to get socket options
*     level: level at which the option resides
*     name: name of option to be set
*     val: address of optional user supplied address of buffer to return option
*     valsize: length of user supplied address buffer
*
* Return Values:
*     0 if setsockopt operation succeeded
*    -1 if setsockopt operation failed.  In this case, errno is set to
*             indicate the error.
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*       rpc_sockaddr_t rpcname;
*       int rpcvallen;
*       char tmpval[MLEN];
*       int st;
*
* Algorithm:
*       call getsock(s) to get pointer file structure from file descriptor
*       if getsock() failed
*              provide error from getsock() to caller
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       if vallen > MLEN
*              provide EINVAL error to caller
*              return
*       if val != NULL
*              copy in vallen bytes from val to tmpval
*              if copy in fails
*                     provide EFAULT error to caller
*                     return
*       call rso_setsockopt RPC with the following parameters
```

```
*               RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*               remote file structure handle (so->rso_handler.rsh_reopendata)
*               f_flags field from file structure (fp->f_flags)
*               level parameter from user
*               name parameter from user
*               vallen value
*               pointer to rpcvallen
*               pointer to tmpval buf
*               pointer to error return value (&st)
*       if an error occurred (st != 0)
*               provide error code to caller
*               return
*       return
*/
```

The following specifies the server routine for the rso_setsockopt RPC.

```
/*
 * rso_setsockopt_server(h, fh, f_flags, level, opt_name, opt_size,
 *                                              opt_len, opt_buf, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * int level;
 * int opt_name;
 * int opt_size;
 * int *opt_len;
 * char opt_buf[];
 * int *st;
 *
 * Abstract:
 *      server for rso_setsockopt RPC
 *
 * Parameters
 *      h: RPC handle for socket
 *      fh: handle for remote file structure (local on server node)
 *      f_flags:  f_flags field from client structure
 *      level: buffer for returning socket name
 *      opt_name: name of option
 *      opt_size: size of data from caller
 *      opt_len: pointer to size of data returned to caller
 *      opt_buf: buffer for returning socket name
 *      st: pointer to variable in which to store error return value
 *
 * Return Values:
```

```
*      socket name is copied into name buffer
*      length of socket name is copied into namelen
*      return indication is made via st variable
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*       struct mbuf *m;
*       int valsize;
*
* Algorithm:
*       fd = file descriptor for file structure referred to in fh.
*              This may require some setup.
*       put fd where setsockopt expects its s argument
*       put level where setsockopt expects its level argument
*       put opt_name where setsockopt expects its name argument
*       put opt_buf where setsockopt expects its val argument
*       put opt_size where setsockopt expects its valsize argument
*       u.u_error = 0
*
*       call setsockopt()
*
*       *st = u.u_error
*       return
*
*/
```

2.1.3.3.2.9 Listen

The listen system call has no unusual states that require specialized handling on the client node. The following specifies the rlisten call:

```
/*
 * rlisten(s, backlog)
 * int s;
 * int backlog;
 *
 * Abstract:
 *      remote version of listen system call
 *
 * Parameters
 *      NOTE: parameters are passed in an architecture dependent way
 *      s: file descriptor of socket to listen on
 *      backlog: maximum length for queue of pending connections
 *
 * Return Values:
```

```
*     0 if listen operation succeeded
*    -1 if listen operation failed.  In this case, errno is set to
*             indicate the error.
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*
* Algorithm:
*       call getsock(s) to get pointer file structure from file descriptor
*       if getsock() failed
*             provide error from getsock() to caller
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       call rso_listen RPC with the following parameters
*             RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*             remote file structure handle (so->rso_handler.rsh_reopendata)
*             f_flags field from file structure (fp->f_flags)
*             backlog parameter from user
*             pointer to error return value (&st)
*       if an error occurred (st != 0)
*             provide error code to caller
*             return
*       return
*/
```

The following specifies the server routine for the rso_listen RPC.

```
/*
 * rso_listen_server(h, fh, f_flags, backlog, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * int backlog;
 * int *st;
 *
 * Abstract:
 *     server for rso_listen RPC
 *
 * Parameters
 *     h: RPC handle for socket
 *     fh: handle for remote file structure (local on server node)
 *     f_flags:  f_flags field from client structure
 *     backlog: maximum length of queue of pending connections
 *     st: pointer to variable in which to store the error return value
```

```
*
* Return Values:
*     return indication is made via st variable
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*
* Algorithm:
*       fd = file descriptor for file structure referred to in fh.
*              This may require some setup.
*       put fd where listen expects its s argument
*       put backlog where listen expects its backlog argument
*       u.u_error = 0;
*
*       call listen()
*
*       *st = u.u_error
*       return
*
*/
```

2.1.3.3.2.10 Sending Data

The system calls send(), sendmsg(), and sendto() are the socket specific calls provided to send data. They each go through a common routine called sendit(), which in turn calls sosend(). The routine sosend() is also used by the fileop routines for write() and writev() on socket type files.

The strategy for **FUSION** is to replace the send(), sendmsg(), and sendto() entry points with replacement system calls rsend(), rsendmsg(), and rsendto(). These routines will be identical to the standard routines, except they will call rsendit() instead of sendit(). The routine rsendit() will be identical to sendit(), except that it calls rsosend(). A common RPC rso_send() will be used for all cases of sending data, and it will be called from the routine rsosend().

The following specifies the rsend() system call:

```
/*
 * rsend(s, msg, len, flags)
 * int s;
 * char *msg;
 * int len;
 * int flags;
 *
 * Abstract:
 *     remote version of send system call
```

```
 *
 * Parameters
 *     s: file descriptor of socket to send data to.
 *     msg: message to send
 *     len: length of message
 *     flags:  flags associated with message
 *
 * Return Values:
 *     0 if sendto operation succeeded
 *    -1 if sendto operation failed.  In this case, errno is set to
 *              indicate the error.
 *
 * Algorithm:
 *       Identical to base send() routine, except call to sendit() is
 *              replaced with rsendit().
 */
```

The following specifies the rsendmsg() system call:

```
/*
 * rsendmsg(s, msg, flags)
 * int s;
 * struct msghdr msg[];
 * int flags;
 *
 * Abstract:
 *     remote version of sendmsg system call
 *
 * Parameters
 *     s: file descriptor of socket to send data to.
 *     msg: message to send
 *     flags:  flags associated with message
 *
 * Return Values:
 *    -1 if sendmsg operation failed.  In this case, errno is set to
 *              indicate the error.
 *     otherwise, the number of bytes sent is returned.
 *
 * Algorithm:
 *       Identical to base sendmsg() routine, except call to sendit() is
 *              replaced with rsendit().
 */
```

The following specifies the rsendto() system call:

```
/*
```

```
* rsendto(s, msg, len, flags, to, tolen)
* int s;
* char *msg;
* int len;
* int flags;
* struct sockaddr *to;
* int tolen;
*
* Abstract:
*     remote version of sendmsg system call
*
* Parameters
*     s: file descriptor of socket to send data to.
*     msg: message to send
*     len: length of the message
*     flags:  flags associated with message
*     to: domain specific address to send message
*     tolen: length of address to send message
*
* Return Values:
*    -1 if send operation failed.  In this case, errno is set to
*              indicate the error.
*     otherwise, the number of bytes sent is returned.
*
* Algorithm:
*       Identical to base sendto() routine, except call to sendit() is
*              replaced with rsendit().
*/
```

The following specifies the rsendit() routine:

```
/*
 * rsendit(s, mp, flags)
 * int s;
 * struct msghdr *mp;
 * int flags;
 *
 * Abstract:
 *     remote version of sendit subroutine
 *
 * Parameters
 *     s: file descriptor of socket to send data to.
 *     mp: pointer to message header of message to send
 *     flags:  flags associated with message
 *
```

```
 * Return Values:
 *     0 if sendit operation succeeded
 *    -1 if sendit operation failed.  In this case, errno is set to
 *              indicate the error.
 *
 * Algorithm:
 *       Identical to base sendit() routine, except call to sosend() is
 *              replaced with rsosend().
 */
```

The following specifies the rsosend() routine:

```
/*
 * rsosend(so, nam, uio, flags, rights)
 * struct socket *so;
 * struct mbuf *nam;
 * struct uio *uio;
 * int flags;
 * struct mbuf *rights;
 *
 * Abstract:
 *     remote version of sosend subroutine
 *
 * Parameters
 *     so: pointer to socket structure
 *     nam: name of remote entity
 *     uio: pointer to uio structure containing data to send
 *     flags:  flags associated with message
 *     rights: access rights associated with this socket
 *
 * Return Values:
 *     0 if sosend operation succeeded
 *     nonzero Error code if sosend operation failed.
 *
 * Local Variables
 *       char buffer[MAXBUFLEN];
 *       int buflen;
 *       int st;
 *
 * Algorithm:
 *       buflen = uio->uio_resid
 *       call uiomove(buffer, uio->uio_resid, UIO_WRITE, uio)
 *       call rso_send RPC with the following arguments
 *              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
 *              remote file structure handle (so->rso_handler.rsh_reopendata)
```

```
*               f_flags field from file structure (fp->f_flags)
*               length of data to send (buflen)
*               buffer of data to send (buffer)
*               message flags (flags)
*               to address (nam)
*               length of access rights (rights->m_len)
*               buffer of access rights (mtod(rights, char *)
*               pointer to st (&st)
*       return *st
*/
```

The following specifies the server routine for the rso_send RPC.

```
/*
 * rso_send_server(h, fh, f_flags, buf_len, buf, flags, to,
 *                                      accrights_len, accrights, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * int buf_len;
 * char buf[];
 * int flags;
 * rpc_sockaddr_t to;
 * int accrights_len;
 * char accrights[];
 * int *st;
 *
 * Abstract:
 *      server for rso_send RPC
 *
 * Parameters
 *      h: RPC handle for socket
 *      fh: handle for remote file structure (local on server node)
 *      f_flags:  f_flags field from client structure
 *      buf_len:  amount of data to send
 *      buf: buffer of data to send
 *      flags: flags associated with message
 *      to: address to send data
 *      accrights_len: length of access rights data
 *      accrights: access rights associated with send operation
 *      st: pointer to variable in which to store error return value
 *
 * Return Values:
 *      return indication is made via st variable
 *
```

```
* Local Variables
*       struct file *fp;
*       struct socket *so;
*       struct uio *uio;
*       struct mbuf *m;
*
* Algorithm:
*       fp = file pointer from fh
*       so = fp->fp_data
*       get a uio structure
*       uio = uio structure
*       set up structure from buf_len and buf
*       get an mbuf (or chain) to hold accrights_len of data
*       m = mbuf pointer
*       copy accrights into mbuf
*       call sosend(so, to, uio, flags, m)
*       *st = return value of sosend()
*       return
*/
```

2.1.3.3.2.11 Receiving Data

The system calls recv(), recvmsg(), and recvfrom() are the socket specific calls provided to receive data. They each go through a common routine called recvit(), which in turn calls soreceive(). The routine soreceive() is also used by the fileop routines for write() and writev() on socket type files.

The strategy for **FUSION** is to replace the recv(), recvmsg(), and recvto() entry points with replacement system calls rrecv(), rrecvmsg(), and rrecvfrom(). These routines will be identical to the standard routines, except they will call rrecvit() instead of recvit(). The routine rrecvit() will be identical to recvit(), except that it calls rsoreceive(). A common RPC rso_receive() will be used for all cases of receiving data, and it will be called from the routine rsoreceive().

The following specifies the rrecv() system call:

```
/*
 * rrecv(s, msg, len, flags)
 * int s;
 * char *msg;
 * int len;
 * int flags;
 *
 * Abstract:
 *     remote version of recv system call
 *
 * Parameters
```

```
 *     s: file descriptor of socket to receive data from.
 *     msg: buffer to hold received message
 *     len: length of buffer
 *     flags:  flags associated with message
 *
 * Return Values:
 *     0 if recv operation succeeded
 *    -1 if recv operation failed.  In this case, errno is set to
 *              indicate the error.
 *
 * Algorithm:
 *       Identical to base recv() routine, except call to recvit() is
 *              replaced with rrecvit().
 */
```

The following specifies the rrecvmsg() system call:

```
/*
 * rrecvmsg(s, msg, flags)
 * int s;
 * struct msghdr msg[];
 * int flags;
 *
 * Abstract:
 *     remote version of recvmsg system call
 *
 * Parameters
 *     s: file descriptor of socket to receive data from.
 *     msg: msghdr structure to hold received message
 *     flags:  flags associated with message
 *
 * Return Values:
 *    -1 if recvmsg operation failed.  In this case, errno is set to
 *              indicate the error.
 *     otherwise, the number of bytes received is returned.
 *
 * Algorithm:
 *       Identical to base recvmsg() routine, except call to recvit() is
 *              replaced with rrecvit().
 */
```

The following specifies the rrecvfrom() system call:

```
/*
 * rrecvfrom(s, msg, len, flags, from, fromlenaddr)
 * int s;
```

```
* char *msg;
* int len;
* int flags;
* struct sockaddr *to;
* int *fromlenaddr;
*
* Abstract:
*     remote version of recvfrom system call
*
* Parameters
*     s: file descriptor of socket to receive data from.
*     msg: buffer to hold received message
*     len: length of buffer
*     flags:  flags associated with message
*     from: domain specific address to receive message from
*     fromlenaddr: pointer to length of from address
*
* Return Values:
*    -1 if recvfrom operation failed.  In this case, errno is set to
*             indicate the error.
*     otherwise, the number of bytes received is returned.
*
* Algorithm:
*       Identical to base recvfrom() routine, except call to recvit() is
*             replaced with rrecvit().
*/
```

The following specifies the rrecvit() routine:

```
/*
 * rrecvit(s, mp, flags, namelenp, rightslenp)
 * int s;
 * struct msghdr *mp;
 * int flags;
 * caddr_t namelenp;
 * caddr_t rightslenp;
 *
 * Abstract:
 *     remote version of recvit subroutine
 *
 * Parameters
 *     s: file descriptor of socket to receive data from.
 *     mp: pointer to message header to hold message received
 *     flags:  flags associated with message
 *
```

```
 * Return Values:
 *     0 if recvit operation succeeded
 *    -1 if recvit operation failed.  In this case, errno is set to
 *              indicate the error.
 *
 * Algorithm:
 *       Identical to base recvit() routine, except call to soreceive() is
 *              replaced with rsoreceive().
 */
```

The following specifies the rsoreceive() routine:

```
/*
 * rsoreceive(so, aname, uio, flags, rightsp)
 * struct socket *so;
 * struct mbuf **aname;
 * struct uio *uio;
 * int flags;
 * struct mbuf **rightsp;
 *
 * Abstract:
 *     remote version of soreceive subroutine
 *
 * Parameters
 *     so: pointer to socket structure
 *     aname: pointer to name of remote entity
 *     uio: pointer to uio structure containing data to send
 *     flags:  flags associated with message
 *     rightsp: pointer to access rights associated with this socket
 *
 * Return Values:
 *     0 if soreceive operation succeeded
 *     nonzero Error code if soreceive operation failed.
 *
 * Local Variables
 *       char buffer[MAXBUFLEN];
 *       int buflen;
 *       int st;
 *
 * Algorithm:
 *       call rso_receive RPC with the following arguments
 *              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
 *              remote file structure handle (so->rso_handler.rsh_reopendata)
 *              f_flags field from file structure (fp->f_flags)
 *              length of data to send (uio->uio_resid)
```

```
*               pointer to returned length of data received (&buflen)
*               buffer of data to send (buffer)
*               message flags (flags)
*               pointer to fromaddress address (aname)
*               length of access rights (*rights->m_len)
*               pointer length of access rights (&(*rights->m_len))
*               pointer to buffer of access rights (mtod(*rights, char *))
*               pointer to st (&st)
*       call uiomove(buffer, uio->uio_resid, UIO_READ, uio)
*       return *st
*/
```

The following specifies the server routine for the rso_receive RPC.

```
/*
 * rso_receive_server(h, fh, f_flags, buf_size, buf_len, buf, flags,
 *                    from, msg_size, msg_len, accrights, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * int buf_size;
 * int *buf_len;
 * char buf[];
 * int flags;
 * rpc_sockaddr_t from;
 * int msg_size;
 * int *msg_len;
 * char accrights[];
 * int *st;
 *
 * Abstract:
 *     server for rso_receive RPC
 *
 * Parameters
 *     h: RPC handle for socket
 *     fh: handle for remote file structure (local on server node)
 *     f_flags:  f_flags field from client structure
 *     buf_size:  size of user's receive buffer
 *     buf_len:  pointer to caller's size variable
 *     buf: buffer to hold data received
 *     flags: flags associated with receive call
 *     from: address to receive data from
 *     msg_size: length of access rights data
 *     msg_len: pointer to caller's access rights size
 *     accrights: received access rights
```

```
*     st: pointer to variable in which to store error return value
*
* Return Values:
*     return indication is made via st variable
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*       struct uio *uio;
*       struct mbuf *m;
*
* Algorithm:
*       fp = file pointer from fh
*       so = fp->fp_data
*       get a uio structure
*       uio = uio structure
*       call soreceive(so, to, uio, flags, &m)
*       *buf_len = uio->uio_resid
*       call uiomove(buffer, uio->uio_resid, UIO_READ, uio)
*       *msg_len = m->m_size
*       copy m->m_size bytes from mtod(m, caddr_t) to accrights
*       *st = return value of soreceive()
*       return
*/
```

2.1.3.3.2.12 Sending and Receiving Access Rights

The sendmsg() and recvmsg() socket calls allow processes connected through a local Unix domain socket to exchange file descriptors. This feature is not implemented by all existing socket implementations due to security considerations. Consequently, the **FUSION** remote socket implementation will not support this feature. The RPCs provided are capable of supporting should a need to provide such support be needed at a later date.

2.1.3.3.2.13 Shutdown

The shutdown system call has no unusual states that require specialized handling on the client node. The following specifies the rshutdown call:

```
/*
 * rshutdown(s, how)
 * int s;
 * int how;
 *
 * Abstract:
 *     remote version of shutdown system call
 *
```

```
* Parameters
*     NOTE: parameters are passed in an architecture dependent way
*     s: file descriptor of socket to get socket options
*     how: how the socket should be shut down
*
* Return Values:
*     0 if shutdown operation succeeded
*    -1 if shutdown operation failed.  In this case, errno is set to
*              indicate the error.
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*
* Algorithm:
*       call getsock(s) to get pointer file structure from file descriptor
*       if getsock() failed
*              provide error from getsock() to caller
*       fp = return value from getsock()
*       so = pointer socket structure from file structure (fp->f_data)
*       call rso_shutdown RPC with the following parameters
*              RPC handle from socket structure (so->rso_handle.rsh_rpchandle)
*              remote file structure handle (so->rso_handler.rsh_reopendata)
*              f_flags field from file structure (fp->f_flags)
*              how parameter from user
*              pointer to error return value (&st)
*       if an error occurred (st != 0)
*               provide error code to caller
*               return
*       return
*/
```

The following specifies the server routine for the rso_shutdown RPC.

```
/*
 * rso_shutdown_server(h, fh, f_flags, how, st)
 * handle_t h;
 * rso_rfh_t fh;
 * int f_flags;
 * int how;
 * int *st;
 *
 * Abstract:
 *     server for rso_shutdown_RPC
 *
```

```
* Parameters
*     h: RPC handle for socket
*     fh: handle for remote file structure (local on server node)
*     f_flags:  f_flags field from client structure
*     how: how the socket should be shutdown
*     st: pointer to variable in which to store error return value
*
* Return Values:
*     return indication is made via st variable
*
* Local Variables
*       struct file *fp;
*       struct socket *so;
*
* Algorithm:
*       fd = file descriptor for file structure referred to in fh.
*              This may require some setup.
*       put fd where shutdown expects its s argument
*       put how where shutdown expects its how argument
*       u.u_error = 0
*
*       call shutdown()
*
*       *st = u.u_error
*       return
*
*/
```

2.1.3.4 NIDL Prototype for Remote Socket RPCs

The following provides a sample of a NIDL definition for the remote socket RPCs described above. This is provided only as a sample, the further substantiate the design described previously. It has not been compiled and tested.

```
[ uuid(<UUID for remote socket RPCs>)
 , version(1.0), port("ip:[<PORT for remote socket RPCs]") ]

interface rsocket {

        const long      RSO_MAXPATHLEN          = 1024;

        /*
         * domains supported
         * from /usr/include/sys/socket.h
         */
```

```
const short     RSO_AF_UNIX             = 1;
const short     RSO_AF_INET             = 2;

/*
 * hardware types
 */
const short     RSO_ETHER               = 1;
const short     RSO_IEEE802_3           = 2;
const short     RSO_IEEE802_5           = 3;
const short     RSO_X_25                = 9;

/*
 * requester's id
 */
typedef long    handle_t;

/*
 * sockaddr structure
 */
typedef union switch(short sa_family) {
    case RSO_AF_INET:
        unsigned short  sin_port;
        unsigned short  sin_addr;
        break;
    case RSO_AF_UNIX:
        string0[RSO_MAXPATHLEN]sun_path;
        break;
}rpc_sockaddr_t;

/*
 * if dependent data type
 */
typedef union switch (short arp_type){
        case RSO_IEEE802_3:
        case RSO_ETHER:
                break;
        case RSO_IEEE802_5:
                unsigned short  arp_ref;
                unsigned short  arp_seq[8];
                break;
        case RSO_X_25:
                unsigned long   arp_channel;
                break;
```

```
} if_dependent;

typedef struct {
        rpc_sockaddr_t  arp_pa;         /* protocol address */
        rpc_sockaddr_t  arp_ha;         /* hardware address */
        int             arp_flags;      /* flags */
        unsigned short  arp_halength;   /* length address */
        if_dependent    arp_ifd;        /* hardware dependent info */
        unsigned long   arp_type;       /* interface type */
}rpc_arpreq;

/*
 * handle passed to socket server
 * to identify the client's file descriptor
 */
typedef struct rso_handle rso_rfh_t;

/*
 * reopen data returned by
 * rso_prep_export()
 */
typedef struct rso_reopendata {
        int rso_rfileid;
        int rso_uniqeid;
} rso_reopendata_t;

/*
 * RPC for remote accept() system call
 */
int rso_accept (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [out]           rso_reopendata_t        *reopendata,
        [out]           rpc_sockaddr_t          *addr,
        [in, out]       int                     *addrlen,
        [out]           unsigned32              *st
);

/*
 * do_reopen RPC called by accept() and bind() RPC server routines
 */
int rso_doreopen (
```

```
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [out]           unsigned32          *st
);

/*
 * RPC for remote bind() system call
 */
int rso_bind (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            rpc_sockaddr_t      name,
        [out]           unsigned32          *st
);

/*
 * RPC for remote connect() system call
 */
int rso_connect (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            rpc_sockaddr_t      name,
        [in]            int                 namelen,
        [out]           unsigned32          *st
);

/*
 * RPC for remote getpeername() system call
 */
int rso_getpeername (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [out]           rpc_sockaddr_t      *name,
        [in,out]        int                 *namelen,
        [out]           unsigned32          *st
);

/*
 * RPC for remote getsockname()
 */
```

```
int rso_getsockname (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [out]           rpc_sockaddr_t      *name,
        [in,out]        int                 *namelen,
        [out]           unsigned32          *st
);

/*
 * RPC for remote getsockopt() system call
 */
int rso_getsockopt (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 level,
        [in]            int                 opt_name,
        [in]            int                 opt_size,
        [out]           int                 *opt_len,
                        char                [max_is(opt_size),
                                            last_is(opt_len),
                                            in, out]
                                            opt_buf[],
        [out]           unsigned32          *st
);

/*
 * RPC for remote setsockopt() system call
 */
int rso_setsockopt (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [out]           int                 level,
        [in]            int                 opt_name,
        [in]            int                 opt_size,
        [out]           int                 *opt_len,
                        char                [max_is(opt_size),
                                            last_is(opt_len),
                                            in, out]
                                            opt_buf[],
        [out]           unsigned32          *st
```

```
);

/*
 * RPC for remote listen() system call
 */
int rso_listen (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     backlog,
        [out]           unsigned32              *st
);

/*
 * RPC for remote receive data calls:
 *      recv()
 *      recvmsg()
 *      recvfrom()
 *      read()
 *      readv()
 */
int rso_receive (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     buf_size,
        [out]           int                     *buf_len,
                        char                    [max_is(buf_size),
                                                last_is(buf_len),
                                                out] buf[],
        [in]            int                     flags,
        [in, out]       rpc_sockaddr_t          *from,
        [in]            int                     msg_size,
        [out]           int                     *msg_len,
                        struct msg              [max_is(msg_size),
                                                last_is(msg_len),
                                                out] accrights[],
        [out]           unsigned32              *st
);

/*
 * RPC for remote send data calls
 *      send()
```

```
 *      sendmsg()
 *      sendfrom()
 *      write()
 *      writev()
 */
int rso_send (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 buf_len,
                        char                [last_is(buf_len),
                                            in] buf[],
        [in]            int                 flags,
        [in]            rpc_sockaddr_t      to,
        [in]            int                 accrights_len,
                        char                [last_is(accrights_len),
                                            in] accrights[],
        [out]           unsigned32          *st
);

/*
 * RPCs for remote shutdown() system call
 */
int rso_shutdown (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 how,
        [out]           unsigned32          *st
);


/*
 * single integer input ioctl(s):
 *      FIONBIO
 *      FIOASYNC
 *      SIOCSHIWAT
 *      SIOCSLOWAT
 */
int rso_ioctl_inint (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
```

```
        [in]            int                     cmd,
        [in]            int                     data,
        [out]           unsigned32              *st
);

/*
 * single long input argument ioctl(s):
 *      SIOCSPGRP
 */
int rso_ioctlinulong (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [in]            unsigned long           data,
        [out]           unsigned32              *st
);

/*
 * single output int argument ioctls:
 *      SIOCGHIWAT
 *      SIOCGLOWAT
 *      SIOCATMARK
 */
int rso_ioctl_outint (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [out]           int                     *data,
        [out]           unsigned32              *st
);

/*
 * single output unsigned long argument ioctl(s):
 *      FIONREAD
 */
int rso_ioctloutulong (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [out]           unsigned long           *data,
```

```
        [out]           unsigned32              *st
);

/*
 * adding and deleting ARP information ioctl(s):
 *      SIOCSARO
 *      SIOCSARP_802_5
 *      SIOCSARP_x_25,
 *      SIOCDARP
 *      SIOCDARP_802_5
 *      SIOCDARP_X_25
 */
int rso_ioctlinARP (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [in]            rpc_arpreq              arp_req,
        [out]           unsigned32              *st
);

/*
 * retrieving ARP information ioctl(s):
 *      SIOCGARP
 *      SIOCGARP_802_5
 *      SIOCGARP_X_25
 */
int rso_ioctloutARP (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [in, out]       rpc_arpreq              arp_req,
        [out]           unsigned32              *st
);

/*
 * ifconf ioctl(s):
 *      SIOCGIFCONF
 */
int rso_ioctlIFCONF (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
```

```
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [in]            int                     max_index,
        [out]           int                     *out_index,
        struct {
                string0[14]     ifr_name,
                sockaddr_t      ifr_addr,
        } [ last_is (out_index), max_is(max_index), out ] ifr_req[],
        [out]           unsigned32              *st
);

/*
 * ioctl(s):
 *      SIOCGIFFLAGS
 *      SIOCGIFMTU
 *      SOICGIFREMMTU
 */
int rso_ioctloutIFshort (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [in]            string0[14]             ifr_name,
        [out]           short                   *ifr_short,
        [out]           unsigned32              *st
);

/*
 * ioctl(s):
 *      SIOCGIFMETRIC
 */
int rso_ioctloutIFint (
        [in]            handle_t                h,
        [in]            rso_rfh_t               fh,
        [in]            int                     f_flags,
        [in]            int                     cmd,
        [in]            string0[14]             ifr_name,
        [out]           int                     *ifr_metric,
        [out]           unsigned32              *st
);

/*
 * ioctl(s):
```

```
 *      SIOCSIFMETRIC
 */
int rso_ioctlinIFint (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 cmd,
        [in]            string0[14]         ifr_name,
        [in]            int                 ifr_metric,
        [out]           unsigned32          *st
);

/*
 * ioctl(s):
 *      SIOCSIFMTU
 *      SIOCSIFREMMTU
 *      SIOCSIFFLAGS
 */
int rso_ioctlinIFshort (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 cmd,
        [in]            string0[14]         fr_name,
        [in]            short               ifr_short,
        [out]           unsigned32          *st
);

/*
 * ioctl(s):
 *      SIOCGIFADDR
 *      SIOCGIFBRDADDR
 *      SIOCGIFDSTADDR
 *      SIOCGIFNETMASK
 */
int rso_ioctloutIFaddr (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 cmd,
        [in]            string0[14]         ifr_name,
        [out]           sockaddr_t          *ifr_addr,
        [out]           unsigned32          *st
```

```
);

/*
 * ioctl(s):
 *      SIOCSIFADDR
 *      SIOCSIFNETMASK
 *      SIOCSIFBRDADDR
 *      SIOCSIFDSTADDR
 */
int rso_ioctlinIFaddr (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 cmd,
        [in]            string0[14]         ifr_name,
        [in]            sockaddr_t          ifr_addr,
        [out]           unsigned32          *st
);

/*
 * ioctl(s):
 *      SIOCADDRT
 *      SIOCDELRT
 */
int rso_ioctlSIFRT (
        [in]            handle_t            h,
        [in]            rso_rfh_t           fh,
        [in]            int                 f_flags,
        [in]            int                 cmd,
        [in]            sockaddr_t          rt_dst,
        [in]            sockaddr_t          rt_gateway,
        [in]            short               rt_flags,
        [out]           unsigned32          *st
);

/*
 * socketx() call
 */
 int rso_socketx (
        [in]            handle_t            h,
        [in,out]        rso_rhf_t           fh,
        [out]           int                 *st
);
```

```
};
```

2.1.4 Remote Select

2.1.4.1 Introduction

This document describes a design for supporting the select and poll system calls, within **FUSION**, where special files may be controlled on remote hosts. The select system call was introduced in one of the BSD releases, to provide a mechanism for a single Unix process to monitor more than one I/O device at the same time. It allows a process to await input on any number of character devices, for example.

The poll system call, which came from System V, provides essentially the same function as select. The syntax differs slightly but the semantics are almost the same.

Note also that the select call in AIX 3.1 differs syntactically from the select call in BSD. In this document we will refer to all of these system calls as "select".

The essential syntax of the select function is described in NIDL as follows :

```
select_return_val_t
select
(
    [in]       int  endpoint_count,
    [in, out] endpoint_query_t array[size_is(endpoint_count)];
    [in]       long timeout
);

typedef enum {timeout, error, okay} select_return_val_t;

typedef struct
{
    endpoint_t      endpoint;
    event_set_t     awaited_events;
    event_set_t     events_which_occurred;
} endpoint_query_t;
```

The essential semantics are as follows: a set of I/O endpoints is provided. For each endpoint a set of awaited events is provided. An overall timeout is provided. The calling process is to be suspended if necessary, until at least one of the awaited events has occurred, or the timeout has expired. An endpoint can be anything named by a file descriptor (file, pipe, device, socket). The events that can be awaited include "input available", "ready for output" and "exception occurred". Some Unix versions extend this set somewhat (e.g. System V.4). "Input available" actually means "a single character read() will not block". "Ready for output" actually means that "a single character write() will not block". "Exception occurred" is not precisely defined. For the case when the endpoint is a device driver for an RS-232 serial line, "exception occurred" is used to indicate loss of carrier.

The timeout can be specified as any value between zero and infinity, inclusive. When the select call returns, the caller is returned enough information to determine what happened, i.e. which awaited events on which endpoints have occurred, or whether the timeout expired.

2.1.4.2 Requirements for FUSION

The primary requirement for FUSION is to support the select system call when the endpoint is remote, i.e. located on a host different to that on which the calling process executes.

Another requirement is that of heterogeneity. Non-identical kernels should be able to inter-operate, using the remote select functionality. Since different kernels implement select in rather different ways, this requirement means that the RPC interfaces which support remote select should be independent of specific kernel implementation details.

2.1.4.3 Rationale for design

2.1.4.3.1 Select implementations vary widely

The mechanisms described below were designed to be as independent as possible from kernel-specific details of select implementation. It is especially important that the RPC interfaces be independent. Some of the internal kernel interfaces may, however, need to change to fit in with various kernels.

2.1.4.3.2 A call back mechanism is needed

The semantics of select requires a mechanism for asynchronous notification. The device driver needs to notify its client when the events specified in a select call have actually occurred. This implies the need for a call back mechanism. Each client host which can issue a remote select, must have a server (kernel) process running, to service these remote select notifications.

2.1.4.3.3 Call backs can't be made in interrupt context

Select notifications are normally performed from the interrupt context part of device drivers, because the occurrence of the selected events, for example "input available", is usually discovered at interrupt level.

However, because remote select notifications require an RPC to the selecting process's host, these notifications cannot be made directly from interrupt context. This is because an RPC typically requires the caller to block, awaiting acknowledgement, and it is not permissible for interrupt code to block.

Consequently, the call backs required for remote select notifications will be performed by a dedicated kernel process. The interrupt code which notices that select notification must be done, will record in global kernel memory which call back needs to be done, and will just make the dedicated process runnable.

2.1.4.3.4 Optimization: avoid unnecessary rescan of endpoints

Implementation fo select vary in the amount of information that accompanies select notification. In the BSD, for example, no information is passed, and the notified

process must re-poll all the endpoints it is selecting, to discover which events have occurred, and at which endpoints. In AIX 3.1, on the other hand, there is a pre-process list of pending endpoint selects. The notification code records, in this list, information about the reason for the notification.

Since re-polling remote endpoints is even more expensive, the remote select notification RPC will pass enough information to avoid the need for re-polling over the network.

2.1.4.3.5 Optimization: avoid expired notifications

If a process calls select specifying a finite timeout, and the timeout expires before any of the selected endpoint events occur, it is best to avoid subsequent unnecessary remote select notifications, since the selecting process has already been woken by the timeout. For this reason, the RPC for the asynchronous remote poll operations passes the timeout value. If the timeout expires before an event occurs, the data structures are cleaned up and no call back is made.

2.1.4.4 Design details

2.1.4.4.1 pending incoming notification list

Each host will maintain a list called the pending incoming notification list(PINL), which will contain an element for each possible pending remote select notification which might ARRIVE AT this host. There will be an entry for each combination of selecting process and remote selected endpoint. This list would be scanned prior to a remote select operation, any select on the same object would be removed from this list and also automatically removed on the remote host when the select is sent there. An entry will be added to this list just prior to each call to the remote_selpoll() RPC. A dedicated kernel process will check this list to remove the entries which has X minutes old.

Some fields in the entries will be kernel-specific, but there will be enough information to allow:

- validation of the notification call back
- local notification of the appropriate selecting process
- avoiding wakeups of processes whose select's have already timed-out

2.1.4.4.2 pending outgoing notification list

Each host will maintain a list called the pending outgoing notification list(PONL), which will contain an element for each possible pending remote select notification which might be sent from this host. An entry will be placed in this list each time an remote_selpoll () RPC returns a negative response. The entry will be removed when the appropriate remote_selnotify() RPC is made.

Some fields in the entries will be kernel-specific, but there will be enough information to allow:

- validation of the notification call back
- location of the appropriate remote call back server, to which the remote_selnotify() RPC will be made
- avoiding call backs to notify processes whose select's have already timed-out

2.1.4.4.3 Select control block structure

AIX3.1 has a select control block structure in the process table. The control block is a link list which can be used to implement the pending incoming notification list and the pending outgoing notification list. One pointer "*rem_corl" is added to point to a structure which stores information for remote select operations.

```
struct  sel_cb
{
        struct sel_cb   *proc_chain;    /* next blk on proc chain       */
        struct sel_cb   *dev_chain_f;   /* next blk on hash chain       */
        struct sel_cb   *dev_chain_b;   /* prev. blk on hash chain      */
        ushort          reqevents;      /* requested events             */
        ushort          rtnevents;      /* returned events              */
        int             dev_id;         /* device id: devno, etc.       */
        int             unique_id;      /* unique id: chan, etc.        */
        struct proc     *procp;         /* ptr to proc table entry      */
        int             corl;           /* correlator: fp, etc.         */
        struct rem_corl *rem_corl;      /* ptr to a correlator struct   */
                                        /* for remote select            */
        void            (*notify)();    /* function ptr for nested poll */
};

struct rem_corl {
        int          rem_node;          /* remote selecting node        */
        int          rem_corl;          /* corl of the client node      */
        int          rem_flag;          /* flag(read/write,exception)   */
        int          rem_pid;           /* remote selecting process     */
        ulong        rem_timeout_msec;   /* timeout value               */
};
```

2.1.4.4.4 remote select file-op interface

1.

```
rem_select(fp, corl, reqevent, rtneventsp, notify)
struct file *fp;
int corl;
ushort reqevents;
ushort *rtneventsp;
```

```
void (*notify)();
{
        Scan pending incoming notification list

        If the selected file descriptor/file pointer already exists
        on the PINL list with the same correlator, then remove it from
        the list.

        Add a new entry on the pending select notification list

        remote_selpoll(h, corl, reqevent, curnode, flags, pid,
                                timeout, &rtnevent, &uerror, &status)
}

/*
 * This operation will be performed by a dedicated kernel process,
 * at host where the endpoint exists, to avoid unnecessary remote
 * select notification
 */
stale_selscan(PONL entry)
{
        scan the pending outgoing list of registered process
        if the PONL entry is valid then remove it
}

/*
 * This routine is called to notify the selecting process that
 * the selected event has occurred
 */
notify(sel_id, unique_id, rtevents, pid)
{
        find entry in the PONL[pid]
        setup call remote_selnotify(h, select_id, unique_id, rtevents)
        remove entry PONL[pid]
}
```

2.1.4.4.5 NIDL interface details

```
/*
 * remote_selpoll() -- poll a remote endpoint, enabling
 *                     subsequent asynchronous notification.
 */
[idempotent]
remote_selpoll(
```

```
        [in]  handle_t                  h,
        [in]  rendpoint_t               endpoint,
        [in]  rpoll_events_t            events,
        [in]  port_t                    callback_port,
        [in]  pid_t                     pid,
        [in]  ulong                     msec_timeout,
        [out] rpoll_events_t            *returned_events,
        [out] errno_t                   *copy_of_u_error,
        [out, comm_status] status_t)    *status
{

        Fake a file struct

        if an entry exists in the PONL remove it
        add new entry to the pending select outgoing notification list
        set up timeout for msec_timeout
        if (timeout)
             setup call kernel process to call stale_selscan(PONL[pid])
             to do the cleanup.

        /*
         * file_id - file ptr or file descriptor
         * corl    - correlator
         * reqevent- requested event
         * *rtneventsp - ptr to occurred event
         * flags
         * notify - it knows how to notify remote selecting
         *          process via remot_selnotify() RPC.
         */
        selpoll(file_id, corl, reqevent, rtneventsp, flags, notify)

}


/*
 * remote_selnotify -- notify a remote selecting process
 *                          of event occurrence.
 */
    [idempotent]
    void
    remote_selnotify
    (
        [in]  handle_t          h,
```

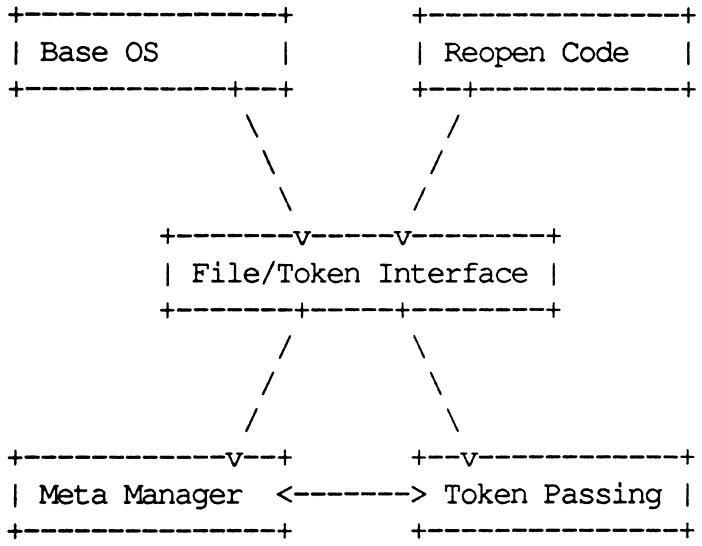
```
    [in]  int              sel_id,
    [in]  int              unique_id,
    [in]  ushort           events_occurred,
    [out, comm_status] status_t      *status
)
{
    selnotify(sel_id, unique_id, rtnevents)

}
```

2.1.5 File Offset Coherency

2.1.5.1 Overview

One of the interesting aspects of process transparency is that when directly related processes run on different nodes, multiple nodes need to share open-file structures as if they were on a single node. In essence, a single logical file struct can have physical manifestations at several different nodes. This means that the file offset and file flags must appear to be shared as though through a single shared struct file in memory. To accomplish this, **FUSION** controls the sharable fields in the file structure with a token mechanism known as "file block tokens" or "file offset tokens".



The basic idea behind file block tokens is that the file offset and file flags in an open file structure (aka "file block") are only known to be valid and modifiable at a given node if that open file's token is present and valid at the node. Implementing this scheme requires the following logical parts:

Base Hooks

The Base OS must consistently use the file block token macros when accessing the controlled members of the file structure. This requires a trivial change to each of three dozen or so routines. For efficiency, two fields are added to the file structure for fileblock token bookkeeping.

Reopen Hooks

The **FUSION** reopen code needs to initiate the fileblock token mechanism when related processes move between nodes.

FileBlock to Token

This module provides the only tie between the fileblock token and the file

it controls. It provides the fileblock interface hook macros to the Base OS, the reopen related functions to the reopen hooks, and the functions which the token passing code invokes to pass the file's fields to and from the token's caboose.

Meta Token Manager

This module provides the functions which create, delete, move, and recover fileblock token management. To do its job, it keeps track of the list of nodes using a given fileblock token. It is capable of recovering from lost tokens or from a lost token manager. It provides problem reporting functions to be called from the Token Passing module or from keep alive services to invoke its recovery functionality.

Token Passing Protocol

This module implements the medium frequency operations of requesting, recalling, and granting tokens. When tokens arrive, the token's operation branch table is used to pass the token controlled data to or from the file structure (allowing use of the token management function in other contexts). to install or revoke the file's data.

This design is broken down as described to allow replacement of individual modules to change policy or improve performance. However, any replacement modules must provide the interfaces specified here, and should meet the following goals, which this design strives for.

Base Hooks

- Should require minimal changes to base.
- Should be nearly free if fileblock tokens aren't used.

Reopen Hooks

- Must fit in cleanly with Reopen architecture.

FileBlock to Token Interface

- Explicit tokens only created when sharing occurs.
- Explicit tokens should disappear when there is only one node using a file block.

Meta Token Manager

- No support required from the file object's "storage node" (i.e., doesn't depend on AFS, NFS, remote sockets, etc.)

- Token manager should be one of the using nodes when possible.
- Recover from loss of token manager as cleanly as possible.
- Reasonably simple recovery protocol.
- Handle loss of token in a reasonable way.

Token Passing Protocol

- Good performance: no extra round trips in common cases.
- Fairness (sharing nodes guaranteed to get a chance).
- Simple protocol.
- Extensible to conflicting tokens.

2.1.5.2 Hooks

As mentioned in the introduction, Fileblock Tokens require hooks in base OS and in the **FUSION** file reopen code. This section describes these hooks.

2.1.5.2.1 Hooks in the Base

The hooks in the base OS consist of the addition of macros to lock and unlock the file block token whenever the file's offset or flags are consulted or modified. In AIX 3.1 the macro is added in about 3 dozen places. These macros are easily added to any UNIX like OS.

As an aside, the OSF1 kernel (implemented on top of the OSF1 micro-kernel) already locks and unlocks file structures in these (and a few other) places, because its kernel code is generally preemptable. The use of FP_LOCK/UNLOCK also helps OSF1 work correctly in an MP environment. An OSF1 implementation of **FUSION** would presumably take advantage of the existing locks by simply changing a subset of the FP_LOCK/UNLOCK calls to call an extended macro which also handle's File Block Tokens.

NEEDSWORK: more detail... cleanup org.

In falloc(), add code to clear out the new fields in the file structure (set state to no-token, and pointer to null).

In the path from any system call (or its equivalent) which uses or modifies fp->f_offset, file-block token must be obtained before using fp->f_offset or f_flag, and held until done.

```
inline void FBTOKEN_LOCK(struct file *fp);
inline void FBTOKEN_UNLOCK(struct file *fp);
```

After FBTOKEN_LOCK is called, and until FBTOKEN_UNLOCK is called, a file's f_offset and f_flag fields can be freely used with the normal UNIX kernel semantics. (These are actually macro's, to eliminate any significant performance impact of fileblock tokens when the token is explicitly or implicitly present).

The file block token hooks also require a minor addition to the base OS's file structure. The utility of these fields will be discussed in the Fileblock Token Interface section.

```
    /*
     * The part of a file block token's state always in struct file.
     */
    struct file_token_hook
    {
            short ft_tlckcnt;
            struct ftoken *ft_tcb;
    }

    struct file
    {
            ...
#ifdef FILE_OFFSET_TOKENS
            struct file_token_hook f_ftok;
#endif FILE_OFFSET_TOKENS
    }
```

The following version of AIX/3.1's version of rwuio() shows, via ifdef's, how the fileblock token macros are used. Rwuio is is the low level routine eventually called when doing a read or write (kreadv() and kwritev()) under AIX/3.1. (Some comments and blank lines have been stripped out for presentation in this document).

```
    static
    int
    rwuio (fp, iov, iovcnt, ext, seg, rw, countp, fflag)
    register struct file    *fp;            /* file pointer of file for I/O */
    register struct iovec   *iov;           /* vector of I/O buffers        */
    register int            iovcnt;         /* number of I/O buffers        */
    int                     ext;            /* device driver ext value      */
    int                     seg;            /* segment with I/O buffers     */
    register enum uio_rw    rw;             /* dir:  UIO_READ or UIO_WRITE  */
    register int            *countp;        /* return of I/O byte count     */
    int                     fflag;          /* extra file flags             */
    {
            struct uio      auio;   /* uio structure describing I/O         */
            int             i;      /* loop counter over buffers            */
            int             rc;     /* return code                          */
```

```
        int             count;  /* count of bytes read or written       */
        label_t         jbuf;   /* exception return point               */

        /* ...  */
        if ((fp->f_flag & (rw==UIO_READ ? (FREAD|FEXEC) : FWRITE)) == 0)
                return EBADF;

#ifdef TOKENS
        FBTOKEN_LOCK(fp);
#endif /* TOKENS */
        auio.uio_iov    = iov;
        auio.uio_iovcnt = iovcnt;
        auio.uio_segflg = seg;
        auio.uio_fmode  = fp->f_flag | fflag;
        auio.uio_offset = fp->f_offset;
        auio.uio_resid  = 0;

        for (i = 0; i < auio.uio_iovcnt; i++) {
                if (iov->iov_len < 0)
#ifndef TOKENS
                        return EINVAL;
#else /* TOKENS */
                {
                        rc = EINVAL;
                        goto eout;
                }
#endif /* TOKENS */
                auio.uio_resid += iov->iov_len;
                if (auio.uio_resid < 0)
#ifndef TOKENS
                        return EINVAL;
#else /* TOKENS */
                {
                        rc = EINVAL;
                        goto eout;
                }
#endif /* TOKENS */
                iov++;
        }
        count = auio.uio_resid;
        /* ... */
        if ((rc=setjmpx(&jbuf)) == 0)
        {
```

```
                rc = (*fp->f_ops->fo_rw)(fp, rw, &auio, ext);
                clrjmpx(&jbuf);
        }
        else if (rc != EINTR)
#ifndef TOKENS
                longjmpx(rc);                   /* certain death!       */
#else /* TOKENS */
        {
                FBTOKEN_UNLOCK(fp);
                longjmpx(rc);                   /* certain death!       */
        }
#endif /* TOKENS */
        /* check for interrupted system call */
        if (rc == EINTR)                        /* interrupted?         */
                if (count == auio.uio_resid)    /* no data transferred? */
                        rc = ERESTART;          /* restart (perhaps)    */
                else
                        rc = 0;                 /* partial transfer     */
        count -= auio.uio_resid;
        fp->f_offset = auio.uio_offset;

        if (rw == UIO_READ) {
                sysinfo.sysread++;
                sysinfo.readch += count;
        }
        else {
                sysinfo.syswrite++;
                sysinfo.writech += count;
        }
        if (countp)
                *countp = count;

#ifdef TOKENS
eout:
        FBTOKEN_UNLOCK(fp);
#endif /* TOKENS */
        return rc;
}
```

A complete list of the places where these need to be applied.

2.1.5.2.2 Hooks in Reopen

This section describes how the reopen code propagates fileblock token state between nodes. **NEEDSWORK:** more intro stuff here.

2.1.5.2.2.1 Export Side

(See the reopen design).

```
In pproc_export_file_state, we  will call f_prep_export
for each open file being exported.

f_prep_export does the following:
    exports the file type (nfs, afs, ...)
    Calls the file-type specific export op (FOP_PREP_EXPORT).
    does file-offset-token setup (calls
    prep_export_fileblock_tokens()).



prep_export_fileblock_tokens does the following:
        if (fp->f_fbtokenp == NULL) {
                fbtoken_init(fp);
        }

        pass fp->f_fbtokenp->fbtk_id to remote node in
        the file-block token data (pointer to location provided
        by f_prep_export.
```

2.1.5.2.2.2 Reopen Side

```
when calling (*fp->fo_reopen)(),

        /* See also reopen.pcode */
        if (file_info.fbtk_id != NULL_FBTKID) {
                fbtkp = find_fileblock_token(file_info.fbtk_id);
                if (fbtkp != NULL && (fp = fbtkp->filep)) {
                        simulate a dup to appropriate fdes;
                        return;
                }
                else {
                        fbtkp->filep = new file structure...
                        fp->f_fbtokenp = fbtkp;
                }
        }

struct fbtoken *
find_fileblock_tokenp(fbtk_id)
{
        if (already have fileblock with that id in our hash table)
```

```
                return a pointer to that fbtoken;

        allocate a new fbtoken
        newfbtoken = new_fbtokenctrlb();

        newfbtoken->fbtk_id = fbtk_id;
        /* lock the new token? */

        do FBTK_USING_FILEBLOCK RPC to fbtk_id.tkm_node,
        so he will add us to the list of using nodes.
        Can "...USING..." fail?  What about when the TKM is trying to
        give manager authority to another node?
        If so, it could fail with "go to this node instead",
        in which case we would just try FBTK_USING... to the new TKM.
        return newfbtoken;
}

RPC (At the TKM)
fbtk_using_fileblock(fbtk_id, using_host)
{
        find the token;
        if didn't find it, return ENOSUCHTOKEN;
        (
                if token is in "moving to new TKM" state,
                        return new TKM with ETRYNEWTKM
        )
        add the using host to the list of hosts on this token;
}

when calling the last fp->fo_close (before freeing the file block),
call FBTK_CLOSING_FILEBLOCK(fp->f_fbtokenp->fbtk_id);


at the TKM:
fbtk_closing_fileblock(fbtk_id, closing host, offset, flags)
{
        find the token;
        if didn't, return ENOSUCHTOKEN;
        (same checks as using_fileblock about trying to move TKM).

        if the node had the token, take the offset and flags
        from this call and stuff them into the token.
```

```
                delete the specified host from the token's list of hosts.
                if only one host is using the token,
                        give that host the token,
                        and tell him to stop using tokens.
                        free token control block
                else if no more hosts are using the token,
                        free the token control block.

                (if closing host was the local host and > 1 using host left,
                        choose a new TKM and start the Switch_TKM protocol
                )
                return OK;
        }
```

2.1.5.3 Fileblock to Token Interface

This module connects the fileblock token hooks to the token mechanism and to the file reopen code.

Some internal macros:

```
#define  _FBTK_PRESENT(fp) \
    ((fp)->f_fbtok.fb_tokenp == NULL || \
        ((fp)->f_fbtok.fb_tokenp->tk_flags & (PRESENT|REVOKED)) \
        == PRESENT

/* Lockcnt in the file structure and in token structure are shadows
 * of each other.  This decides which to use at a given time.
 */
#define _FBLOCKCNT(fp)  (*( fp->f_fbtok.fb_tokenp \
                                ? &(fp->f_fbtok.fb_tokenp->lockcnt) \
                                : &(fp->f_fbtok.fb_lockcnt) ) )

#define _FBTK_HOLD(fp)    (++_FBLOCKCNT(fp))
#define _FBTK_RELE(fp)    (--_FBLOCKCNT(fp))

inline
FBTOKEN_LOCK(file *fp)
{
    if (!FBTK_PRESENT(fp))
        gtk_acquire(tkp);
    _FBTK_HOLD(fp);
    return;
}
```

```
inline
FBTOKEN_UNLOCK(file *fp)
{
    tkp = fp->f_fbtok.f_tokenp;
    ASSERT(tkp == NULL || (tkp->tk_flags & PRESENT));
    _FBTK_RELE(fp);
    if ((tkp = fp->f_fbtok.fb_tokenp) != NULL) {
        if (_FBLOCKCNT(fp) == 0 && (fbtkp->tk_flags&REVOKED)) {
            gtk_release(fbtkp);
        }
    }
}

fbtoken_init(fp)
{
        choose a TKM host. (must support FBtoken protocols.
                            current node is usually the best choice).

        tkp = gtkm_newtoken(preferred manager node);
        fp->f_fbtokenp = tkp;
        tkp->data = (char *)fp;
        tkp->ops = fbtok_ops;
}
```

2.1.5.4 Last Close

NEEDSWORK: Last close of an open file block can have side effects for some kinds of open files (eof on a pipe or socket, dropping carrier on a tty). The easiest way to avoid having the last close of a file occur is to bump the reference count in the file structure. Thus a remote child will keep last close from happening for sockets, fifos, and devs. However, do we want a way to move the fileblock token management AWAY from the socket/fifo/dev node?

NEEDSWORK: If so, what keeps the underlying object from being closed? Who bumps the ref count on that file structure, and when does the ref count get decremented.

NEEDSWORK: Are such files treated differently? How does the file layer know whether the underlying object supports multiple independent opens as identical to a single open with dups? Does this require a flag or return code from the prep-export or something?

NEEDSWORK: Note that if the server code for fifos, sockets, etc. use the file structure, they should be careful not to try to acquire the fileblock token at the server side, since

it was just acquired by their client, and getting the token would really thrash! Instead such server routines should receive f_flag and f_offset (if needed) as parameters, rather than trusting the fields in the file structure at the object's "storage" node.

2.1.5.5 Generic Token Module

The token implementation used by Fileblock Tokens is composed of two modules: the token passing and the meta-token management. The token passing module implements the token passing and fairness protocol. The meta-token management module implements a protocol for choosing a token manager node, and for recovering from the failure of nodes involved with a given token.

Although it was originally hoped that the DFS token mechanism could be used to manage the tokens for fileblock tokens, it falls short in two areas: the token management responsibilities can't be easily moved between nodes, and it can't recover from the loss of the token manager node. Because of these shortcomings, File Block Tokens provide their own (relatively) general token mechanism. However, because of the modularity of this design, any token management scheme which fulfills the requirements of Fileblock Tokens can be used.

For reasons of simplicity, this design describes a single token scheme, rather than a multiple reader/single writer scheme, and there is only a single token which covers both the file flags and the file offset. **NEEDSWORK:** However, the protocol should be extensible to include a more complex token scheme without changing its RPC interface, so a node which implements only the single-token protocol can participate in a cluster which allows multiple tokens.

2.1.5.5.1 Data Structures

NEEDSWORK: Need some intro here... need more meat on DS's. The expanded data structures should be entirely generic. They should include handler ops for caboose code.

```
/* A file block token id implies both the id and the TKM node. */
struct token_id {
        token manager node;
        unique file block id cookie;
};

struct token
{
        /* Token Using Node Parts */
        Token_id
        int lockcnt;        /* replaces ft_tlckcnt, if tokens active */
        int wantcnt;        /* num procs waiting for token to arrive. */
```

```
            flag
                AM_MGR | PRESENT | REVOKING | REQUESTING | RECOVERING
            struct tok_locus * using_nodes;
            struct tok_locus * wanting_nodes;
            struct token_ops * ops;
            caddr_t     *data;
    }
```

2.1.5.5.2 Token Passing

NEEDSWORK: This is where the token fairness is... also calls out to the token's registered arrival, recall, and released handlers.

This module implements the granting and revoking of the generic tokens onto which fileblock tokens are piggy-backed. The operations provided are:

```
/* acquire the token.  Returns with the token "held". */
gtk_acquire(tkp)
{
        ++tkp->wantcnt;
        if (tkp->flag & REQUESTING) {
                /*someone already asked for it.*/
                sleep(&tkp->flag, PTOKWAIT);
                ASSERT(tkp->flag & PRESENT);
                return;
        }
        tkp->flag |= REQUESTING;
        request_token(tkp, &tok_data);
        tkp->flag |= PRESENT;
        tkp->flag &= ~REQUESTING;
        tkp->lockcnt += tkp->wantcnt;
        tkp->wantcnt = 0;
        (*tkp->ops->acquired)(tkp, &tok_data);
}

/*
 * called at the requesting node to get token... caller didn't know
 * if TKM was local or not.
 */
request_token(tkp, datap)
{
        if (tkp->flag & AM_MGR)
                srv_request_token(tkp, local_node, datap);
        else
                rpc_request_token(tkp, local_node, datap);
```

```
}

srv_request_token(tkp, local_node, datap)
{
        if (tkp...
}


gtk_release(tkp)
{
        tkp->flag &= ~(PRESENT|REVOKED);
        if (tkp->wantcnt)
                release_flag |= RE_REQUEST;
        release_token(tkp, local_node, release_flag, &status);
        if (status & GRANTED) {
            /* Is this ok?  Is it a problem for tkm to grant token
             * via an rpc response?  Should be ok, as long as
             * the keep-alive's work.
             */
            tkp->flag |= PRESENT;
            tkp->lockcnt += tkp->wantcnt;
            if (status & WANT_BACK)
                    tkp->flag |= REVOKED;
            (*tkp->ops->acquired)(tkp);
        }
}
```

2.1.5.5.3 Meta Token Manager

This section describes the functioning of the meta-part of the token manager. This module creates and destroys tokens, keeps track of which nodes are can use the token, and deals with recovery when a using or managing node is lost. The module is comprised of four parts:

- create and destroy tokens
- track nodes using each token
- token recovery
- token manager re-election

```
gtkm_newtoken(TKM)
{
        if (TKM is local host) {
                tkp->fbtk_id = unique_fbtk_id();
```

```
                add an entry indicating that the local node
                is using (and has) the token.
                tkp->tk_flags = PRESENT | LOCAL_MGR;
        } else {
                tkp->fbtk_id = remote_new_fb_token(TKM.host);
        }
        add tkp to the file-block-token hash table (keyed on token-id).
        ##?? more here ??
}

unique_fbtk_id()
{
        bump a counter... return a unique token (preferably
        unique across reboots, but critical).
}

RPC
remote_new_fb_token(tkm_host)
{
        perform "Be FileBlock Token Manager" RPC to the host.
        return the token id that is returned, or an error if the
        RPC fails.
}
```

Normal Ops

FBTKM is initial file-block node. a unique token id is assigned (should be unique across reboots).

Sharing nodes get token before using or modifying offset+flags.

When file is not in use at FBTKM (perhaps after a timeout or a number of token ops) want to choose a new tkm.

Also need to handle loss of tkm gracefully.

Clearly switching tkm and recovery should be largely common code and protocol.

To do this, the TOKEN CARRIES the list of USING NODES.

This is cheap, and since the token must be available to recover cleanly anyway, imperfect recovery if token is lost is not made much worse in most cases.

ISSUE: limit on the number of copies of the file descriptor (i.e. on the number of using nodes? can we degrade slightly if this limit is exceeded?)

ISSUE: (minor) policy of when the token manager should move is an interesting issue: could do as soon as tkm node closes the file, or could wait a while... would probably depend on how many nodes are using the file at that time. If its just one, then move immediately. Otherwise might wait to see if the file is really being actively used.

TKM's using-node-set updates must be atomic w.r.t. sending back set with token.

NEEDSWORK: FBTK Suppression When the FBTK notices that the set of using nodes is reduced to 1 node, he proposes to that node that the token overhead be eliminated. If the node accepts, the token is made implicit at that token holding node. If the number of using nodes is reduced to 0, the token is simply freed.

2.1.5.5.3.1 Recovery

This design attempts to provide full recovery from most failures, and partial recovery in all cases. The various failure modes possible are:

- Loss of a participating node which didn't hold a token.
- Loss of a node which held a token (a.k.a. loss of token).
- Loss of the token manager node.
- Loss of the token manager node and the token.

kicked off when tkm decides to give up its token-managing, (usually because it no longer has the file open locally). or when some node notices that the token manager is gone (node-down).

Old TKM Selects new tkm (current token holder, if any, otherwise a random using node for this token). Mark token as "in motion" (to prevent new requests or adding new nodes). Sends BecomeTKM RPC to new TKM, passing current list of token using nodes. (TKM rejects subsequent ops on the token, since he's not the manager anymore).

New TKM Create the token manager data structures. We have a "hint" of the using nodes, but don't have confirmation yet. (particularly in lght of the problem with keeping an accurate list of nodes at the non-TKM nodes) Performs NewTKM RPC to each node on the list of using nodes for the token that he's the new TKM. Each node which returns an ACCEPT code to the RPC is added to the list of using nodes for the token. The response may also add to the hint list of nodes to query, to handle the race between an rexec to a new node and the movement of the token manager.

ISSUE: is there a problem figuring out when to take nodes OUT of the list of possible using nodes? Could maintain a guess set and a known set. When receive a new set from the TKM, we do guess &= ~known... hence getting rid of node if TKM knew of

it. However, if the tkm found out about the node we added to our guess list, but didn't tell us until that node had gone out of our guess list, we wouldn't delete it. **NEEDSWORK:** We only add nodes to the guess-set at a US when exporting the file.

Other

case 1: A token related RPC is rejected by the node we thought was the token manager (but he hasn't crashed). assume that some other node is becoming the new TKM; wait a reasonable amount of time for the TKM to change, then ... fall through to case 2:

case 2: No response/node-down on a token related RPC to old TKM. Perform a topchg like protocol with the rest of the known token using nodes. (see later section about TKM-down recovery).

2.1.5.5.3.1.1 FBTKM-down Recovery

When a node notices that the FBTKM has "crashed" (connection is lost), the remaining using nodes must be coordinated. This is done via a protocol vaguely similar to our current topchg: Since each node knows a subset of the list of token using nodes, these subsets are merged. A new token manager is elected (node holding token at time of TKM crash, or least loaded node?). The new TKM then performs the protocol described above under "New TKM".

NEEDSWORK: merge protocol, and election of new TKM.

Issue: Does the TKM move if there are requests pending?

Its not hard to do:

enter "in-motion" state, wakeup kprocs waiting for tokens. This automatically rejects each pending request.

Also simplifies things, since it may not be easy to determine if there are requests pending.

Issue: Does adding a using node to TKM's list require recalling the token? Doing so would have the advantage of increasing the chance that the token would carry a complete list of using nodes. However, it wouldn't make it certain, and we have to handle the case where the token's using-node-set is incomplete anyway, so don't bother (would slow things down even more).

2.1.5.5.3.2 RPC's

Be_TKM RPC:

Like token grant:

token id

using node list (a guess)

Does this imply the token is granted to new TKM?

(can it be refused?)

am_new_TKM RPC:
tok id
tok using node list (IN-OUT!)
Returns more nodes to poll,
did called node have the token?

Recovery — TKM down

node which first notices TKM down becomes candidate TKM

Candidate TKM:
Poll known using nodes with "am_new_TKM" RPC.
if no using nodes are found, must wait long enough for other nodes (which may have larger US lists) to notice the TKM is down.

ASSERT: all existing using nodes can find each other if they put their info together. i.e., if they all notice that the TKM is down, they CAN find each other and compose a complete list of using-nodes.
(casual proof: assume sequenced access to TKM's US list. assume not: hence there exist two nodes that have no "link" between them (neither appears in other's US list). Since one or other must have received token first, that node MUST appear in the later node's US list.)

ISSUE: what happens if we have two candidate TKM's collecting nodes!?!? NEEDSWORK!!!

2.1.5.6 General Assumptions

node-down notification

distinction between rejection and timeout or stale handle.

2.1.5.7 Performance

expected with this design

potential optimizations

Batch multiple token messages when doing large scale PT ops ("I'm using the following fileblocks whose tokens you manage") to save round trips.

2.1.5.8 Packaging

As the design shows, fbtokens can be broken into several parts. Of those parts, only those enumerated in the "Hooks in the Base" section of this document must be present. All the other parts can be installed along with the rest of DEE. However, **FUSION**'s reopen does require all of fileblock tokens in order to operate completely correctly.

Another aspect of this design is that the separation into 5 major modules is done along lines which will allow replacement of meaningful pieces of the implementation. For example, an implementation which wanted a different token passing policy (?to reduce thrashing) could replace the token passing module. An implementation which wanted to provide a more complex recovery protocol could provide a replacement for the "Meta-Manager" module.

Finally, the token mechanism provided for fileblock tokens is general enough so that it could easily be used to support other uses of tokens which need similar recovery properties.

2.1.6 File Reopen and Lock Inheritance

2.1.6.1 Overview

This section of the FUSION Design Specification describes the method FUSION uses to allow the various flavors of open files to be moved between nodes to support full Process Transparency. It explains the software structure which provides this functionality, the changes this imposes on a base operating system, and the data structures used.

This section does not describe the RPC's actually used to send the file state between nodes. Instead, it describes the mechanism for unambiguously representing that state in a format which can be moved between nodes via RPC's described in other parts of the FUSION Design Specification. The mechanisms provided here (generically known as "file reopen") are used in a number of places throughout the FUSION architecture, including, but not limited to, rfork, rexec, and migrate.

A UNIX process's file related state consists of more than just its set of open file descriptors. It also includes a current working directory vnode, a current root directory vnode, and a set of vnodes associated with its current executable and any attached shared libraries. In addition, each individual open file can have associated record locks, flags, and an offset value. This design provides ways of moving as little as a single vnode, as much as a process's entire file related state, or anything meaningful in between.

This design provides the required functionality with minimal changes to the base operating system and remote file access protocols (AFS and NFS). It handles all current file related objects and state, and its hierarchical, modular structure will readily accommodate future extensions.

See also DE PTrans Functional Specification, Sections 3.5.3.2 and 3.4.10.

2.1.6.2 Reopening Files — Top Level

Where the words "export" and "import" are used in sections related to file reopen, they mean:

Export

> To package the complete state of an object so it can be sent to another node. In practice this means filling in a strongly-typed data structure, based on the state of the object. Export does NOT imply destruction of the object.

Import

> To reproduce an object which exists on another node by decoding the data structures produced by an "export". This is most often called "reopen" in this document.

When performing one of the major FUSION Process Transparency operations (migrate, rexec, or rfork) which creates a process on a destination host, the originating process

exports all of its file-related state. This consists of:

- current working directory and root vnodes
- open files and any associated record locks
- file offset tokens for shared open-files

When process file state is decomposed, shows its hierarchical structure. For example:

```
open-file is
    file descriptor number
    per-open-file flags
    pointer to a file structure

file structure is
    flags
    file offset
    file type
    file-type dependent pointer

file-type dependent pointer points at
    vnode
or
    socket

vnode is
    VFS pointer
    ops pointer
    VFS-type dependent data
```

The **FUSION** Reopen Design parallels this hierarchy, taking advantage, where possible, of the object oriented nature of the file and vnode interfaces. Thus each file or vnode type "knows" how to export and import itself. This also allows the internal workings of a given level in the hierarchy to be opaque to higher levels.

The overall picture is of parallel hierarchies of export and import operations. A given export or import operation may invoke the import or export operation of next level below itself. The export and reopen operations for a given level (file, vnode, record lock, socket...) are peers; data produced by the export can ONLY be decoded by the corresponding import operation.

2.1.6.2.1 export_process_file_state

Export_process_file_state() is the top level function which invokes the mechanisms which allow the file state to be reproduced remotely. Its "output" is a potentially large, multi-level data structure (a proc_file_state_t) which can be used as an argument to any one of the process-transparency RPC's. It fills in this data structure by passing

pointers to the subcomponents of the data structure to the export "ops" of the process's various vnodes and open files. (The details of the export "ops" are explained later in this chapter.)

Pseudo-code: (our prototype contains considerably more detail than this, but this is the basic idea).

```
export_process_file_state(
    proc_file_state_t *pfsp,        /* output file state */
    ptrans_op_t     op_type         /* rexec, migrate, rfork */
)
{
    export current working directory into pfsp->cwd.
    export root directory into pfsp->root.
    for each open file descriptor,
        call f_prep_export(fd, pfsp->open_files[], op_type)
        to export the struct file's internal state.
}
```

2.1.6.2.2 import_process_file_state

"Import_process_file_state()" is the converse of export_process_file_state. Its function is to decode the data structure produced by the export, and recreate all open files and file related state for the process. A pointer to the target user structure is passed to allow flexibility in the implementation of the process transparency operations which call this function; In the current **FUSION** prototype, import_process_file_state is called in the parent of the process being created.

Pseudo-code:

```
void
import_process_file_state(
        pt_open_files_info_t *fstate;
        struct user *up         /* target user struct for info */
)
{
        call vnode_reopen to reopen the current working directory.
        if rootdir specified
                call vnode_reopen to reopen the current root.
        /*
         * Reopen all the process's files, and fill in the target
         * process's u.u_ofiles (up->u_ofiles).
         */
        for each open file exported,
                call f_reopen(fstate.of_ofiles[this file])
                to reopen the given up->u_ofile[].
}
```

The Current Working Directory vnode and Current Root vnode (if specified) vnodes are reestablished via an extended VFS operation (VFS_XVGET) which will be described in this document's section on vnode recreation.

2.1.6.3 Struct-File Recreation

Part of reproducing a process's file state on another node involves recreating an each open file (i.e., struct file). This section will discuss the operations which perform this feat, as well as the changes they require to the set of file operations. Since there is more than one type of file, it will also discuss the requirements placed on the new file operations for each file type.

2.1.6.3.1 f_prep_export

The function f_prep_export() is responsible for exporting a struct file. This means both the file's internal state, and its "identity". The file's external state is exported via the file's own PREP_EXPORT operation. The file's "identity" (important to support UNIX semantics for files shared by multiple processes) is exported via the file-offset token package, initiated here by f_prep_export (File Offset Tokens are more fully discussed in their own section of this design.)

The data structure built by f_prep_export() is approximately as follows (see the prototype's NIDL for the more complex truth):

```
{
    file-descriptor,                (0, 1, 2....)
    file-block-token-id,
    file-type,
    file-type specific "reopen handle"
}
```

The following is a simplified version of f_prep_export from the **FUSION** prototype.

```
f_prep_export(int fd, fdes_reopen_handle_t *fdrhp, op_type_t op_type)
{
    file_obj_reopen_handle_t *forhp;

    fdrhp->fdh_poflags = u.u_pofile(fd);
    if (op_type == EXEC && fd is close-on-exec)
        return;
    fdrhp->fdh_fdes_num = fd;
    forhp = &fdrhp->fdrh_handle_union;
    forhp->frh_ftype = f_export_file_type(fp->f_type);
    FOP_PREP_EXPORT(fp, &forhp->frh_union);
    prep_export_fileblock_tokens(fp, &ofrhp->ofrh_fbtok_id);
}
```

2.1.6.3.2 f_reopen

The function f_reopen interprets the fdes_reopen_handle_t created by f_prep_export to reproduce an identical open file. Due to its interaction with FUSION File Offset Tokens, this also reconnects multiple file descriptors referring to a single struct file (i.e. "dups").

```
int
    f_reopen(
        struct user *up,
        fdes_reopen_handle_t *fdrhp
    )
    {
        fbtoken_tcb_t       *fbtkp;
        struct file *fp;

        fbtkp = get_fileblock_token(&fdrhp->fdh_file.ofrh_fbtok_id);
        if (fp = fbtk_get_filep(fbtkp)) {
            /* File already open locally: really easy */
            set_ufd(up, fdrhp->fdh_fdes_num, fp, fdrhp->fdh_fdes_flags);
            fp->f_count++;
            return 0;
        }
        /* Translate the file-type from net representation to local */
        f_type = import_ftype(fdrhp->fdh_file.ofrh_handle_union.frh_ftype);
        fpalloc((struct vnode *)NULL,
                fdrhp->fdh_fdes_flags,          /* Per-open flags */
                f_type,                         /* file type (sock or vnode) */
                ftype_to_file_ops(f_type),
                &fp);
        /* Let file-block token code know about this new file & f_offset */
        fbtk_set_filep(fbtkp, fp,
                fbtkp->fbtk_token.fb_f_flag, fbtkp->fbtk_token.fb_f_offset);
        err = FOP_REOPEN(fp, &fdrhp->fdh_file.ofrh_handle_union.frh_union);
        set_ufd(up, fdrhp->fdh_fdes_num, fdrhp->fdh_fdes_flags, fp);
        return err;
    }
```

2.1.6.3.3 file types

In order to decide which file ops a file uses to decide which file-reopen-op to call, file reopen only needs to know the "file-type" of the file. In AIX 3.1 has a numeric file-type in each file table entry. This is are currently:

```
DTYPE_VNODE
DTYPE_SOCKET
DTYPE_GNODE
```

DTYPE_GNODE is only used internally by certain device access cases such as

reading a device to mount it, so this is never exported. This set of file types doesn't currently conflict with any of OSF's exported file types. However, to ensure interoperability with other operating systems in the future, those file-types are exported in an OS independent representation. This is translation is performed by the routines "f_export_file_type()", and "f_import_file_type".

2.1.6.4 Extended File Ops

The file structure in VFS based kernels contains a pointer to the set of operations which can be done on that file, and each unique type of file (currently only sockets and vnodes) must provide its own set of these operations. These are normally:

```
fo_rw
fo_ioctl
fo_select
fo_close
fo_stat
```

To allow the export_process_file_state to export and remotely reopen files, we add the following operations to this set:

```
fo_prepare_for_export
fo_reopen
```

2.1.6.4.1 fo_prepare_for_export

The "fo_prepare_for_export" for a given file type is invoked via the FOP_PREP_EXPORT macro, and has the following responsibilities:

1. perform whatever modifications are necessary to the local file and its underlying data structures to allow the file to be shared with a remote host.
2. prepare any file locks on this file for export.
3. provide a "reopen handle" which can be used at the destination host to remote host to reopen the file.
4. provide an indication of what file-locks need to be reacquired on the destination node.

2.1.6.4.2 fo_reopen

The "fo_reopen" operation for a particular type of file, invoked by the FOP_REOPEN macro, does the reverse: it takes a prototype file block and "reopen handle", and performs the magic appropriate to the file type to make the file block refer to the same object as the original open file, with all its locks and other state intact.

2.1.6.4.3 vno_prep_export

For vnode based files, the export file operation is implemented by the function "vno_prep_export()". This function just finds the file's vnode, and passes it to vnode_prep_export, which knows how to export a vnode (and is separately callable for use in other parts of FUSION).

```
vno_prep_export(
        struct file *fp,
        frh_union_t *frhup      /* File Reopen Handle union */
)
{
        vnode_reopen_handle_t *vrhp;
        struct vnode *vp;

        vrhp = &frhup->frhu_vnode_h;
        vp = fp->f_vnode;
        vnode_prep_export(vp, fp->f_flag, vrhp);
        if (process has file locks, and this file descr's
            pofile flags says this file may have locks)
                VNOP_PREP_EXPORT_LOCKS(vp, fp,
                                &vrhp->vrh_vnode_reopen_data);
}
```

2.1.6.4.4 vno_reopen

For vnode based files, the reopen file operation is implemented by the function "vno_reopen". This simply uses the new Vnode and lock recreation mechanisms provided by **FUSION** to complete the initialization of a partially filled in file structure. See Vnode Recreation, 2.1.6.7, and Relock, 2.1.6.5.2, for the design of these mechanisms.

```
vno_reopen(struct file *fp, vnode_reopen_handle_t *vnode_rhp)
{
        int err;

        err = vnode_reopen(vnode_rhp, fp->f_flag,
                        &fp->f_vnode, &fp->f_vinfo);
        if (err)
                return err;
        err = VOP_RELOCK(fp->f_vnode, &vnode_rhp->vrh_file_locks);
        return err;
}
```

2.1.6.5 File Locks

Several substantially different kinds of file locking currently coexist in the UNIX world. These include BSD style "flock" and SVID style fcntl locks. "Lockf" from /usr/group can be implemented as a subset of fcntl locks, so lockf doesn't affect this design. Microsoft also specifies XENIX and DOS locks, both of which are enforced locks.

Of these flavors, only fcntl locks are currently accepted by POSIX and available in this design's primary target OS. Flock style locks will fit within the **FUSION** process

transparency architecture, but the detailed design of their export-lock/relock implementation is not included here.

To implement its file-lock related subcommands, the typical fcntl system call (like AIX 3.1) does some correctness checking, then builds a parameter block and invokes the file's vnode's VOP_LOCKCTL operation. The implementation of the locking data structures is (in principle) left up to the individual vfs implementation.

For the purposes of lock inheritance across node boundaries, we only need to consider file systems which can export files and locks between nodes. These are

NFS
: This requires the **FUSION** NFS extensions. See section 3.1.2, NFS Interoperability.

DFS exported file systems
: This means both normal native file systems exported through a "glue" layer, and Episode file systems.

DFS imported file systems
: File systems managed by the DFS Cache Manager.

Vnodes of these VFS types must now support two new vnode operations whose interfaces are described below. Design for how these operations will be implemented in these VFS's can be found in the appropriate NFS and DFS sections of this design specification.

2.1.6.5.1 Prep_export_locks

The prep_export_locks vnode operation is performed by vno_prep_export, the prep-export file op for vnode based files. Like other prep-export operations, it is done at the originating node of a process performing a process transparency operation which requires lock inheritance. Under current (fcntl/lockf) lock semantics, rexec and migrate require relock, but this could easily be changed to support any reasonable lock inheritance scheme in the future.

This operation is actually invoked as VOP_PREP_EXPORT_LOCKS, as shown in the pseudo-spec below (actually a macro, of course):

```
VOP_PREP_EXPORT_LOCKS(
        IN struct vnode *vp,
        IN ident,       /* tagged: pid or file id */
        OUT struct file_locks *locks);
```

Each implementation of the operation must build and return (via *locks) a data structure which describes any locks that the ident held on the vnode. It must also do whatever token magic is necessary to allow exclusive locks to be reacquired remotely (since the local locks won't have been removed by then). Ident is left flexible here since it seems possible that it may eventually be necessary to provide locks keyed by independent-open (file table slot) (like flock), in addition to the current POSIX locks,

keyed by pid.

Fcntl lock's are lost on exit of the locking process, yet the implementation of rexec and migrate depend on the ability of the originating process to exit through relatively normal means. This apparent contradiction is handled in DFS and the FUSION version of NFS locking by actually having two conflicting independent locks granted at the same time for the file. The exiting of the original physical process at originating node on successful migrate, or the exiting of the destination physical process on migrate failure thus only cause one of the two conflicting sets of locks to be released; the correct regions of the file remain locked continuously in either case.

Correctly implementing flock style locks with rfork would require changes in DFS. DFS would need to grant lock tokens to nodes based on an Ident, not just based on normal token conflict rules. However, since true flock style locks aren't currently called for in POSIX or other standards, and since flock style locks are currently emulated incorrectly by most OS's which support lockf/fcntl locks, this is unlikely to be an issue.

2.1.6.5.2 Relock

At the destination node of a migrate or rexec, the vno_reopen will invoke the RELOCK vnode operation of any vnode based files which need locks reestablished.

```
VOP_RELOCK(
        IN struct vnode *vp,
        IN struct file_locks *locks);
```

This operation must interpret the locks specified and perform any internal magic necessary to reacquire them. Again, these locks will be reacquired BEFORE they are released at the originating node, so the VFS must allow this, given the appropriate verifiers placed in the file_locks structure by its peer, pre_export_locks().

2.1.6.6 Sockets — Non-Vnode Based Files

For the migration of file descriptors for non-VNODE based files, the file type must have its own versions of "fo_prep_export" and "fo_reopen" which do what is necessary to setup the file block and its underlying data structures.

For example, the "fo_prep_export" for socket-type files operation would create its socket-side light-weight server process, and hand the original socket to that process. The "fo_reopen" for sockets would replace the file's set of "file-ops" with a pointer to the set of remote-socket file ops, and would change the open file's f_type on the destination host to indicate that it is a "remote-socket" rather than just a socket. The file ops for remote-sockets would, of course, know how to talk to the socket-host's server process to do the actual socket operations.

Note that another interesting thing about the file-ops for remote sockets is that its "fo_prep_export" function could be different than that for a normal socket: it would not be desirable to have the socket data go through multiple hops just because the

process which has the socket open had been migrated more than once, so the data sent to the destination side would indicate the original socket host and the id of the socket. (Of course, the normal socket migration messages could also be general enough to handle this case).

2.1.6.7 Vnode Recreation

When an open file being moved is one whose implementation uses vnodes, its file-type specific routines export and reopen file ops (vno_prep_export and vno_reopen) invoke new vnode ops and VFS ops which have been added to support Process Transparency. These operations are:

VOP_PREP_EXPORT

Creates a "reopen handle" which can be used at the destination to reopen the vnode. This also does whatever is necessary for the VFS type to allow the reopen to be done at the destination (magic with exclusive mode tokens for AFS, for example).

VOP_PREP_EXPORT_LOCKS

See Relock, section 2.1.6.5.2.

vfs_xvget

This VFS op (not Vnode op!) uses the "reopen handle" produced at the origin node to get a held vnode in the specified VFS. This closely resembles the existing vfs op "vfs_vget". It is a separate op mainly due to the fact that the "standard" vget operation varies slightly between different Operating Systems. In any case, vfs_xvget is straight forward to write in terms of an existing vfs_vget.

VNOP_REOPEN

This VNODE op resembles the standard open vnode op: it "opens" a vnode, given a pointer to the vnode and some information about the open. It is different in that no access checking is done, and conflicts due to exclusive open modes are ignored (since the file was already open).

VNOP_RELOCK

See Relock, section 2.1.6.5.2.

2.1.6.7.1 vnode_prep_export

Although the prep_export and reopen Vnode/VFS ops implement most of vnode reopen, they need help to do the job; the prep_export op doesn't know how to export the complete logical identity of its own VFS, and since reopen is a VFS operation, the VFS must be known before it can be called. I.e., since each VFS type can have its own format of vnode handle, some mechanism must wrap these functions to identify the format of the handle.

To solve this problem, these ops are always invoked by the VFS independent functions vnode_prep_export and vnode_reopen. These two functions are also called

independent of open file structs to allow recreation of a processes current working directory, text and shared library vnodes, and from any other place in **FUSION** which needs to unambiguously and cleanly transmit the identity of a Vnode to another node.

```
/*
 * Invoke the export mechanism for a vnode.
 * Also exports the vfs_type so the remote reopen code knows how to
 * interpret the vnode_reopen_handle to find the the right vfs.
 */
int
vnode_prep_export(struct vnode *vp, int modes, vnode_reopen_handle_t *vrhp)
{
        int err;

        err = VNOP_PREP_EXPORT(vp, modes, &vrhp->vrh_vnode_reopen_data);
        if (err == 0)
                vrhp->vrh_vfs_type = vp->v_vfsp->vfs_type;
        else
                vrhp->vrh_vfs_type = MNT_BADVFS;
        return err;
}
```

2.1.6.7.2 vnode_reopen

This function unwraps the VFS handle, and uses information in its "wrapper" to find or create the correct incore VFS structure. When it has done so, the VFS's reopen operation is called to recreate the desired vnode. Naming VFS's is discussed in its own section.

```
int
vnode_reopen(
        vnode_reopen_handle_t *vnode_rhp,
        long flags,
        struct vnode **vpp,
        caddr_t *vinfop
)
{
    struct vfs *vfsp;
    extern struct vfs *nfs_find_vfs(struct nfs_reopen_data *);
    extern struct vfs *dfs_find_vfs(struct nfs_reopen_data *);
    caddr_t dummy_vinfo;

    if (vinfop == NULL)
        vinfop = &dummy_vinfo;
    switch (vnode_rhp->vrh_vfs_type) {
    case MNT_NFS:
```

```
            /* Scan the vfs for the one which has this handle */
            vfsp = nfs_find_vfs(&vnode_rhp->vrh_vnode_reopen_data);
            break;
        default:
            return EINVAL;
        case MNT_DFS:
        case MNT_DFS_GLUE:
            /* find or create the appropriate logical volume. */
            vfsp = dfs_find_vfs(&vnode_rhp->vrh_vnode_reopen_data);
            break;
        }
        err = VFS_XVGET(vfsp, vnode_rhp->vrh_vnode_reopen_data, vpp);
        if (err)
            return err;
        if (IS_SPECIAL_FILE(*vpp)) {
            /* Allow device & fifo code to setup the vnode's ops */
            special_vnops(*vpp);
        }
        if (flags) {
            err = VNOP_REOPEN(*vpp, flags, ext, vinfop);
            if (err)
                VNOP_VRELE(*vpp);
        }
        return err;
    }
```

To export the process's current working directory, uses one of the extended vnode operations which will be discussed in the section of his document specifically about migrating vnode based files (VN_PREP_EXPORT).

The vnode export and reopen ops will also likely be used to reacquire a processes text and shared library vnodes on process migration or rfork (remote-fork). See the appropriate area of those design sections for more information about text and library reopen.

2.1.6.7.3 VFS Naming

Naming a specific vnode given a file system is straight forward, though the precise format of the file-handles varies between file system types. The difficult part about migrating open files is naming the file system. In particular, we need to name the specific VFS, which includes the place where the actual file system is mounted in the case of AIX 3.1, which can mount the same piece of a file system on more than one mount point.

2.1.6.7.3.1 Clustered VFS Naming

In DCE without **FUSION**, information about a file system is normally brought into core on a specific host when a process on the host traverse down across a mount or junction point into the file system. The file system will then be located either via the VLDB or the Cluster Mount Server, depending on whether it is a junction point or an "/etc/mount" mount, respectively.

However Process Transparency adds the twist that it may be necessary to reopen a vnode in a particular VFS when the neither the VFS nor the VFS it is mounted on are in core at the destination host.

The logical first step in being able to produce the needed VFS is to be able to name the VFS. In a **FUSION** cluster, there are two possible kinds of names for VFSs, though there could conceivably be arbitrarily many.

Logical VFS-ID

If the file system is mounted using an "/etc/mount" (NFS or otherwise), the mount is given a VFSid by the AIX 3.1 mount code. However, AIX 3.1 VFSids are unique only at the node where they are generated, so the following tuple will be used:

<Mounting-Node, VFSid>

to name these mounts. In clustered operation, the the VFSid could be assigned by the CMS. The actual network location of the file system (uuid?), its root vnode, and the the inode the file system is mounted on (i.e., where it appears in the cluster-wide name space) can either be obtained from the Cluster Mount Server, or they can be pulled from the originating node.

Interrogating the originating node:

- Involves fewer nodes, and is hence more robust
- Facilitates out-of-cluster operations, since it may be difficult or impossible for a host to interrogate the CMS of a cluster he isn't a member of.
- Can work in inter-cluster or non-clustered operation.
- Has potential security problem, since doesn't require extraordinary privilege to cause mounts to appear on a node.

Interrogating the CMS:

- May be more likely to be "correct" (fewer races).
- Gives the CMS more control over and information about which hosts know of which mounts, facilitating unmounts.

Given the above, this design assumes that the mount information is pulled from the CMS. If later code is added to support pulling the information from the origination node, the security issue will need to be dealt with.

VOL-ID

When a file system appears in the DCE name space due to a junction-point (aka "funny symlink"), the file system will not have a VFS-id, but will have a Volume ID which the VLDB will be able to turn into the location of the file system. Hence we will use

<VOLUME-ID>

to name these VFS's.

However, note that the the important distinction (currently) between a junction and a mount is that a junction point works like a symbolic link with respect to ".." traversal out of its root: ".." across a junction point yields the "preferred" mount point (specified when the file system is "attached"?), rather than the original path from "/".

Again, there is a choice as to whether to pull information about the volume from the originating node or to get the needed information about the volume from the VLDB. In this design, the VLDB will always be used, otherwise the local VLDB cache might never get filled in for some volumes.

NOTE: if the destination node doesn't know about the CELL of a volume, it will need to ask either the origin node or the CMS about that cell (to add the cell to its cell table). This should probably be done in cm_GetCell() when the cell is not found. (or the afs_reopen op could try to do a cm_TryBind if the reopen fails).

Now that we know the ways that VFS's can be named, we need a way to search the set of incore VFS's at a host for a particular name. The current AIX 3.1 code has a similar operation which we will use as a base for our extended "find_vfs".

If we're unlucky and the VFS doesn't already exist at the destination host, we need to create and setup a VFS, more or less as if a local process had traversed the Cluster's name space into the desired VFS.

One difference between a VFS created for a migrating file-open and and one created via an actual path traversal is that the path traversal will automatically bring the path of VFS's to which the desired VFS is attached into core. Hence the target VFS's mounted-on-vnode pointer will point to a real vnode, which will point to a real VFS, and so on, up to the root.

While creating this chain would usually be possible, it would most often be a waste, and it could add considerable overhead to normal Ptrans-OPS. In addition, it will be impossible to create that chain when the process is migrating outside the cluster, since

there will NOT be a path from the file system to the "alien" root, or when the mounted on vfs has gone away due to an unmount or a node failure.

Instead of always building the complete VFS chain, we propose that:

> The VFS points at a "magic" mounted-on vnode which contains information about how to find the mount-point when a ".." is performed.

The code in the base OS's lookup loop will need to be enhanced to allow **FUSION** to catch and properly handle '..'.

The details of how to perform this pseudo-mount need to be worked out, and they will vary between VFS types (particularly between AFS and those VFS types already supported by AIX 3.1). Hopefully this will basically boil down to:

- create a place-holder mount-point vnode (the vnode to be hidden).
- if using "magic" mounted on vnode scheme mentioned above, Initialize this vnode with private data about how to find the real mounted on vnode and VFS.
- perform the guts of a "normal" mount operation to mount the vfs on the placeholder vnode.

2.1.6.7.3.2 Non-Clustered VFS Naming

When performing an rexec or similar operation between two nodes which are not in the same cluster (or not in clusters at all), there is no CMS to query about mount information. When this occurs, files can only be reopened if their VFS is otherwise present or obtainable.

For DFS junctions, since the volume id is part of each file's "fid", a file can be reopened if its volume is already present at the destination node, or if there is a VLDB available and the VLDB knows about the volume.

For /etc/mounts, the actual VFS can not be recreated without a CMS. This is not fatal since without a CMS, there isn't general name space transparency. Instead, since a file's physical volume ID is sent to the destination as part of its reopen handle, the non-clustered code will scan for any anchored VFS with the same physical volume. If it fails to find such a VFS, it will create a "floating" VFS for the volume, and do the reopen in that VFS. If the VFS or volume can't be recreated, or its server can't be contacted, the reopen fails.

This policy allows an administrator to establish a "cluster-like" environment by doing the majority of the /etc/mounts in common for his set of machines. When a process moves between these machines, it will usually find the "equivalent" VFS for each of its open files and current working directory.

2.1.6.7.4 Specific File System Types

This section briefly outlines the changes required by **FUSION** reopen in each of the network visible types of file systems in DCE. More detail for each type of file system can be found in sections specifically about them (DFS in the next section, FRFS and

NFS in their own sections elsewhere in this design document).

NFS

The NFS Token Manager will need to support the new extended vnode operations for open-vnode migration. The mechanism is discussed in detail in the "NFS Token Manager Design".

TRFS (Tightly Replicated File System)

TRFS will also need to support the reopen operations. It may be necessary to change storage nodes when migrating an open file in a TRFS, but hopefully this will be supported without extra work since changing storage nodes is already required functionality. This area may need more work, however.

AFS

AFS refers to Episode file systems, and to UFS or journaled file systems which are "exported" using the Transarc's PFS glue layer.

2.1.6.8 AFS changes

This section describes the changes needed AFS to support Process Transparency.

FUSION's remote-fork, run, remote-exec, and process migration operations put special requirements on DFS file system protocol exporters, and on the generic Token manager, which are not present without **FUSION**. In particular, some tokens (guarantees) which formerly could never be granted to more than one client at a time may now need to be granted to several clients at once. In addition, it may be necessary to move members of the Lock class of tokens "atomically" from one host to another.

When process transparency operations are used, the rights which are associated with a single open, lock, or other operation on a file may need to be moved or shared with one or more other clients. This is basically because processes on two or more clients must be able to view a given open as if the processes were on the same client.

2.1.6.8.1 Token Types

The AFS Token Manager supports at least the following types of tokens.

Open:

read
write
exclusive
shared-read

Data:

read
write

Lock:

read lock range
write lock range

stat(aka sync):

read
write

Most of the token types can be "shared" using the existing token services:

- Open-Write and Open-Read tokens can already be granted to more than one client at a time. Hence recreating these tokens on another client isn't a problem.
- Read & Write Data and Read & Write Status are all short term tokens, so there should never be a problem with revoking them to allow operations from another client. There is no reason to explicitly obtain these tokens when migrating.

However, the following tokens will need to be used in ways that the currently proposed TKM does not support.

- Open-Exclusive (O_NSHARE): The destination host of a remote-process operation like rfork() or run() will need to "reopen" the file in exclusive mode, even though an exclusive mode open token has already been granted.
- Open-Shared-Read: Because AIX-V3 allows opening a file for Shared-Read AND Write (O_RSHARE|O_WRONLY) it must be possible for the remote-proc-op destination host to reopen the file with the same mode, even though that mode is self-conflicting.
- File Locks (Read & Write): The TKM's Lock tokens will be presumably be used as part of the implementation of both lockf() and flock() (i.e., lockfx()). We will need to transfer such locks between hosts when a process holding locks migrates.

2.1.6.8.2 Proposed Changes

The proposed extensions to VFS+ consist of four new operations: "prepare-for-export", "reopen", "prepare-for-lock-export" and "relock". The "prepare" calls would export the tokenID from the original open or lock, and the reopen/relock ops would provide these token ID's as "proof" that the reopen/relock is for good cause and should be allowed, although the open or lock would violate normal exclusivity rules.

These new operations need to be supported in the Client Cache Manager's VN-Ops, in the AFS Protocol Exporter, and in the PFS-Glue's VN-Ops.

Below we outline the requirements placed on the AFS components (CM, the PFS glue, the PE and the TKM) by the 3 new VNODE ops and 1 new VFS op.

Cache Manager:

a. For "prepare" on exclusive opens and write-locks, the source side cache manager must provide a way to mark the original exclusive token as "invalidate

on last release". If this were not done on either the source, receiving or both nodes, subsequent independent opens on either machines could incorrectly believe they had exclusive access. Invalidate-on-last-close is provided by the Cache Manager's CM_TOKENLIST_RETURNL and CM_TOKENLIST_RETURNO flags.

b. In addition, since a single write-lock token may be used for locks on more than one subrange, any write-lock token being exported must be subdivided so that the token range exactly matches the lock range. It can then be marked as "invalidate-on-last-release" without harmful side effects. To perform this subdivision, "cm_prep_export()" will need to perform a new call (Subdivide-Lock) to the Protocol Exporter.

   Subdivide-Lock: this is almost trivial, just by re-requesting the token parts you've already got. The TKM interface seems to allocate a new token ID, then grant it automatically if you've already got the tokens.

PFS Glue:

a. For "prepare" on exclusive opens or write locks, the glue must provide a way to mark the original exclusive token as "invalidate on last release" (see Cache Manager, above). Invalidate (or release) token on last release seems to be accomplished by the TS_FLAG_WAITING bit in the current glue-token-cache code.

b. Any lock tokens will need to be subdivided to exactly match the range actually locked, as for the CM.

PE:

a. The PE must support a GetToken operation which takes a tokenid as an argument, and perhaps a special flag indicating that the specified token should be granted without checking for conflicting tokens.

   Since AFS_GetToken() already accepts a token id as part of one of its arguments, reopen can use that interface. The "don't-check-conflicts" flag will only need to be added if the token-id passed to AFS_GetToken is normally filled in. Otherwise we can use the valid token-id as an indication to the TKM not to check for conflicts.

b. If the simple solution for Subdivide-Lock mentioned above under CM proves to be unworkable, the PE will need to export an operation to subdivide a (write-lock) token. This would just be passed to the TKM.

TKM:

a. An interface is needed which allows the PE or the PFS Glue to request a token which would normally conflict with already granted tokens. The existing routine tkm_GetToken() will be modified to allow this when its flag argument includes

the "TKM_NO_CONFLICT_CHECK" bit. This flag must in turn be passed down through

```
tkm_GetTokenNoVolCheck()
    tkm_TokenList_NoConflicts()
        tkm_Token_CheckRevokeReqd()
```

In tkm_Token_CheckRevokeReqd, the comparisons of the "hostPossessingToken" fields are considered equal if the new flag contains the TKM_NO_CONFLICT_CHECK bit.

b. The TKM [redacted]MAY[redacted] need to provide a new operation which subdivides a token into up to three subranges. This will only be necessary if the trick mentioned earlier (re-requesting the existing lock as up to three separate locks) is unworkable. This would be similar in concept to the swap-token which is performed when attempting a lockf, but different in that it can't fail due to a conflict (since the subdivision of the lock is requested by the lock token holder). The TKM would presumably then call the host module to tell it that the token had been subdivided. This could probably use the swap_token interface, with the minor extension that there are up to three subdivisions given to a node, instead of just two.

```
tkm_Subdivide(in_token,
    range1, range2, range3,
    out_token1, out_token2, out_token3);
```

2.1.6.9 General Assumptions

It is assumed throughout that the underlying kernel will be implemented using VFS's, and will use the file-ops structure (which we will be able to extend).

It is also assumed that each type of file and VFS will be required to support the operations necessary to allow migration of open files. If this is not true for a given type of file or VFS, processes with such files open will not be able to migrate.

This also implies that AFS is able to export any UFS file system. This means that either all UFS's are "glued" at mount time, or the AFS code is able to modify the existing vnodes in core to "glue" them.

2.1.6.10 Error Handling

Errors occurring during any of the new file, VFS, and vnode operations return errors in the normal way.

If any errors occur during a process movement operation, the operation fails. Any files which had been successfully reopened on the destination node will eventually be closed when the new process cleans up and exits.

2.1.6.11 Security

The export/reopen protocol described here could be a tempting security hole: an open file's reopen handle is sent to another node, potentially in cleartext. Whether or not to

encrypt or otherwise protect the reopen handle in transit depends on several factors.

If the handle itself is or contains a "secret" in the security structure of the underlying remote file system, and would normally be encrypted for that reason, then clearly transmission in the clear is dangerous. A scheme like this might be used since it allows Unix open/chmod/read semantics to work properly.

If access checking is done by authenticating the requesting user and then checking the requesting user's rights to access the file, then sending the handle in the clear doesn't compromise the file. However, if the handle is transmitted in the clear, then the handle is subject to modification by gateways.

To avoid these problem, the various reopen handles should be sent encrypted when the possibility of them being modified in transit exists. Whether or not to encrypt them should be at least an installation option, but individual file systems can encrypt their handles just by having their prep-export op do so, and their reopen op decrypt.

In addition, the special handles generated for devices and fifo's by the export ops of **FUSION** fifo, devices, and sockets must either be encrypted or have some other security policy. It might even be necessary to double-encrypt such handles: in a server's secret key to prevent forging of a handle, and in a session key, to prevent interception of the server's secret handle. Interception of the server's secret key would allow an interloper to access the device.

2.1.6.12 Performance

2.1.6.12.1 Expected

2.1.6.12.2 Potential Optimizations

The export loop for a process's file state could realize that a given file has already been exported, and avoid actually doing the second export for the current pt op. (i.e., only do stdin, not stdout & stderr... the latter two being handled by their fileblock tokens).

2.1.6.12.2.1 Parallel and Batched RPC

Reopen need to do a number of operations which need not be serialized. This means that reopen could take significant advantage of parallelizing and batching if they were made available from the RPC implementation.

To allow batching some RPC's should be delayed until a synchronize operation is invoked. The RPC's would then be sorted by destination, and sent in batches. That would be straight forward:

- Add a new attribute "asynchronous" to RPC language of your choice. All such rpc's must "return void" (enforce by RPC compiler). Code must be careful not to use "OUT" parameters until the sync op has returned (not enforced).
- For each such rpc, three pieces of client code are generated instead of one:
  The first just stashes its arguments in a queue.

The second client side routine knows how to interpret the stashed arguments, marshal them, and put them into an area provided by the rpc batching code.
The third stub would have code to unmarshal the response data out of the area provided by the rpc batch handling code. (note that most of this code is already generated by the compiler).

- Code called from the sync op would just scan, sort, and perform (in parallel) chunks of RPC's via a single round-trip to each node involved.

2.1.6.13 Packaging

The file-reopen and lock inheritance code is relatively non-invasive, requiring only that a few new vnode ops be added to the base and to file systems the base provides. In general, the upper level reopen modules can be dropped into an OS as an extension, without otherwise disturbing it.

NEEDSWORK: We need to examine how the reopen stuff interacts with the CMS. In particular, how does the reopen code know that the CMS isn't present?

2.2 Remote Processing Support

2.2.1 Vprocs

The **FUSION** vproc interface is a protocol used to implement remote process management with minimal intrusion in the AIX V3.1 Unix operating system. The vproc architecture which this design is based upon is described in the Locus Computing Corportation vproc architecture paper entitled "Vprocs: An Architecture Supporting Process Transparency".

The design issues with regards to implementing vprocs within a remote process environment are divided into sections.

1. The vproc architecture has already been defined in the functional specifications document. There are certain issues which pertain to the vproc architecture which are implementation defined. This first section describes these areas.

2. The private vproc data, defined within the vproc structure as `vp_pvproc` is an implementation defined area which needs much discussion as to its design. This section goes into gory detail about the private data area of the vproc for this implmentation.

3. Once the implementation issues for the vproc design have been clarified, the modifications to the AIX V3.1 code can be discussed. Even if remote processing is not supported in the vproc implementation described above, the modifications to the base operating system kernel still must be done to provide support for the vproc layer.

4. The sections above, when implemented, allow a single node system to run with the vproc architecture. Remote support must still be added to complete the design of vprocs for the **FUSION** implementation. Support for remote processing, including the design of the message passing system and any additional support is given in this section.

2.2.1.1 Base Vproc Interface

The base vproc interface is concerned with any implementation dependencies of the vproc object. The `vp_vops` field is described in detail. The table of operations is given, along with the function declarations; The rules for the existence of a vproc are given which are specific to this implementation. The reference count field `vp_ref_count` is used to keep track of how many references currently exist for a vproc. `vp_loc`, the field defined to be used to find vproc objects on a system has its structure defined. In addition, the semantics of how to locate a vproc associated with a process is also discussed, along with the operations used to implement this functionality.

2.2.1.1.1 Vproc Operations

The table below defines the set of vproc operations which are used in this implementation. Next to each vproc operation is the calling convention used to invoke

the routine. All but the last three operations are the operations as specified in the vproc functional specifications document.

vops_fork_relationships	**VOPS_FORK_RELATIONSHIPS()**
vops_exit_relationships	**VOPS_EXIT_RELATIONSHIPS()**
vops_wait	**VOPS_WAIT()**
vops_proc_nice	**VOPS_PROC_NICE()**
vops_pgrp_nice	**VOPS_PGRP_NICE()**
vops_sigproc	**VOPS_SIGPROC()**
vops_sigpgrp	**VOPS_SIGPGRP()**
vops_set_stop_state	**VOPS_SET_STOP_STATE()**
vops_setpgid	**VOPS_SETPGID()**
vops_setsid	**VOPS_SETSID()**
vops_get_pgrp_sid	**VOPS_GET_PGRP_SID()**
vops_setpinit	**VOPS_SETPINIT()**
vops_setpri	**VOPS_SETPRI()**
vops_getpri	**VOPS_GETPRI()**

pvproc_setpinit, *pproc_setpri*, and *pproc_getpri* are three extended vproc operations which are specific to the AIX V3.1 operating system. These operations are not required to be supported on other implementations of vprocs.

The vp_ops is an array of indirect function pointers used to implement the standard set of vproc operations to implement POSIX semantics for process management.

All vproc operations return an integer value. A zero return value signifies the operation was successful; a nonzero value returned is interpreted to mean an error occurred during the operation and the value signifies a UNIX errno which the calling operation can subsequently store in `u.u_error` of the calling process.

2.2.1.1.1.1 vops_fork_relationships — Initialize a new vproc for a process

```
int (*vops_fork_relationships)(pp, v, pid, procp)
IN struct vproc *pp;
INOUT struct vproc **v;
IN pid_t pid;
IN struct proc *procp;
```

Parameter

pp — Pointer to parent's vproc which is used to initialize new process.
v — Pointer to child's vproc which is being created.
pid — Process ID for the new process. This is a unique value.
procp — Pointer to physical process structure allocated for child process.

Description

A new vproc is allocated and initialized for the process being created, specified by the pid parameter. The physical process data object procp, which has already been created, is passed in as a parameter such that it can be attached to the vproc through the private vproc data area. Information used to create the new process is taken from the pp parameter, which will be the parent of the new process. The vproc created for the new process is returned using the v parameter.

In addition to the vproc, the private vproc data area must also be created and any process relationships such as parent-child-sibling lists, process group lists, and uid relationships must be addressed.

The semantics of this operation are such that this should be performed only after all physical process creation of the new process has been successfully performed.

2.2.1.1.1.2 vops_exit_relationships — terminate process relationships

```
int (*vops_exit_relationships)(v)
IN struct vproc *v;
```

Parameter

v — vproc associated with terminating process.

Description

Perform the various process relationship operations with respect to when a process terminates. The process which is terminating is specified by the v parameter.

- Any children of the terminating process must be reassigned to the INIT process
- the terminating process sends a SIGCHLD signal to its parent to notify it of an impending terminating child
- if the parent of the terminating child is ignoring the SIGCHLD signal, the terminating process must then be removed from its parents' parent-child-sibling list along with resigning from its current process group and session.
- In relation to the process group, the terminating process must also determine the effects of creating orphaned process groups of the process group which it is exiting along with the process groups of all of its children.

2.2.1.1.1.3 vops_wait — wait for process termination

```
int (*vops_wait)(v, pid, options, wstat, ru_loc, found_child,
                    wait_satisfied, ret_val)
IN struct vproc *v;
```

```
IN pid_t pid;
IN int options;
INOUT char *wstat;
INOUT struct rusage *ru_loc;
INOUT int found_child;
INOUT int wait_satisfied;
INOUT int ret_val;
```

Parameter

v -
pid -
options -
wstat -
ru_loc -
found_child -
wait_satisfied -
ret_val -

Description

Allow the calling process specified by the v parameter to obtain status information pertaining to one of its child processes. The operation is used to search through the process relationship information which is located in the private vproc data. Various options permit status information to be obtained for child processes that have stopped or terminated. The pid and options parameters specify the set of child processes which status is requested. The wstat parameter is used to hold the child exit status; ru_loc contains the resource usage of the exited child.

found_child, wait_satisfied, and ret_val are used to keep track of the whether a successful wait operation has occurred, and to return the process ID of the child.

This operation only determines which terminated or stopped child process will status information be retrieved. Any physical wait operations, such as getting the resource statistics or deallocating any resources still held by the stopped or terminated child, must still be performed on the node where the physical process currently exists.

2.2.1.1.1.4 vops_proc_nice — Read/write process nice value

```
int (*vops_proc_nice)(v, nice, flag)
IN struct vproc *v;
INOUT nice_t *nice;
IN int flag;
```

Parameter

v — vproc associated with process which operation is performed.
nice — nice value to be set, or returned nice value.
flag — determines whether nice value is read or written.

Description

Retrieve or set the nice value for a process specified by the v parameter. The flag parameter is set to **VPROC_SET** to set the nice value; **VPROC_GET** is used to get the nice value of the process. The nice parameter will contain the nice value to be set, or the returned nice value, depending upon the operation being performed.

2.2.1.1.1.5 vops_pgrp_nice — Nice operation on process group

```
int (*vops_pgrp_nice)(g, nice, flag)
IN struct vproc *g;
INOUT nice_t *nice;
IN int flag;
```

Parameter

g — vproc associated with process group leader.
nice — nice value.
flag — determines whether nice value read or written.

Description

Retrieve the lowest nice value of all members of a process group. The process group leader is specified by the g parameter. The flag parameter is set to **VPROC_SET** to set the nice value; **VPROC_GET** is used to get the nice value of the process group. The nice parameter will contain the nice value to be set, or the returned nice value, depending upon the operation being performed.

2.2.1.1.1.6 vops_sigproc — Send a signal to a process

Signal operations may require appropriate permissions before the signal is sent. For both operations defined below the signal privilege semantics are now defined.

Privilege is allowed if the process sending the signal has its real or effective uid match the real or effective uid of the process receiving the signal, or if the sending process has appropriate privileges. Appropriate privileges can be interpreted to mean superuser privilege. In the case of SIGCONT, the user ID tests are not applied if both processes are in the same session.

```
int (*vops_sigproc)(v, signo, effuid, realuid, sid, has_priv, flag)
IN struct vproc *v;
IN int signo;
IN uid_t effuid;
```

```
IN uid_t realuid;
IN pid_t sid;
INOUT int *has_priv;
IN long flag;
```

Parameter

v — vproc associated with process being sent the signal.
signo — signal being sent.
effuid — effective uid of calling process.
realuid — real uid of calling process.
sid — session identifier of calling process.
has_priv — returns whether or not calling process can send signal.
flag — determines if checking just for privilege or sending signal.

Description

Send a signal as specified by the signo parameter to the process specified by the v parameter. The parameters effuid, realuid, and sid are used to verify the process sending the signal has correct privileges. If has_priv is nonzero, the process sending the signal has superuser privilege; otherwise the remaining tests must still be done. has_priv will return a nonzero value if the process has correct privilege to send the specified signal and flag is set to **VPROC_PRIV**.

2.2.1.1.1.7 vops_sigpgrp — Send a signal to a process group

```
int (*vops_sigpgrp)(g, signo, effuid, realuid, sid, has_priv)
IN struct vproc *g;
IN int signo;
IN uid_t effuid;
IN uid_t realuid;
IN pid_t sid;
IN int has_priv;
```

Parameter

g — vproc associated with process group leader.
signo — signal being sent.
effuid — effective uid of calling process.
realuid — real uid of calling process.
sid — session identifier of calling process.
has_priv — determines if calling process is superuser.

Description

Send a signal specified by the signo parameter to the process group specified by the g parameter. The parameters effuid, realuid, and sid are used to verify the process sending the signal has correct privileges. If has_priv is nonzero, the process sending the signal has superuser privilege.

2.2.1.1.1.8 vops_set_stop_state — Set/clear process stopped state

```
int (*vops_set_stop_state)(v, flag)
IN struct vproc *v;
IN int flag;
```

Parameter

v — vproc of physical process.
flag — determines action of operation.

Description

Set or clear the stopped status of a process specified by the v parameter. The flag parameter is specified as **SET_STOP_STATE** if the process is to be in stopped state; **UNSET_STOP_STATE** takes a process out of stopped state.

2.2.1.1.1.9 vops_setpgid — set process group ID

```
int (*vops_setpgid)(v, g, pid, sid)
IN struct vproc *v;
IN struct vproc *g;
IN pid_t pid;
IN pid_t sid;
```

Parameter

v — vproc associated with process having process group set.
g — vproc associated with process group leader.
pid -
sid -

Description

The process specified by the v parameter either joins an existing process group or creates a new process group within the session of the calling process. The group leader of the process group which v is joining is specified by the g parameter. If a new process group is being created, v and g are the same. The parameters pid and sid are used for error checking when adding the process to its new group.

The process must first resign from its current process group and session before being added to its new process group. If this is a new group, the group is added to the appropriate session list.

The effects of changing process groups in relation to creating orphaned process groups must also be determined. The affected process groups that must be checked are the group which the process is leaving, the process groups of all children of this process, and the new process group of this process. The effects of changing process groups will create new orphaned process groups or unorphan process groups.

2.2.1.1.1.10 vops_setsid — Create a new session

```
int (*vops_setsid)(v)
IN struct vproc *v;
```

Parameter

v — vproc associated with process becoming new session.

Description

The process specified by the v parameter must first resign from its current process group and session; a new session is thus created, with this process being the session leader. A new process group is also created with this process being the process group leader.

A determination must be made as to whether leaving the old process group will create an orphaned process group. In addition, the process group of each child of this process must also be checked for becoming an orphaned process group. The new process group as created by the session operation is by definition an orphaned process group.

2.2.1.1.1.11 vops_get_pgrp_sid — Get process group and session identifiers

```
int (*vops_get_pgrp_sid)(v, pgrp, sid)
IN struct vproc *v;
INOUT pid_t *pgrp;
INOUT pid_t *sid;
```

Parameter

v — vproc associated with process.
pgrp — return value of process group identifier.
sid — return value of session identifier.

Description

Retrieve the process group and session ID's of a process. The v parameter specifies the process to be queried. The pgrp and sid parameters are used to return the values retrieved from the process.

2.2.1.1.1.12 vops_setpinit — Create a kernel process

```
int (*vops_setpinit)(v)
IN struct vproc *v;
```

Parameter

v — vproc associated with kernel process.

Description

The specified process is to become a kernel process. It is removed from its current parent and reattached to the INIT process. This is an extended operation specific to the AIX V3.1 implementation of the vproc layer.

2.2.1.1.1.13 vops_setpri — set the priority of a process

```
int (*vops_setpri)(v, pri)
IN struct vproc *v;
INOUT int *pri;
```

Parameter

v — vproc associated with process having priority set.
pri — priority value to set, returns previous priority.

Description

Set the priority of the process specified by the vproc parameter. The physical process structure is accessed through the vproc object and the priority is set if no error conditions are met. The previous priority of the process is returned in pri, -1 if an error occurred. This is an extended operation specific to the AIX V3.1 implementation of the vproc layer.

2.2.1.1.1.14 vops_getpri — get the priority of a process

```
int (*vops_getpri)(v, uid, ruid, priv, pri)
IN struct vproc *v;
IN uid_t uid;
IN uid_t ruid;
IN int priv;
OUT int *pri;
```

Parameter

v — vproc associated with process specified.
uid — effective uid of calling process.
ruid — real uid of calling process.
priv — privilege level of calling process.
pri — return value for priority of process.

Description

The priority of a specified process is returned. v is used to reference the physical process entry. If the calling process has correct permissions as specified by the uid, ruid, and priv parameters, the priority is returned within the pri parameter. This is an extended operation specific to the AIX V3.1 implementation of the vproc layer.

2.2.1.1.2 Where Vprocs Exist

There are a prescribed set of rules which govern when a vproc will exist or not exist on a given node. The conditions as to when each rule is met is given below. In the context of this section, "marking a vproc" implies incrementing the vp_ref_cnt of the vproc. "Unmarking a vproc" implies decrementing the vp_ref_cnt of the vproc. When decrementing, if the reference count goes to zero, the vproc is deallocated from the system. This also implies the private vproc data area is also deallocated with the vproc.

Marking and unmarking a vproc reference count is performed using the operations **VPROC_HOLD(v, str)** and **VPROC_RELEASE(v, str)**. The v parameter specifies the vproc which this operation takes place. The str parameter is a string value identifying the reason for changing the reference count. For example, given the rule defining a child process, the vproc reference count would be incremented with the operation **VPROC_HOLD(v, "CHILD")**. Of course the string has its value as a debugging instrument and to make the code more readable. Under normal operation the string value would not be used.

Active Mark the vproc because it is a process that is currently executing on this node. The vproc is unmarked when it no longer is executing on this node (e.g. process migration, process termination).

Original Node Mark the vproc because it is a process that was initially created on this node (the "original execution node") and either (a) the process is still active or (b) the process ID is still the process group leader or session leader for a set of processes.

Condition (b) satisfies the rule that the origin node always knows the execution node of the vproc. If for example a process group leader migrates to another node, the possibility exists that the origin node would not have a vproc for the process group leader anymore.

Processes in the process group requesting the location for the process group leader (more specifically, the process group list) would not be able to find it. Marking the vproc (or not unmarking the vproc) by the caller due to condition (b) will always keep the vproc for the process group leader and session leader on the origin node.

The vproc will be unmarked if the process (which is the group leader) is exiting and there are no other processes within this process group. In this situation there is no further need to keep the vproc for the group leader on the origin node since there are no other vprocs within this process group.

Parent Mark the vproc because it is the parent of a process currently executing on this node. The child vproc marks its parent vproc on the execution node of the child process.

The primary use of this rule is to instantiate a parent vproc on the execution node of the child process. According to section 2.2.1.2.1.3 a parent vproc must exist on the execution node of a child vproc while it still is active.

Child Mark the vproc because it is the child of a process currently executing on this node. The parent vproc marks a child vproc on the execution node of the parent process.

The primary use of this rule is to instantiate a vproc for a child process on the execution node of the parent, since the execution node of the parent vproc contains a complete list of child vprocs that are currently active.

Process Group Leader Mark the vproc because it is the process group leader of a process currently executing on this node. The process group leader vproc gets marked by a process group member on the execution node of the process group member.

Note that the operations are also used by process group leaders, and thus are applied to the process itself if the process group leader is a member of its own process group.

Process Group Member Mark the vproc because it is a member of a process group for which the process group leader is active on this node.

This rule is used to instantiate a vproc on the execution node of the process group leader to be included in the list of processes in the process group. Note that the operations are also used by process group leaders, and thus are applied to the process itself if the process group leader is a member of its own process group. See section 2.2.1.2.1.4 for more information on process group leader, process group member

lists.

Session Leader Mark the vproc because it is the session leader of a process currently active on this node. A session member executing on the node of the session leader marks the session leader.

Note that the operations are also used by the session leaders, and thus are applied to the process itself, since a session leader is always a member of its own session.

Session Member Mark the vproc because it is a process group leader and it is a member of a session for which the session leader is executing on this node.

This rule is used to instantiate a vproc for a process group leader on the execution node of the session leader to be included in the list of sessions in the session list. Note that the operations are also used by the session leaders, and thus are applied to the process itself, since a session leader is always a member of its own session. See section 2.2.1.2.1.5 for more information on session lists.

Generic Mark the vproc because it is for a process being "temporarily" referred to. A "temporary" referral is one where a process ID is referred to (e.g. in a system call) and the process being referenced may not have a vproc on the current node (i.e. none of the above conditions may be true). In this case a vproc may be created for the duration of the reference (i.e. for the duration of the system call).

There are other instances where a vproc is marked temporary. The most widely used example is when a vproc is locked and unlocked. A lock operation implies a temporary marking on the vproc (even if the lock isn't immediately satisfied and the process attempting the lock becomes blocked), and an unlock operation removes the temporary marking. This design removes the race condition where a process requests a lock for a vproc, becomes blocked waiting for the lock, then the vproc gets deallocated due to an operation performed by the process which currently has the vproc locked.

2.2.1.1.3 Finding Vprocs

vp_loc is defined to be a structure containing two pointers which implement a hash chain for vprocs. More information is given in the section describing how vprocs are located. The structure is accurately defined as follows:

```
typedef struct {
    struct vproc *vp_hashfwd;
    struct vproc *vp_hashbwd;
} location_t;
```

In relation to the various rules that govern the existence of a vproc on specific node, three different methods are defined to access a specific vproc.

When the vproc is known to exist (*e.g. the currently executing process needs to access its vproc*), a call to **VPROCPTR(pid)** will return a pointer to the vproc associated with the pid argument, or NULL if the vproc cannot be found.

If access to a vproc on the current node is required, but there is no certainty that the vproc exists on the current node, a call to **LOCATE_VPROC_PID(pid)** will return a pointer to the vproc associated with the pid argument. If the vproc exists on the system, that vproc is returned. If the vproc is not found, a new vproc is created with information specified by the calling process. The vproc does get initialized just as other vprocs on this node are initialized, with the vproc being put on the hash chains as long as it is in existence on this node. In either case, the vproc is marked as a temporary vproc according to the rules specified. If LOCATE_VPROC_PID(pid) is used and a vproc is returned, the vproc must be unmarked after its use with a call to **RELEASE_VPROC(vproc)** and possibly free the vproc if its use count goes to zero.

If the vproc is known not to exist on the current node, **VPROCLOC(pid, orig_node, exec_node_hint)** should be used. A new vproc is allocated according to values specified and this new vproc is returned.

2.2.1.1.3.1 vprocloc()

```
struct vproc * vprocloc(pid, orig_node, exec_node_hint, flag)
IN pid_t pid;
IN node_t orig_node;
IN node_t exec_node_hint;
IN int flag;
```

Parameter

pid — process ID of vproc to find or create.
orig_node — original execution node of vproc.
exec_node_hint — guess as to where process currently executes.
flag — flag argument.

Description

This routine is used to implement VPROCLOC() and LOCATE_VPROC_PID() defined in the previous section. vprocloc() will attempt to find a vproc on the current node; if it cannot be found, a new vproc is allocated and initialized with information from the calling process. If an existing vproc is found, a pointer to the vproc is returned without modifying any of its information.

The vproc and pvproc operations tables are initialized by analyzing the pid argument to determine if the process is executing on a local or remote node (see the section describing pid allocation for more information). If the pid describes a local process the vproc operations tables will point at the set of local operations. If the pid describes a remote process a determination needs to be made as to which remote operations tables the vproc should refer.

The implementation of vprocs will support multiple client/server protocols and as such a description of which protocol to use to talk to specific nodes is needed. A table is generated containing entries describing a node and pointers to the associated remote operations to be used to communicate with that node. Information within the table should be updated as soon as a node establishes communication with the local node.

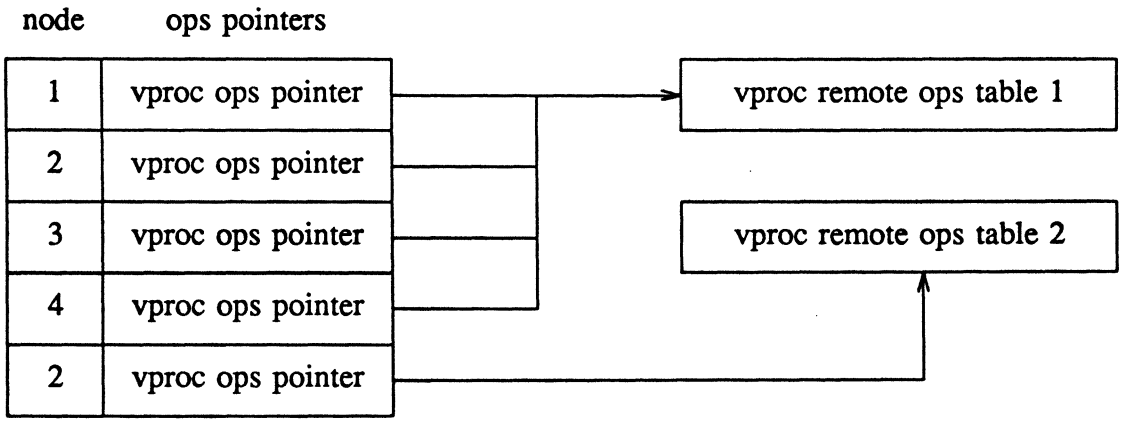


Figure 6. Mapping Remote Nodes with Supported Protocols

The set of remote operations can be determined by a lookup into the table using the exec_node_hint value. If the value of the orig_node and exec_node_hint arguments is NO_NODE, extrapolate the node value from the pid for all references to a node. Of course a single node may have multiple protocols which it supports and as such multiple entries for that node will be in this table, as shown in Figure 6. The first entry for the node found in the table is used to specify the set of remote operations.

The flag argument specified is provided for remote processing support, specifically if a process is migrated to another node. In this instance a proc entry is allocated and attached to the vproc that was either found or has been allocated.

2.2.1.1.3.2 vprocptr()

```
struct vproc * vprocptr(pid)
IN pid_t pid;
```

Parameter

pid — process ID of vproc to locate.

Description

Locate a given vproc that should already be in existence on this node using the pid of the process. This function is used by the VPROCPTR() operation when the operation cannot quickly find the vproc.

2.2.1.1.3.3 pidgen()

```
pid_t pidgen()
```

Parameter

n/a

Description

Generation of a unique process ID for a process is performed by using this new routine. When creating a new process ID, the vproc hashchains are checked to verify the uniqueness of the identifier. The new process identifier is then returned to the calling process.

2.2.1.2 Private Vproc Data Interface

Each vproc contains an implementation specific pvproc data structure. The pvproc data structure is not required to conform to any predefined structure. Its use is to contain information required only by the current vproc implementation. Access to this information is through the vproc ops routines.

Additional support for an implementation of vprocs must be done using the private vproc data area.

2.2.1.2.1 Private Vproc Data Object

The specification for the vp_pvproc data structure designed for FUSION is given. The sections below describe in detail each field within this data structure.

```
struct pvproc {
        long              vp_flag;
        struct proc      *vp_pproc;
        struct vproc     *vp_child;
        struct vproc     *vp_siblings;
        struct vproc     *vp_ganchor;
        struct vproc     *vp_pgrpl;
        struct vproc     *vp_ttyl;
        struct pvproc_ops *vp_ops;
        node_t            vp_orig_node;
```

```
        node_t          vp_exec_node_hint;
        pid_t           vp_ppid;
        pid_t           vp_sid;
        pid_t           vp_pgrp;
        locks_t         vp_lock;
        long            vp_orphan_cnt;
        char            vp_error;
        char            vp_pad;         /* padding */
        caddr_t         vp_pageno;
        /* padding to align pvproc structure accordingly */
};
```

2.2.1.2.1.1 Pvproc Attributes (vp_flag)

This field specifies any implementation specific flags. The values that are currently supported are defined below:

PV_NOSETSID process currently becoming a process group leader. This flag is needed since a remote process can specify another process to become group leader.

PV_PGRPLEADER process is a process group leader and VONSITE is set.

PV_SESSIONLEADER process is session leader and VONSITE is set.

PV_ORPHAN_PGRP_CONTRIBUTOR specifies if a process contributes to its process group not becoming orphaned. Used by this implementation to check for orphan process groups.

PV_SZOMB Process is in zombie state. Set in child vproc on execution node of parent when child process exits.

PV_SSTOP Process is in stopped state. Set in child vproc on execution node of parent when child process becomes stopped.

PV_MIG_INCOMPLETE migrate operation is in the process of moving data pages to execution node of process.

2.2.1.2.1.2 Process Dependent Data Pointer (vp_pproc)

vp_pproc is used to point at the physical process entry (**struct proc ***) of a process on the execution node of a process. On those nodes for which the process does not execute, vp_pproc will be NULL. This proc structure is the same as a UNIX proc structure except for the possible removal of fields which will be supported in the vproc structure:

- the parent-child-sibling chain fields
- the process group membership chain fields
- the session membership chain fields
- any hashing fields used to locate the process given a process ID.

The fields in the proc structure are replaced with dummy fields so that kernel binary compatibility with existing kernel add-on modules such as device drivers is maintained. In addition, the proc structure fields p_pid, p_ppid, p_pgrp, and p_sid are maintained and updated in both the proc structure and vproc structure. This is also to maintain binary compatibility with existing add-on kernel modules such as device drivers which may use these fields.

2.2.1.2.1.3 Parent-Child-Sibling Relationship (vp_child, vp_siblings, vp_ppid)

There exists a parent-child-sibling chain of vprocs which defines all the children for a parent vproc. The parent-child-sibling chain is maintained only on the execution node of the parent process. This implies a vproc for every child is kept on the execution node of their parent. This list is analogous to the parent-child-sibling list for processes that exist in standard UNIX kernels. The list is now implemented using vprocs. A parent vproc will exist on every node for which a child process is executing, even though the parent-child-sibling list is not kept on those nodes. This is for when a child process needs access to its parent such as when it exits the system. The list of child vprocs is kept on the execution node of the parent such that when the parent exits, any children still executing can be reassigned quickly to the init process on the execution node of each child.

A parent vproc will have its vp_child point to the first of its children; this child will then have its vp_siblings point to the next sibling (child of the parent).

The vp_ppid is a field that identifies the parent process ID. This field is kept updated only on the current execution node of the child process. otherwise).

Figure 7 illustrates the parent-child-sibling relationship of vprocs where the parent (vproc1) and one child (vproc2) are currently executing on node one, while two other children (vproc3 and vproc4) are executing on node two. There is a parent vproc on both nodes since there are children which execute on nodes one and two; the parent-child-sibling list exists only on node one, along with a vproc for every child of this parent.

2.2.1.2.1.4 Group Leader — Group Member Relationship (vp_ganchor,vp_pgrpl,vp_pgrp)

There exists a process group leader-process group member chain of vprocs (vp_ganchor, vp_pgrpl) to keep track of all processes in a process group. The execution node of the process group leader maintains a list of all vprocs which are process group members within this process group. This implies processes executing on remote nodes will also have a vproc on the execution node of the process group leader. There is also a vproc for the process group leader on all nodes which have a process in the process group executing, though a process group list not kept on these remote nodes.

The process group leader uses its vp_ganchor field to point at the first vproc within the process group. The remaining vprocs within the process group use their vp_pgrpl fields to point at all subsequent vprocs. The two fields are used since a process group

Figure 7. Parent-Child-Sibling Relationships

leader can exist within another process group. If the group leader is in its own group, the group leader vproc points at itself first using the vp_ganchor field, then points at subsequent vprocs within the process group using its vp_pgrpl field.

If the process group leader moves (migrates or rexec's) to another node, the entire list is rebuilt on the remote node.

Each vproc also has a process group ID field (vp_pgrp). This field is kept updated only on the current execution node for the process.

Figure 8 illustrates the process group chains of vprocs where the process group leader (vproc1) and one process group member (vproc3) are executing on node one while three other process group members (vproc2, vproc4, and vproc5) are executing on node two. Node 1 contains a vproc for the process group leader, a vproc for the process (vproc3) executing on this node, and vprocs for the three processes (vproc2, vproc4, vproc5) which are executing on node 2. On node 2, their is a vproc for the process group leader (vproc1), and vprocs for the three processes that are executing on this node. The execution node of the process group leader contains a complete list of processes within the process group (on all nodes).

2.2.1.2.1.5 Session Relationships (vp_ttyl, vp_sid)

The session list on the execution node of the session leader keeps an entire list of process group leaders in the session. In addition, on all nodes where a session member is executing, there will be a session leader vproc. This also implies that a process group leader vproc is also on this node. Since a session leader cannot reside in another session, the vp_ttyl field is sufficient to implement the session list.

If the session moves (migrates or rexec's) to another node, the entire list is rebuilt on the new node.

Each vproc also has a session ID field (vp_sid). This field is kept updated only on the current execution node for the process.

Figure 9 illustrates the session leader chains that will exist. Node 1 contains a session leader (vproc1), which according to POSIX definition makes it a process group leader. Also on node 1 is another process group leader (vproc2) which is in this session. A separate process group leader (vproc3) within this session is executing on node 2.

The session leader (vproc1) exists on both nodes since there are session members (e.g. process group leaders within this session) which are executing on both nodes (vproc1, vproc2 on node 1; vproc3 on node 2). Vproc3 also exists on node 1 since it belongs in a session (vproc1) which is executing on node 1. The complete list of process group leaders belonging to this session is kept on node 1, which is the execution node of the session leader.

2.2.1.2.1.6 Pvproc Operations (vp_vops)

Implementation specific operations performed on the pvproc structure that is associated with a vproc use the set of operations defined in the pvproc operations table.

Figure 8. Process Group chains

vp_vops is an array of indirect function pointers used to access the operations. The naming convention used to access these operations is defined to be a set of macros with the prefix **PVPOP_**<pvops name>. The complete list of operations defined in this table is list below. A complete description is given in a subsequent section.

pvops_wait3	**PVOPS_WAIT3()**
pvops_reassign_child	**PVOPS_REASSIGN_CHILD()**
pvops_rmv_child_from_parent	**PVOPS_RMV_CHILD_FROM_PARENT()**
pvops_rmv_child_if_no_sigchld	**PVOPS_RMV_CHILD_IF_NO_SIGCHLD()**
pvops_proc_flag	**PVOPS_PROC_FLAG()**
pvops_proc_status	**PVOPS_PROC_STATUS()**
pvops_pvproc_flag	**PVOPS_PVPROC_FLAG()**
pvops_resign_pgrp	**PVOPS_RESIGN_PGRP()**
pvops_add_pgrp_list	**PVOPS_ADD_PGRP_LIST()**
pvops_rmv_pgrp_list	**PVOPS_RMV_PGRP_LIST()**
pvops_add_session_list	**PVOPS_ADD_SESSION_LIST()**
pvops_rmv_session_list	**PVOPS_RMV_SESSION_LIST()**
pvops_orphan_child_pgrp	**PVOPS_ORPHAN_CHILD_PGRP()**
pvops_adjust_orphan_count	**PVOPS_ADJUST_ORPHAN_COUNT()**

2.2.1.2.1.7 Original and Execution Node (vp_orig_node, vp_exec_node_hint)

These two fields are used when a remote operation to perform some instance of vproc manipulation is necessary. vp_exec_node_hint contains a current "guess" as to where the vproc is executing. A remote operation will first try this node. The remote node can then return "OK", meaning the vproc is executing here; A return value of "I've never heard of this vproc on this node" implies the remote operation should go directly to the origin node since the origin node will always know where a process is executing (which originated from this node); A return value of "execution node isn't here, but try <node>" is a guess that is made by the remote node. This new execution node is remembered for future operations and the request is subsequently made to this new node.

The remote node also has the option of returning a message to the sender specifying the request cannot be handled at this time and the sender should resend the request in <t> seconds. If the sender will not wait to resend the request, the message can be specified as **urgent** and must be accepted at the destination node.

There is the possibility in the third scenario that the new guess will also be incorrect, and the remote node will then send another "guess", which could lead to a loop. To resolve this situation, a maximum number of tries is allowed before the remote request goes directly to the origin node. There is a small chance that the origin node will not be able to correctly identify where the vproc is executing. In this instance, a maximum number of tries should be made with the "guess" received from the origin node before returning an error that the vproc cannot be located.

Figure 9. Session Leader chains

These fields are required for Process Transparency to reliably perform remote process location. The origin node will always be kept updated in all pvprocs, however only a best guess in most instances can be expected for the vp_exec_node_hint on those nodes that do not have the executing process.

2.2.1.2.1.8 Vproc Lock Mechanism (vp_lock)

Operations requiring access or modifications to vproc structures requires the appropriate vprocs to be locked. Note that locking a vproc only locks a single vproc, not all instances of this vproc on all nodes. In the case of reading information, a shared locking mechanism (many readers, no writers) will suffice. In the case of modifying data, an exclusive locking mechanism (one writer) will be used. The lock object is kept in the pvproc data structure, but the actual lock pertains to the entire vproc, which is inclusive of the pvproc structure. This field is valid for all vprocs on all nodes.

The list below defines the types of locks supporting the vproc layer. The vproc fields which are covered by the specified type of lock are also listed.

GENERIC The generic lock covers all fields in the functionally defined vproc object and the flag field within the implementation defined pvproc object. The generic lock has a primary use to determine whether or not the vproc is executing on a local process, though modifying flag values is an important operation.

PGRP Operations on the process group leader and the associated process group list should use this set of operations to lock the process group leader vproc. This lock covers the pvproc fields vp_pgrp, vp_pgrpl, and vp_ganchor. If modifications are to be made, an exclusive lock should be used. For traversing a list or reading information, a shared lock should be used.

SESSION Operations on the session leader and the associated session list should use this set of operations to lock the session leader vproc. This type of lock covers the pvproc fields vp_sid and vp_ttyl. If modifications are to be made, an exclusive lock should be used. For traversing a list, the shared locks should be used.

PARENT Operations on the parent vproc and the associated parent-child-sibling list implemented with the pvproc fields vp_child and vp_sibling should use this type to lock the parent vproc. If modifications are to be made, an exclusive lock should be used. For traversing a list or reading information, a shared lock should be used.

MIGRATE The migrate type of lock is a union of the generic, pgrp, session, and parent locks. This type of lock is used only when a process is to be migrated from one node to another and a guarantee is needed that none of the lists which the process may be participating in are changed during this operation. The specified locking hierarchies are adhered to when gaining the locks for the migrate operation.

2.2.1.2.1.9 Process Group Orphan Count(vp_orphan_cnt)

This implementation supports the POSIX definition of orphaned process groups. An orphaned process group is a process group in which the parent of every member is either itself a member of the group or is not a member of the group's session.

This design uses a count to determine whether or not a process group has become an orphan process group. When the count is nonzero, there are processes within the group which contribute to the group not becoming orphaned. A process within a group is a contributor when it has a parent process not in the same process group, but within the same session.

2.2.1.2.1.10 Rfork Error Number (vp_error)

Errors during the execution of a remote fork operation (rfork()) are kept in the vp_error field of the pvproc structure. For more information regarding the values stored in this field, consult the section describing process migration and remote execution.

2.2.1.2.1.11 Data Page Requests (vp_pageno)

The section describing process migration and remote execution requires a field to store data page requests such that the page fault server can notify the stub process on the source node to send the requested page. The field vp_pageno is used during remote operations to store such page requests.

2.2.1.2.2 Vproc Private Data Operations

The operation defined below are used to implement process relationships within the vproc private data object. As with the vproc operations, each operation returns an error value; a zero is interpreted as a successful operation. A nonzero value is interpreted to mean an error occurred during the operation and the value itself is defined to be a well known UNIX errno which the calling process can store in `u.u_error`.

2.2.1.2.2.1 pvops_wait3 — get resource statistics from waited child

```
int (*pvops_wait3)(v, wstat, ru_loc, p_cpu, wait_satisfied, stat)
IN struct vproc *v;
OUT int *wstat;
OUT struct rusage *ru_loc;
OUT u_short *p_cpu;
OUT int *wait_satisfied;
OUT char *stat;
```

Parameter

v — vproc associated with process being waited upon.
wstat — return xp_stat field of terminated process.
ru_loc — return resource statistics of terminated process.

p_cpu — return machine architecture type of terminated process.
wait_satisfied — TRUE if child successfully waited upon.
stat — return status of physical process.

Description

Perform the wait operation on a physical process. This operation is called by a vproc interface operation once the parent has found a child which can be waited upon. If the process is in **SZOMB** state, resource statistics, machine architecture, and xp_stat statistics are returned to the calling process. If the process is in **SSTOP** state, only the physical process status is returned. In either case, if the child process was in either of these two states, the wait_satisfied field is set to **TRUE** to notify the calling process that the child was successfully waited upon.

2.2.1.2.2.2 pvops_reassign_child — reassign child to INIT

```
int (*pvops_reassign_child)(v)
IN struct vproc *v;
```

Parameter

v — vproc associated with process being reassigned.

Description

Reassign a child process to the INIT process at the execution node of the child process. Add the child process to INIT's parent-child-sibling list and apply the appropriate existence rules to the child process and INIT process. If the child process state is **SZOMB**, send SIGCHLD to INIT to clean it up.

2.2.1.2.2.3 pvops_rmv_child_from_parent — remove child process from current parent

```
int (*pvops_rmv_child_from_parent)(pp, v)
IN struct vproc *pp;
IN struct vproc *v;
```

Parameter

pp — vproc associated with parent process.
v — vproc associated with [child] process.

Description

Remove the child process from the parent-child-sibling list of its parent process. This list is kept by the parent on its execution node. Apply the appropriate existence rules to both parent and child to account for one less child.

2.2.1.2.2.4 pvops_rmv_child_if_no_sigchld — send SIGCHLD to parent

```
int (*pvops_rmv_child_if_no_sigchld)(pp, v, sigchld_ignored)
IN struct vproc *pp;
IN struct vproc *v;
OUT in *sigchld_ignored;
```

Parameter

pp — vproc associated with parent process.
v — vproc associated with terminated process.
sigchld_ignored — return **TRUE** if parent ignores SIGCHLD signal.

Description

Send the parent process a SIGCHLD signal to notify it of a terminating child process. If the parent ignores SIGCHLD, the child process must be removed from the parent-child-sibling list of its parent. The value of **TRUE** is returned in the sigchld_ignored parameter if the parent process is ignoring SIGCHLD signals.

2.2.1.2.2.5 pvops_proc_flag — Set/clear physical process flag

```
int (*vpop_proc_flag)(v, flag, set_get_clear)
IN struct vproc *v;
INOUT int *flag;
IN int set_get_clear;
```

Parameter

v — vproc associated with process.
flag — holds value to set or return value.
set_get_clear — value specifies semantics of operation.

Description

Access or modify the physical process flag value associated with the vproc argument specified. The flag value will either hold the flag value to set, or will return the flag value of the physical process. set_get_clear specifies either **VPROC_SET** to set the flag value, **VPROC_GET**, to retrieve the flag value, or **VPROC_CLEAR** to clear the flag value as specified by the flag argument.

2.2.1.2.2.6 pvops_proc_status — Set/get physical process stat value

```
int (*vpop_proc_status)(v, stat, flag)
IN struct vproc *v;
```

```
INOUT int *stat;
IN int flag;
```

Parameter

v — vproc associated with physical process.
stat — physical process stat value.
flag — determines semantics of operation.

Description

Set or get the p_stat value from the physical process associated with the v parameter. The stat parameter will hold the stat value to set, or the return value, depending upon whether flag is set to **VPROC_SET** or **VPROC_GET**.

2.2.1.2.2.7 pvops_pvproc_flag — Set/clear pvproc flag value

```
int (*vpop_pvproc_flag) (v, pid, flag, set_clear)
IN struct vproc *v;      /* determines execution node */
IN pid_t pid;            /* process ID of vproc to perform op */
IN int flag;             /* flag value */
IN int set_clear;        /* set or clear flag in pvproc */
```

Parameter

v — vproc which determines execution node of operation.
pid — process ID of vproc which to perform operation upon.
flag — flag value.
set_clerar — set or clear flag in pvproc.

Description

This operation is performed on the execution node of the v parameter specified, but the operation is performed on the vproc associated with the pid parameter. flag holds the value; set_clear will set the flag value if it is set to **VPROC_SET**, or will clear the value if it is set to **VPROC_CLEAR**.

2.2.1.2.2.8 pvops_resign_pgrp — resign process from current process group

```
int (*pvops_resign_pgrp) (v)
IN struct vproc *v;
```

Parameter

v — vproc associated with process resigning from process group.

Description

Remove the process specified by v from its process group list. Apply the appropriate existence rules for process group relationships to account for the loss of a process group member.

2.2.1.2.2.9 pvops_add_pgrp_list — add process to process group list

```
int (*pvops_add_pgrp_list) (g, v)
IN struct vproc *g;
IN struct vproc *v;
```

Parameter

g — vproc associated with process group leader.
v — vproc associated with process being added to process group.

Description

Add a process to the process group list of the specified process group. Mark the vproc being added to the list as it is now a member of the process group.

2.2.1.2.2.10 pvops_rmv_pgrp_list — remove process from process group list

```
int (*pvops_rmv_pgrp_list) (g, v)
IN struct vproc *g;
IN struct vproc *v;
```

Parameter

g — vproc associated with process group leader.
v — vproc associated with process being removed from process group.

Description

Remove a process from the process group list specified by the process group leader g. If this is the last process in this process group then resign the process group from the session list. Unmark the process being removed as it no longer is a member of this process group.

2.2.1.2.2.11 pvops_add_session_list — add process group leader to session list

```
int (*pvops_add_session_list) (s, g)
IN struct vproc *s;
IN struct vproc *g;
```

Parameter

s — vproc associated with session leader.
g — vproc associated with process group leader being added to session.

Description

Add a process group leader to the session list specified by the session leader s. Mark the process group leader as being a member of the session list. If the process group leader is in fact the session leader (i.e. we are creating a new session), initialize the session pointer vp_ttyl rather than adding this process group leader to the list. Mark the process group leader as being a member of the session list.

2.2.1.2.2.12 pvops_rmv_session_list — remove process group leader from session list

```
int (*pvops_rmv_session_list)(s, g)
IN struct vproc *s;
IN struct vproc *g;
```

Parameter

s — vproc associated with session leader.
g — vproc associated with process group leader being removed from session.

Description

Remove the process group leader from the session list. Unmark the process group leader as it no longer is a member of this session.

2.2.1.2.2.13 pvops_orphan_child_pgrp — check if process group becomes orphaned

```
int (*pvops_orphan_child_pgrp)(v, flag)
IN struct vproc *v;
IN long flag;
```

Parameter

v — vproc associated with child process being checked.
flag — determines the semantics of how this function is called.

Description

Determine if the effect from a parent process changing process groups, sessions, or terminating will cause the process group associated with the child process to become an orphaned process group.

2.2.1.2.2.14 pvops_adjust_orphan_count — adjust orphan count of process group

```
int (*pvops_adjust_orphan_count)(g, adjustment)
IN struct vproc *g;
IN long adjustment;
```

Parameter

g — vproc associated with process group leader.
adjustment — adjustment value, either 1 or −1.

Description

Adjust the orphan count kept by the process group leader. If the orphan count goes to zero from an adjustment of −1, the process group becomes orphaned. If the orphan count is incremented from zero due to an adjustment of 1, the process group is no longer considered an orphaned process group. Any other combinations do not affect the process group becoming orphaned in a direct manner.

2.2.1.2.3 Vproc Locks

One of the requirements for the vproc layer is to provide a consistent view of all processes within the system. This includes adding, deleting vprocs, scanning a list of vprocs, even modifying fields within a vproc.

For each vproc object that can be locked, there exists two types of locks; an exclusive lock exclusively locks a vproc. This guarantees the vproc is held only by the current process. An exclusive lock is used when a vproc operation will be modifying data within the vproc. In other terms, an exclusive lock is a write lock.

The other lock is a shared lock. A shared lock will allow many other shared locks on the same vproc, but no exclusive locks are allowed. This type of lock is equivalent to a read lock.

There is a locking hierarchy designed into the vproc layer that must be followed to guarantee data consistency. This hierarchy of gaining and releasing the locks must be followed to prevent any deadlock conditions from occurring.

2.2.1.2.3.1 Lock object

The lock object used to implement the locking strategy on AIX V3.1 is given below.

```
struct locks {
        pid_t   pid;
        int     excls;
        int     shares;
        short   sh_cnt;
        short   ex_cnt;
```

```
}

typedef struct locks   locks_t;
#define LOCK_AVAIL     ((lock_t) -1)
#define LOCK_UNK_OWNER ((pid_t)(LOCK_AVAIL-1))

#define <different types of vproc locking>
#define CONDITIONAL_LOCK
```

pid The pid contains the process which owns the lock. This field can contain LOCK_AVAIL (the lock is available), and LOCK_UNK_OWNER (the owner of the lock is unknown). This last condition applies when multiple read locks are made (there is no provision to remember all the processes making read locks, only the last read lock).

excls The specification of the exclusive lock function is such that the process should block until the lock can be retrieved. For this implementation this field specifies the exclusive event word.[1] The kernel uses the event_word parameter to anchor the list of processes sleeping on this event. The event_word parameter must be initialized to EVENT_NULL before its first use.

shares The specification of the shared lock function is such that the process should block until the lock can be retrieved. For this implementation this field specifies the shared event word.[2] The kernel uses the event_word parameter to anchor the list of processes sleeping on this event. The event_word parameter must be initialized to EVENT_NULL before its first use.

sh_cnt Used by V3 implementation to keep a count of shared locks currently held on the associated vproc.

ex_cnt Used by V3 implementation to keep a count of exclusive locks currently held on the associated vproc. The implementation of exclusive locks is such that the same process can gain an exclusive lock on the same vproc numerous times, though this is not a recommended operation.

The definitions below the lock structure (above) define the various types of vproc locks that can be attained. Each vproc lock will lock only specific fields of the appropriate vproc. This is a fine grained locking strategy that will (hopefully) allow

1. The usage of the excls field depends upon the semantics of how the process gets blocked when the lock cannot be gained.
2. The usage of the shares field depends upon the semantics of how the process gets blocked when the lock cannot be gained.

other operations on the same vproc to continue if each operation is mutually exclusive.

Used in conjunction with the lock types above to conditionally gain a lock if it can be obtained immediately, the **CONDITIONAL_LOCK** flag can be used with the lock types when attempting to gain a lock. If the lock is unobtainable, the conditional lock will not wait and returns failure. Normal unlock operations are used to relinquish a conditional lock. A conditional lock is used at high priority times such as at interrupt levels when it is highly undesirable to wait for a lock to become freed.

To implement the locking strategy, various functions need to be defined which will perform the actual lock operations. These are now defined below.

2.2.1.2.3.2 lock_shared()

```
bool_t lock_shared(lock, locktype)
locks_t *lock;
long locktype;
```

Parameter

`lock` — lock structure associated with a vproc.
`locktype` — flags defining locks to gain and the lock type.

Description

Acquire a shared lock, blocking if unavailable. The appropriate lock can be granted if no one currently claims the lock, denoted by LOCK_AVAIL, or if the lock is held by the same process. To determine if the lock can be given out, there cannot be any exclusive locks being held by other processes; as many shared locks can hold this lock, though the lock owner will only be know as the process which gained the lock last.

If the lock cannot be gained immediately and the CONDITIONAL flag is set then return a value of FALSE, otherwise block until the lock becomes available.

2.2.1.2.3.3 unlock_shared()

```
void unlock_shared(lock, locktype)
locks_t *lock;
long locktype;
```

Parameter

`lock` — lock structure associated with a vproc.
`locktype` — flags defining locks to release and the lock type.

Description

Release a shared lock, readying all processes currently blocked which waiting to gain this lock. Verify the lock is currently being held by a process, though the process cannot be verified due to the rules that many processes can hold a shared lock on a specific vproc. If however this process is the specified owner of the lock, change the lock owner to LOCK_UNK_OWNER. If after the unlocking there are no shared or exclusive locks set the owner of the lock to LOCK_AVAIL. If there were any requests for exclusive locks then wakeup those processes.

2.2.1.2.3.4 lock_exclusive()

```
void lock_exclusive(lock, locktype)
locks_t *lock;
long locktype;
```

Parameter

`lock` — lock structure associated with a vproc.
`locktype` — flags defining locks to gain and the lock type.

Description

Acquire an exclusive lock, blocking if the lock cannot be gained. If the CONDITIONAL flag is set and the lock cannot be immediately gained then return a value of FALSE. An exclusive lock can be granted only in two situations; if the calling process has already gained the lock then simply increment the count of exclusive locks on this vproc; otherwise the lock must be available, denoted by the owner field set to LOCK_AVAIL. If the lock is gained then set the lock id to the calling process and set the count of exclusive locks to one.

2.2.1.2.3.5 unlock_exclusive()

```
void unlock_exclusive(lock, locktype)
locks_t *lock;
long locktype;
```

Parameter

`lock` — lock structure associated with a vproc.
`locktype` — flags defining locks to gain and the lock type.

Description

Release an exclusive lock, readying all processes waiting for this lock to become free. Verify that the owner of the exclusive lock is the calling process and that the count of

exclusive locks is at least one (note that the same process can gain multiple numbers of the same exclusive lock, and must unlock it the same number of times to become free). Make the lock available again (LOCK_AVAIL) only if we have given up all exclusive locks AND there are no shared locks either. If all exclusive locks are given up and we still have a shared lock, then we still know who owns th elock so LOCK_UNK_OWNER is not set.

2.2.1.2.3.6 Example Locking Situations

This example pertains to a parent process exiting. All of its children must be reassigned to the init process. Note that this is an algorithm and does not necessarily represent the actual function definitions and operations that would be in a vproc implementation.

```
parent_vproc = vproc associated with parent process;
lock_exclusive(parent_vproc, PARENT);
for (child_vproc = all children on the parent_vproc list) {
    remove child_vproc from parent_vproc list;
    VPROC_UNMARK_CHILD(child_vproc);
    unlock_exclusive(parent_vproc, PARENT);
    VPOP_PVPROC_REASSIGN_CHILD(child_vproc);
    lock_exclusive(parent_vproc, PARENT);
}
unlock_exclusive(parent_vproc, PARENT);
```

The parent is initially locked with the parent exclusive lock since we will be modifying the parent-child-sibling list. For each child on this list, it is removed from the parents list, then the parent unmarks the child since the child no longer has a parent executing on this node. The parent then unlocks the exclusive lock before calling VPOP_PVPROC_REASSIGN_CHILD(). This is done because it is a function call; possibly a remote operation to reconnect a child executing on a remote node. Once this operation is complete, the parent gets back the exclusive lock before the next iteration. Once all children are removed, the exclusive lock is given up since it is no longer needed.

2.2.1.2.4 Implementation and Network Flag Values

All flag values used in this implementation of the vproc interface are given below. These values can be used to define the attributes of a process by being set in the vp_flag field of the private data structure, or just be flags used by the operations. If the flag values are involved in a remote operation, a network mapped value must exist such that different vproc implmentations don't interpret a numeric value to mean a different symbolic flag.

The table below lists any flag values currently supported by this vproc implementation. The current implementation value is given, and if the flag can be used in a remote operation the network mapped value is also given. This list of

course excludes all of the UNIX errno values which are too many to list here. However, since errno values can come from remote operations, the network value is defined to be the numeric values as defined by AIX V3.1. Additional errno values which are not defined will be added to this table.

The transport layer is responsible for mapping any values on a specific node to the defined network value before sending that value to another node. The network mapped value must also be converted to the locally defined value before it can be used by a process executing on that node. The table below describes the value used for this implementation, along with the network mapped value. All numeric values specified in the table are decimal.

Flag Name	Implementation Value	Network Mapped Value
PV_NOSETSID	1	1
PV_PGRPLEADER	2	2
PV_SESSIONLEADER	4	4
PV_ORPHAN_PGRP_CONTRIBUTOR	8	8
PV_SZOMB	16	16
PV_SSTOP	32	32
PV_MIG_INCOMPLETE	64	64
DECREMENT	1	1
INCREMENT	2	2
EXITOPERATION	4	4
SESSIONOPERATION	8	8
VPROC_SET	64	64
VPROC_GET	128	128
VPROC_CLEAR	256	256
VPROC_PRIV	512	512
SET_STOP_STATE	64	64
UNSET_STOP_STATE	256	256
NOPGRPOPERATION	13	13*

2.2.1.3 Base Code Modifications

Adding the vproc interface to AIX V3.1 does require some modifications to the base kernel code. The routines described below are those routines which are modified to support the vproc interface. In this context, the phrase "modified routine" can also mean a routine which has been deleted; routines are deleted only if the vproc interface will provide support for the operation.

2.2.1.3.1 strtdisp

```
void strtdisp()
```

Parameter

n/a

Description

Strtdisp() initializes the process dispatcher, init, and wait processes. The additional work which must be done is to create a vproc for each process. There is some process relationship information which *strtdisp()* does modify on the physical process and with the introduction of vprocs to the system, this information must be applied to the associated vproc.

Under normal circumstances the vproc operations set `u.u_error` in case of any errors occuring. Since a u-area has set to be created when *strtdisp()* is called, an internal vproc routine called *vops_fork_relationships1* is invoked. A locally defined error value is passed as an argument to be returned with a value in case a failure occurs.

2.2.1.3.2 newproc()

```
struct proc *newproc(register int check_again, register char *error)
IN int check_again;
INOUT char *error;
```

Parameter

check_again — nonzero, if caller not UID 0.
error — address to return errno value.

Description

The only modification to this routine is the process ID is no longer generated here. In addition, any process relationship operations, such as building the parent- child-sibling lists are no longer done here either; it is no performed in *vops_fork_relationships*.

2.2.1.3.3 freeproc

```
void freeproc(p)
struct proc *p;
```

Parameter

p — address of process entry to be deallocated.

Description

Freeproc() returns a process slot to the free list. Delete the process entry from the UID list and call freeprocslot() to return the proc entry to the free list. The process group and session lists are kept in the associated vproc so the code is modified not to check for these lists.

2.2.1.3.4 update_proc_slot

```
void update_proc_slot(p)
struct proc *p;
```

Parameter

p — physical process slot being deallocated.

Description

Normally for V3, a terminating process sees if the physical processes for the pgrp leader and session leader of the current process can be deallocated because the pgrp leader and session leader are no longer needed (i.e. the last process in process group and process group leader has already exited). This is now a null operation, process groups and sessions are implemented using vprocs. The function is needed due to calls outside of the vproc layer.

2.2.1.3.5 fork

```
pid_t fork()
```

Parameter

n/a

Description

Creation of a new process image from the parent process must also take into consideration that a vproc for the newly created process must also be created. Once the child process has been allocated a physical process structure and a new process image has been successfully created using *procdup()*, the child vproc can be allocated. It is important to understand that the vproc allocation must occur after it is certain that the new process can be created due to the fact that any process relationship information built is almost impossible to reverse.

After the parent process finishes duplicated a new child process, a new process ID is created using *pidgen()*, and *vops_fork_relationships* is called to allocate the child vproc.

2.2.1.3.6 creatp

```
pid_t creatp()
```

Parameter

n/a

Description

Create a kernel process that will eventually be initialized and dispatched by initp(). The caller can set up queues and other resources prior to the process being readied. Once the physical process structure has been allocated, a new process ID is allocated by a call to *pidgen()*. The vproc of the calling process must be found and used as the parent vproc parameter when allocating a new vproc with *vops_fork_relationships*.

2.2.1.3.7 initp

```
void initp(pid, init_func, init_data_addr, init_data_length, name)
pid_t pid;
int (*init_func)();
char *init_data_addr;
int init_data_length;
char name[];
```

Parameter

pid — process identifier.
init_func — pointer to initialization function.
init_data_addr — initialization data address.
init_data_length — initialization data length.
name — name of the process.

Description

Initialize a kernel process, completing the work of building the kernel process that was started in *creatp()*. The caller must be the same process that originally called *creatp()*. The vproc associated with the process ID passed as a parameter is located, then the physical process structure is accessed from the vproc. The process is then initialized accordingly.

2.2.1.3.8 setpinit

```
int setpinit()
```

Parameter

n/a

Description

Set the process parent id to init. This routine may only be called by a kernel process. The kernel process calling this routine must have its parent-child-sibling list updated, and as such the operation *vops_setpinit* is called to perform these operations.

2.2.1.3.9 proc1restart

```
void proc1restart()
```

Parameter

n/a

Description This routine restarts the INIT process when it dies. A new vproc must be allocated, along with rebuilding all of the appropriate process relationship attributes for the INIT process.

2.2.1.3.10 kexit

```
void kexit(wait_stat)
int wait_stat;
```

Parameter

wait_stat — return value to parent.

Description

Kexit() is the common exit code for process termination. It can be called directly from signal code, and it is called by the system call *_exit()*. Release the physical process resources and reassign all of the process' children to the INIT process. If the parent of the exiting process is ignoring SIGCHLD, the vproc of the exiting process is removed from the parent-child-sibling list and reassigned to be a child of INIT. Mark this process as a ZOMBIE. Control should not return to the caller after this point.

All process relationship information as described in the previous paragraph is now handled through the *vops_exit_relationships()* operation, including the process being marked as a zombie process.

2.2.1.3.11 kwaitpid

```
pid_t kwaitpid(stat_loc, pid, options, ru_loc)
int *stat_loc;
pid_t pid;
int options;
struct rusage *ru_loc;
```

Parameter

stat_loc — user location for returned status.
pid — pid value, −1, 0, −process group id.
options — options to vary function, see wait.h.
ru_loc — pointer to child resource usage area.

Description

This function incorporates all the functionality of wait(), wait3(), and waitpid(). A process will call kwaitpid() to wait for zombies or traced child processes. The termination status of the found child is returned, along with any resource statistics which the child process used during its execution.

This function is modified due to the implementation specific aspects of vprocs. After checking for debugging processes, a vproc operation is used to scan the parent-child-sibling list which is located in the pvproc object of a vproc.

2.2.1.3.12 kill

```
int kill(pid, sig)
pid_t pid;
int sig;
```

Parameter

pid — process(es) or process group to signal.
sig — signal to be sent.

Description

Send a signal to a process or multiple processes. If the pid argument is zero, the signal specified is targeted for the process group of the currently running process. A nonzero pid argument specifies that a signal be sent to a specific process. The vproc for the specified process is used as an argument to a vproc operation to determine if the signal can be sent by the currently executing process.

The vproc for the specified process receiving the signal is used when gaining access to the specified process, when checking for privilege to send the signal, and when making the signal request.

The semantics of the *kill -1* operation depends upon the calling process. If the calling process has superuser authority, a signal is sent to all processes except kernel processes; the process table is traversed on the local node and all non-kernel processes are located and a pidsig() operation is performed. If the calling process does not have superuser authority, the signal will be sent to all processes with the same uid in the process's **SOI** (sphere of interest). An RPC operation is sent to all nodes in the SOI; on each node the process table is searched to find those executing processes with the same uid value. A signal is then delivered using the pidsig() operation.

Support for process migration requires an extra argument to specify the node which a process will be migrated. Since the kill() system call supports only two arguments, kill() is modified to call killarg() with a NULL third parameter. Killarg() provides complete support for the kill system call in addition to support for process migration.

2.2.1.3.13 kill3

```
int kill3(pid, sig, node)
pid_t pid;
int sig;
node_t node;
```

Parameter

pid — process (es) or process group to signal.
sig — signal to be sent.
node — if SIGMIGRATE, node to which process is migrated.

Description

Kill3() is a new system call similar to kill, but permits an addition argument to be specified which provides information passed along when the signal is delivered. Kill3() is available for all signals. For **SIGMIGRATE**, the additional information is interpreted as a node specifier indicating the node to which the process or processes should be migrated.

2.2.1.3.14 killarg

```
int killarg(pid, sig, node)
pid_t pid;
int sig;
node_t node;
```

Parameter

`pid` — process (es) or process group to signal.
`sig` — signal to be sent.
`node` — if SIGMIGRATE, node to which process is migrated.

Description

This function is added to support both the *kill()* and *kill3()* system calls. The implementation is such that *killarg()* acquires all the functionality of the previous *kill()* system call, in addition to support for the extra signal argument.

2.2.1.3.15 pgsignal

```
void pgsignal(pid, signo)
pid_t pid;
int signo;
```

Parameter

`pid` — process group to signal.
`signo` — signal to be sent.

Description

Send a signal to all processes in the specified terminal process group. The signal operation is performed by locating the vproc of the process group leader and sending a signal to all processes which have their vproc objects on the process group list.

This routine is modified to operate using a vproc given a process id. A vproc operation is used to gain access to the process group leader, then scan for vprocs on the process group list. Support for process migration requires that an extra parameter be sent by pgsignal; this value is initialized to NULL.

2.2.1.3.16 pidsig

```
void pidsig(pid, signo)
pid_t pid;
int signo;
```

Parameter

`pid` — identifier of process to receive the signal.
`signo` — signal to be sent.

Description

Pidsig() sends a signal to a process. The vproc for which the pid argument identifies is located and then a vproc operation is used to send the signal specified to the process. Support for process migration requires that an extra parameter be sent by pidsig; this value is initialized to NULL.

2.2.1.3.17 sigpriv

The vproc operation *vops_sigproc* is overloaded to perform signal privilege determination. This is done to reduce the overhead of two remote operations when a process must first get privilege to signal a process before the signal is sent. The two operations are now combined, along with the ability to only check for privilege to send a signal to a process.

2.2.1.3.18 stop

```
void stop(p)
struct proc *p;
```

Parameter

p — process to be stopped.

Description

Stop() puts the process in STOP state, signals and/or wakes up the parent. If the STRC flag is not set there exists the possibility that a SIGCHLD signal will be sent to its parent process. A vproc operation is called to potentially send the SIGCHLD signal.

The PV_SSTOP flag is set within the pvproc object of the stopped process to optimize the search for SSTOP and ZOMBIE processes by its parent process during the *kwaitpid()* routine. This optimization will hold true only if remote processing is installed on the system.

2.2.1.3.19 getpri

```
int getpri(pid)
pid_t pid;
```

Parameter

pid — process id or current process.

Description

Getpri() returns the priority of a specified fixed priority process. If the pid argument is 0, use the currently running process's priority; otherwise get the priority from the

vproc associated with the pid. *Getpri()* is modified to operate using vprocs. If the pid argument is zero the vproc of the current process is used rather than the proc structure of the current process. A nonzero pid argument requires that the vproc is located using the pid value.

2.2.1.3.20 setpri

```
int setpri(pid, pri)
pid_t pid;
int pri;
```

Parameter

`pid` — pid of process to change or current process.
`pri` — priority to be set.

Description

Setpri() enables a process to run with a fixed priority. Call *privcheck()* to make sure we can reset the priority of this process. Use the current running vproc or call *LOCATE_VPROC_PID()* to get the vproc for the specified PID.

Setpri() is modified to operate using vprocs. If the pid argument is zero the vproc of the current process is used rather than the proc entry. A nonzero pid argument requires that the vproc is located using the pid value.

2.2.1.3.21 getpriority

```
int getpriority(which who)
int which;
int who;
```

Parameter

`which` — identifies how the "who" parameter is handled.
`who` — process, process group, or UID.

Description

Get the scheduling "nice" value of the process, process group, or user as indicated by "which" and "who". "Which" is one of PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, and "who" is interpreted relative to "which". A zero value of "who" denotes the current process, process group, or user. A non zero "who" value requires the location of the vproc which has the specified pid argument.

The nice value for a process is retrieved using a vproc operation to locate the process. Getting the nice value for a process group requires a vproc operation to scan the list of vprocs on the process group list, getting the nice value for each of them in succession.

Getting the nice value for all processes with a specified uid uses the uidl field on the physical process structure. This field implements a circular list of processes with the same uid. In the case where the uid specified is for the currently executing process, the uidl field of the currently executing process is used to begin a traversal to find the nice value.

If the uid value is specified, all nodes in the SOI (sphere of interest) are located and an RPC operation is performed on each of these nodes; The process table on each node is then scanned to find those processes with a matching uid value and the nice value is retrieved.

2.2.1.3.22 setpriority

```
int setpriority(which, who, nice)
int which;
int who;
nice_t nice;
```

Parameter

which — identifies how the "who" parameter is handled.
who — process, process group, or UID.
nice — nice value to be set.

Description

Set the scheduling "nice" value of the process, process group, or user, as specified by the "which" and "who" parameters. The "nice" parameter is a value in the range –20 to 20. Lower priorities cause more favorable scheduling. If the value is less than –20, –20 is used. If greater than 20, 20 is used.

The priority for a process is set using a vproc operation to locate the process. Setting the nice value for the process group requires the use of a vproc operation to scan the process group list to find all members of the process group, then set the nice value for each in succession.

Setting the nice value for all processes with a specified uid uses the uidl field on the physical process structure. This field implements a circular list of processes with the same uid. In the case where the uid specified is for the currently executing process, the uidl field of the currently executing process is used to begin a traversal to set the nice value.

If the uid value is specified, all nodes in the SOI (sphere of interest) are located and an RPC operation is performed on each of these nodes; The process table on each node is then scanned to find those processes with a matching uid value and the nice value is set.

2.2.1.3.23 get_pgrp_nice

Get_pgrp_nice() is deleted from the fixed base code. Vproc operations are available in the linked base code to perform this function.

2.2.1.3.24 set_pgrp_nice

Set_pgrp_nice() is deleted from the fixed base code. Vproc operations are available in the linked base code to perform this function.

2.2.1.3.25 setpgrp

```
pid_t setpgrp()
```

Parameter

n/a

Description

Setpgrp() will make the currently executing process the leader of a new process group, if not already leader. Call *setpgid()* with both arguments of zero. *Setpgrp()* is modified to return the process group id from the vproc associated with the currently executing process if the operation succeeded.

2.2.1.3.26 setpgid

```
int setpgid(pid, pgrp)
pid_t pid;
pid_t pgrp;
```

Parameter

pid — process group ID of process to set.
pgrp — process group ID to join.

Description

Setpgid() will set the process group id of the specified process in the vproc. Check for errors and then create a new process group with this process as the process group leader and only process within the group.

If the pid argument is zero, use the vp_pid value from the vproc of the currently executing process. Use *LOCATE_VPROC_PID()* to get the vproc's for the process

being modified and the process group leader. Vproc operations are required to remove the process from its current group, creation of a new process group, or adding the process to an existing process group.

After the new process group leader has been established, check to see if any orphan process groups are created due to this operation. This is performed using a vproc operation since certain orphan information is kept in the pvproc object of each vproc.

2.2.1.3.27 setsid

`pid_t` ***setsid()***

Parameter

`n/a`

Description

Setsid() will set the currently executing process to a session leader. If the calling process is not a session leader, a new session is created. Perform the mechanics of creating a new session with no controlling terminal for the currently executing process. Resign from the previous session if this vproc was in one, then call a vproc operation to perform the implementation specific aspects of creating a new session. *

2.2.1.4 Remote process management

Support for remote processing is implemented with a client/server model and a set of *wrapper* routines. The transport mechanism being used is NCS (described in another section) which is used to connect the client routine with the appropriate server routine on the alleged execution node of the process.

The two diagrams below illustrate the various levels of vproc support when only local processing is allowed on a specific node, and when remote processing is installed on a node. Local processing constitutes fixed base code along with routines which make up the local vproc and pvproc operations. The operations installed within the fixed base code call these local operations through the vproc/pvproc tables.

Installation of remote processing adds a certain amount of complexity to the system.

- Local vproc ops and pvproc ops tables get replaced with wrapper routines. These routines perform any required vproc locking and determine if the process is executing on this node.
- The local pproc/pvproc routines do not disappear, rather they are called by the wrapper routines once it has been determined that the vproc for this operation is executing on this node.
- Vprocs for a process that is not executing on this node have their vproc and pvproc ops tables replaced with remote tables which call the appropriate client

routines to initiate a remote operation. Eventually the wrapper routine on the execution node of the vproc gets called to perform the request on the local execution node.

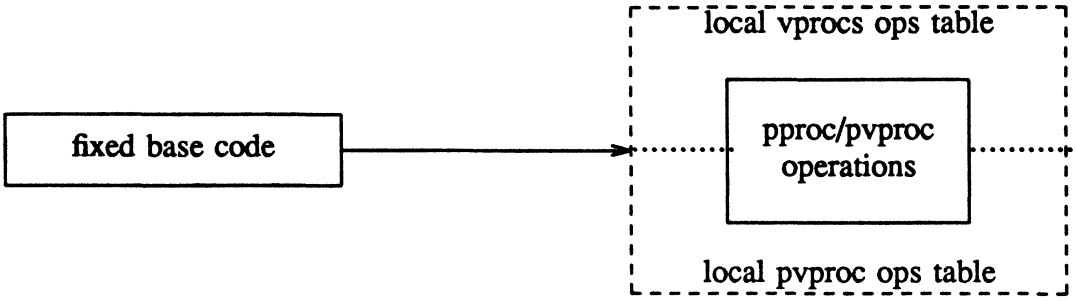


Figure 10. Fixed Base Code Supporting Vprocs Without Remote Processing Installed

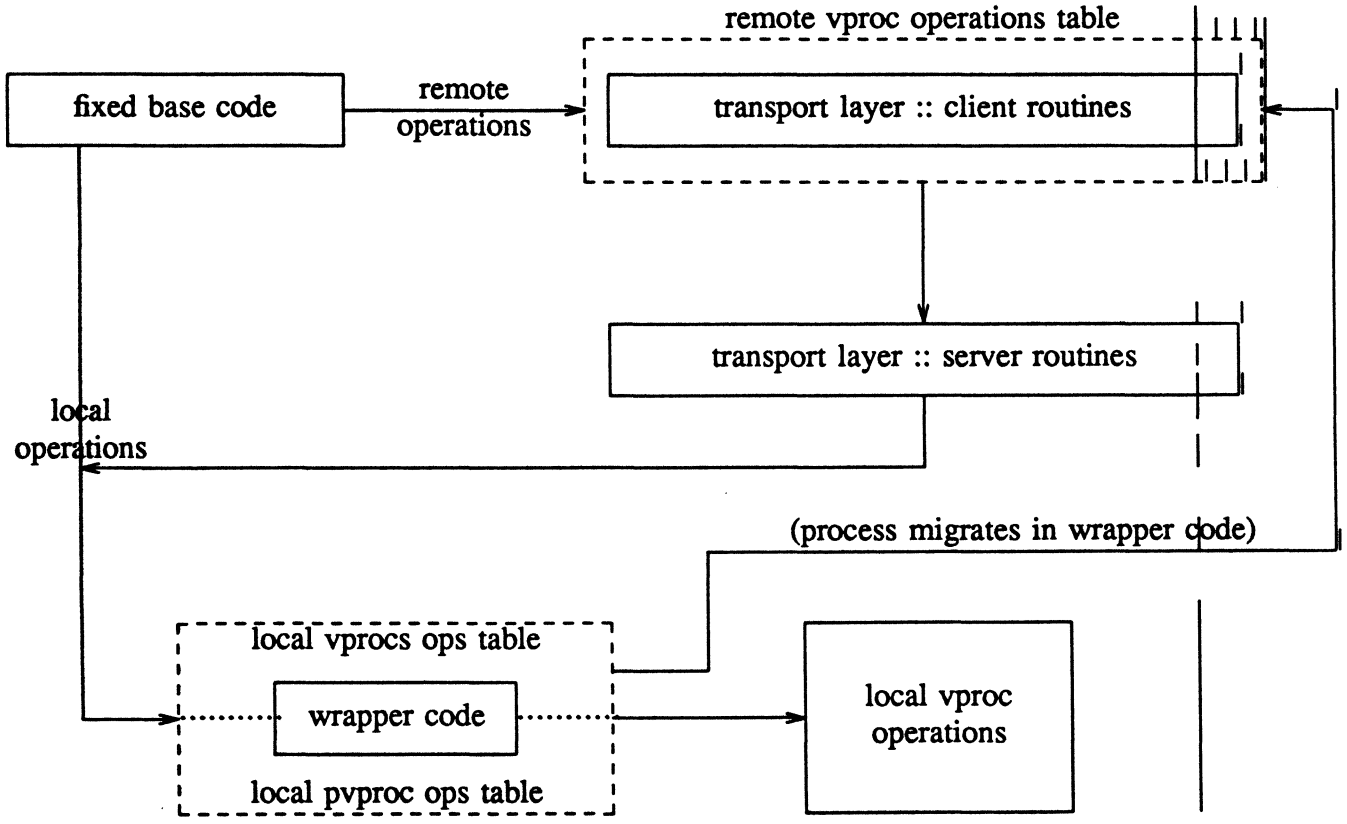


Figure 11. Fixed Base Code Supporting Vprocs With Remote Processing Installed

Local vproc operations call the appropriate wrapper routine from the vproc operations within fixed base code. Provided that the process is still local throughout the wrapper

code the local pproc/pvproc operation is ultimately called to complete the operation.

If the vproc operation is to be performed upon a remote process, the remote vproc tables have been installed for that vproc which call the appropriate client routine. Ultimately the wrapper code on the execution node is call by the server code for which the RPC request from the client has been made.

Once in the wrapper code, any locks required are gained and the test is made to determine if indeed the process is on this execution node. If the process is executing here the appropriate operation is called through the local vproc/pvproc tables installed within this local vproc.

The determination of whether the execution node of the process has been found is through the operation VPROC_EXEC_NODE(v). If the vproc is not executing on this * node, certain conditions must be explored to determine what to do next. The test to determine if we are executing as a server process is through the operation SERVER_PROC().

1. If we were called from server code then the vproc is no longer executing on this remote node. A value is returned such that the client code can try another node where the vproc may be executing.

2. If the call was not from server code, the vproc may have been migrated from this node while we were gaining the appropriate locks. In this instance, the vproc which has migrated will have had its vproc/pvproc tables replaced with the remote tables which are the client routines used to initiate a remote request. The associated vproc/pvproc table routine is called to start the remote request.

   Assuming the client connects with a server on a node for which the process is executing, the server ultimately calls the wrapper routine on the remote node. This wrapper routine will call the local vproc/pvproc operation to complete the request.

In addition to the requirements described in the remote processing model above, | process group operations must be dealt with accordingly such that each member in the | process group is guaranteed to have the specified operation performed on itself exactly | one time. To support this requirement, a caching scheme is introduced.

2.2.1.4.1 Data Objects

Certain definitions and data structures are used by various portions of the remote processing sections.

The client/server routines are implemented as remote procedure calls (RPC) which interface with routines written using the NIDL language. The set of NIDL stub routines are for the most part hidden from the client/ server routines. However, specifying which node the RPC is to be sent requires the use of a **handle** which is a NIDL created data object. The handle is the first argument specified when calling a

server routine. The format of a handle is defined in the following manner.

```
typedef struct
{
    unsigned short data_offset;
} *handle_t;
```

The implementation of vprocs dictates that certain vprocs will exist on remote nodes if a process executes there. The pid, original execution node and current execution node need to be sent to the remote node such that when a vproc gets created on the remote node certain information for the vproc can be filled in. For example, if a process group leader vproc needs to be created on a remote node, the node for which the client initiated this request probably has the best idea as to where the process group leader is really executing. The same holds true for its origin node, and most importantly its process id such that the vproc can initially be found or created.

A data structure is created which contains this information. Since the pid resides in a vproc object whereas the original execution node and current execution node reside in the pvproc data structure, the client routines are responsible for building this data structure.

```
typedef ndr_long_int pt_pid_t;
typedef struct
{
    pt_pid_t vh_pid;
    ndr_long_int vh_orig_node;
    ndr_long_int vh_exec_node_hint
} pt_vproch_t;
```

2.2.1.4.2 Return values

The linked based code defined the return values SUCCESS and FAILURE which can be returned from the vproc and pvproc operations. Remote processing allows an additional return value known as **PROC_NOT_HERE**, to be returned by the client, server, and wrapper routines. This error value is returned when the vproc specified cannot be located on the remote node or is not executing on this node anymore. If the server determines the process is not executing on this node, a guess is made regarding where it is now executing. If the vproc is not found, the server will return the value **UNKNOWN_NODE** such that the client does not use this value and goes directly to the origin node of the process.

2.2.1.4.3 Process Group Caching

Sending a signal to a process group, accessing the nice values of a process group, and orphaning a process group are operations which require that each process within their process group be visited only once within a cluster. When processes all reside on a single node, this operation is simple. However, installing remote processing allows

processes within a process group to execute on remote nodes. A caching mechanism is designed to be installed with remote processes to guarantee that each process within a process group is visited only once for the operations described above.

The cache operations (with the exception of the cache init routine) are called from the wrapper, client and server routines described in the sections which follow. A cache exists on each node and is described in the following manner:

```
#define CACHE_ELEMENTS  1024                /* number of buckets in cache */
#define CACHE_HDR_ELEMENTS CACHE_ELEMENTS /* cache header elements */
#define MAX_RPC_BUCKETS 10                  /* maximum buckets reserved for
                                               an rpc operation */
#define MAX_RPC_ELEMENTS MAX_RPC_BUCKETS  /* maximum elements delivered in
                                               rpc IN and OUT arrays */

struct hashtype {
    long flag;
    union {
        sig_t  signal;          /* signal for pgsignal operation */
        int    pgrp;            /* process group identifier */
    } un;
};

struct hashbucket {
    pid_t pid;                  /* uniquely identifies vproc */
    struct hashtype *ht;
    struct hashbucket *hptr;    /* pointer to next hashbucket */
};

struct hashbucket *hashhdr[CACHE_HDR_ELEMENTS],     /* hash header */
                  *hashfreelist;                    /* hash freelist */
long   hashfree;                                    /* number free */
```

To provide a uniform interface to the cache algorithms, the operations below are provided. This method will effectively hide the implementation of the cache from the client, server, and wrapper routines and does allow the cache to be expanded to handle other data types if required.

```
#define SIG_CACHE_REMOVE_IF_PRESENT(pid, sig, retval) {
            struct hashtype ht;
            ht.flag = SIGNAL_OP;    ht.un.signal = sig;
            retval = cache_remove_if_present(pid, &ht);
        }
#define NICE_CACHE_REMOVE_IF_PRESENT(pid, pgrp, retval) {
```

```
            struct hashtype ht;
            ht.flag = NICE_OP;      ht.un.pgrp = pgrp;
            retval = cache_remove_if_present(pid, &ht);
        }
#define ORPHAN_CACHE_REMOVE_IF_PRESENT(pid, pgrp, retval) {
            struct hashtype ht;
            ht.flag = ORPHAN_OP;      ht.un.pgrp = pgrp;
            retval = cache_remove_if_present(pid, &ht);
        }
#define SIG_CACHE_ADD_TO_LIST(pid, sig, retval) {
            struct hashtype ht;
            ht.flag = SIGNAL_OP;      ht.un.signal = sig;
            retval = cache_add_to_list(pid, &ht);
        }
#define NICE_CACHE_ADD_TO_LIST(pid, pgrp, retval) {
            struct hashtype ht;
            ht.flag = NICE_OP;      ht.un.pgrp = pgrp;
            retval = cache_add_to_list(pid, &ht);
        }
#define ORPHAN_CACHE_ADD_TO_LIST(pid, pgrp, retval) {
            struct hashtype ht;
            ht.flag = ORPHAN_OP;      ht.un.pgrp = pgrp;
            retval = cache_add_to_list(pid, &ht);
        }
```

Software Priority Level

Cache access is guarded by high spl level. This must be so because the cache can be accessed at interrupt level. The number of free buckets and the presence of an element in the cache can be unreliably checked at low spl level, but reliable checks and any list manipulation must be done at high spl level.

The cache on the execution node of the process group leader is used for a specified process group operation. The remote client routine first determines if the vproc specified for this operation is in the cache. If so, the operation has already been performed on this vproc and the client returns success. If not, the client will first reserve the maximum number of hash buckets under the assumption that the pending operation will be completely successful for the maximum case.

The process group list is traversed from its current point (do not start from the beginning of the list) to find the maximum number of vprocs executing on the node for which this operation is for. An array is used to keep a list of all vprocs which satisfy this requirement. The array is passed as an argument to the server routine and is used as an input parameter. Another array of the same size is also passed as a parameter; its use is to return the success or failure of the operation for each vproc in the first array.

The server routine is called, and if the operation was successful, then for each successful operation as specified in the array, that vproc is added to the cache using one of the reserved hash buckets. The remainder of reserved buckets, if any, are then freed.

The server routine is not as complex as the client. The operation is first performed on the vproc argument for which this is for. If the operation was successful, then for each of the vprocs in the input array, perform the same operation. As long as the vproc operation does not return PROC_NOT_HERE, note a successful operation in the appropriate output array entry.

The wrapper code only needs to check the specified vproc is in the cache and this is a process group operation due to the possibility that a vproc may have migrated from a remote node after the process group operation had been successfully performed and the vproc had been added to the cache.

2.2.1.4.3.1 cache_init — initialize the cache

```
void cache_init()
```

Parameter

```
Cn/a
```

Description This routine is responsible for initializing the cache. This entails allocating the hashbuckets for the node and initializing the freelist and the number of hashbuckets allocated.

2.2.1.4.3.2 cache_remove_if_present()

```
bool_t cache_remove_if_present(pid, hashval)
IN pid_t pid;
IN hashtype *hashval;
```

Parameter

```
pid — process ID of vproc to be removed from cache.
hashval — information to create hash value.
```

Description

This routine will determine if the vproc specified by the process identifier and hash value is in the cache. If the vproc is found it is removed from the cache, the hashbucket for which it contained is put back on the freelist and TRUE is returned. If the vproc is not found FALSE is returned. A pointer to a hashtype is passed as an argument such that the correct hash value can be constructed, depending upon the

operation being performed.

Note that the object type is also specified in the hash bucket. This is to uniquely identify the correct hash bucket in the event that a nice value operation and orphan operation on the same vproc (in the same process group) have added it to the cache.

2.2.1.4.3.3 cache_add_to_hashlist()

```
void cache_add_to_hashlist(pid, hashval)
IN pid_t pid;
IN hashtype *hashval;
```

Parameter

pid — process ID of vproc to add to cache.
hashval — information to create hash value.

Description

This routine will add the specified arguments to the cache by first constructing a hash value, removing a hash bucket from the freelist and putting the hash bucket into the cache.

Note that the object type is also specified in the hash bucket. This is to uniquely identify the correct hash bucket in the event that a nice value operation and orphan operation on the same vproc (in the same process group) are adding it to the cache.

2.2.1.4.3.4 cache_reserve()

```
bool_t cache_reserve(cnt)
IN int cnt;
```

Parameter

cnt — number of buckets to reserve.

Description

This routine will reserve the specified number of hashbuckets and return TRUE if successful, FALSE if unsuccessful.

2.2.1.4.3.5 cache_free()

```
void cache_free(cnt)
IN int cnt;
```

Parameter

cnt — number of buckets to free.

Description

This routine increments the number of free buckets in the cache by the specified argument amount.

2.2.1.4.4 Wrapper Routines

Wrapper routines replace the local vproc/pvproc ops tables when remote processing is installed on a node. The wrapper routines are responsible for gaining any vproc locks which may be needed, and ultimately calling either the client or local pproc/pvproc operations to fulfill the requested operation.

For each operation listed in the vproc operations table and pvproc operations table there exists a wrapper routine. The list of wrapper routines are given in the table which follows:

Vproc Wrapper Operations	Pvproc Wrapper Operations
fork_relationships_wrapper	*wait3_wrapper*
exit_relationships_wrapper	*reassign_child_wrapper*
wait_wrapper	*rmv_child_from_parent_wrapper*
proc_nice_wrapper	*rmv_child_if_no_sigchld_wrapper*
pgrp_nice_wrapper	*proc_flag_wrapper*
sigproc_wrapper	*proc_status_wrapper*
sigpgrp_wrapper	*pvproc_flag_wrapper*
set_stop_state_wrapper	*resign_pgrp_wrapper*
setpgid_wrapper	*add_pgrp_list_wrapper*
setsid_wrapper	*rmv_pgrp_list_wrapper*
get_pgrp_sid_wrapper	*add_session_list_wrapper*
setpinit_wrapper	*rmv_session_list_wrapper*
setpri_wrapper	*orphan_child_pgrp_wrapper*
getpri_wrapper	*adjust_orphan_count_wrapper*

Confining the vproc locking code to the wrapper routines removes the overhead for strictly local process execution and reduces the impact of adding vprocs to the fixed and linked based code. In addition, determining whether a process is local or remote is hidden from the local pproc/pvproc operations, thereby reducing the overhead for systems without process transparency. Most wrapper routines will follow the algorithm listed below.

```
int
vproc_operation_wrapper(
    struct vproc *v,
    int arg1,
    int arg2
```

```
)
lock_shared(v, GENERIC);
if (!VPROC_EXEC_NODE(v)) {
    unlock_shared(v, GENERIC);
    if (SERVER_PROC())
        return(PROC_NOT_HERE);
    (void) (*(v)->vp_vops->vpop_operation)(v, arg1, arg2);
    return(SUCCESS);
}
(void) (saved_vops.vproc_operation)(v, arg1, arg2);
unlock_shared(v, GENERIC);
return(SUCCESS);
```

In almost all situations the prescribed set of vproc locks can be gained before calling the local operation. However, if a complex locking/unlocking scheme is required that cannot be performed within the wrapper code alone, the wrapper routine is written such that the local operation is performed entirely within the wrapper and no other calls are made to a pproc/pvproc operation. Appendix E contains a complete listing of all wrapper operations.

2.2.1.4.5 Client Vproc Operations

The underlying server-to-server protocols are invoked to pass information and negotiations outside the normal flow of client/server communications.

Two tables exist which contain the client routines to initiate a remote request for the appropriate vproc operation. These two tables are a one to one mapping of the vproc and pvproc tables described in earlier sections of this document.

Client Vproc Operations	Client Pvproc Operations
vproc_fork_relationships	*vproc_wait3*
vproc_exit_relationships	*vproc_reassign_child*
vproc_wait	*vproc_rmv_child_from_parent*
vproc_proc_nice	*vproc_rmv_child_if_no_sigchld*
vproc_pgrp_nice	*vproc_flag*
vproc_sigproc	*vproc_status*
vproc_sigpgrp	*vproc_pvproc_flag*
vproc_set_stop_state	*vproc_resign_pgrp*
vproc_setpgid	*vproc_add_pgrp_list*
vproc_setsid	*vproc_rmv_pgrp_list*
vproc_get_pgrp_sid	*vproc_add_session_list*
vproc_setpinit	*vproc_rmv_session_list*
vproc_setpri	*vproc_orphan_child_pgrp*
vproc_getpri	*vproc_adjust_orphan_count*

With a few exceptions, the basic operation of the client routines is the same: Determine the execution node of the vproc which this operation is being requested and call the client stub routine to initiate an RPC request to the remote node. Upon completion check the return value for any errors that may have occurred. If the RPC request was sent but the vproc has moved to another node (PROC_NOT_HERE) a new guess as to the execution node will have been sent back by the server. Reattempt to perform the operation using the new execution node until a maximum number of tries has been made, at which time the origin site is queried to determine where the process is executing. There is the possibility that the vproc did not exist on the remote node; not only will PROC_NOT_HERE be returned, but the execution node guess will be sent to UNKNOWN_NODE to notify the client to go directly to the origin node.

Typical steps

The example below describes the operation for vproc_set_flag() and can be considered the general format of each of the client routines.

```
error_status_t
vproc_set_flag(
    vproc_t *v,     /* vproc of process to set flag value */
    int flag,       /* value of flag */
    int set_clear   /* determines whether to set or clear the flag */
    )
    VPROC_HOLD(v, "GENERIC");
    for (rpc_tries = 1;; rpc_tries++) {
        h = handle for node where vproc is executing;
        ret = vprocs_set_flag(h, &v->vp_handle, flag, set_clear, &execnode);
        if (ret = successful vproc operation) {
            VPROC_RELEASE(v, "GENERIC");
            setuerror(ret);
            return(SUCCESS);
        }
        ret = check_failure(v, rpc_tries, &first_try, &origin_node_contacted,
                            &h, execnode);
        if (ret == FAILURE) {
            VPROC_RELEASE(v, "GENERIC");
            return(FAILURE);
        }
    }
```

The vproc is temporarily marked to ensure the vproc is not deallocated for the duration of this operation. The handle retrieved is implementation dependent upon the transport layer used and does not necessarily need defining within this document. The call to *vprocs_set_flag* is a transport layer defined routine which invokes the request to the remote node. Coincidentally its name is the same as the corresponding server

routine which satisfies the request on the specified remote node. They have nothing to do with each other and should not be confused. If the return value from the call is not successful, *check_failure()* will determine the next action to take (described below).

The remainder of this section describes the *check_failure()* routine along with those client operations which will deviate from the general format described above.

2.2.1.4.5.1 check_failure

```
error_status_t
check_failure(v, rpc_tries, first_try, origin_node_contacted, execnode, h)
IN struct vproc *v;
IN int rpc_tries;
INOUT bool_t *first_try;
INOUT bool_t *origin_node_contacted;
IN node_t execnode;
INOUT handle_t *h;
```

Parameter

v — vproc associated with process operation.
rpc_tries — number of RPC attempts for this operation already.
first_try — is this the first RPC for this operation.
origin_node_contacted — has origin node already by contacted.
execnode — latest clue as to the execution node for this vproc.
h — handle for new RPC to be sent.

Description

This operation determines the next action to take if an RPC for a vproc operation resulted in a failure to complete. If the RPC was unable to be delivered, another try is made before returning a failure of ESRCH. In the case when PROC_NOT_HERE is returned, a determination needs to be made if the server node actually had a decent guess as to the location of the process, does the origin node need to be queried, or if we have exceeded the maximum allowable attempts to find the process. A return value of SUCCESS from check_failure() is defined as another RPC should be sent with the values specified; a return of FAILURE indicates a failure in sending the RPC and return to the calling function is done.

2.2.1.4.5.2 vproc_proc_nice()

Either determine the lowest nice value or set the nice value of all processes within a process group. Since this operation occurs on a process group, the caching algorithms are used to guarantee each process within the process group is visited. The server routine is responsible for determining the lowest nice value of those processes which

were successful operations. *

```
int
vproc_proc_nice(                                                    |
    vproc_t *v,         /* vproc for which nice value is being extracted */
    nice_t  *nice,      /* nice value location */                   |
    int flag            /* is this part of a get_pgrp_nice() operation ? */
    )
        pid_t  INarray[MAX_RPC_ELEMENTS];
        bool_t OUTarray[MAX_RPC_ELEMENTS];

    VPROC_HOLD(v, "GENERIC");                                       |
    max = 0;
    if (flag specifies process group operation) {
        NICE_CACHE_REMOVE_IF_PRESENT(v->vp_pid, v->pv_pgrp, retval)
        if (retval == TRUE) {
            VPROC_RELEASE(v, "GENERIC");                            |
            return(SUCCESS);
        }
        cache_reserve(MAX_RPC_BUCKETS);
        for (m = v->pv_pgrp; max < MAX_RPC_BUCKETS; m = m->pv_pgrp) {
            if (execution node of m == execution node of v) {
                INarray[max] = m->vp_pid;
                OUTarray[max++] = FALSE;
            }
        }
    }
    while (max < MAX_RPC_BUCKETS) {
        INarray[max] = PID_INVALID;
        OUTarray[max++] = FALSE;
    }
    for (rpc_tries = 1;; rpc_tries++) {
        h = handle for node where vproc is executing;
        ret = vprocs_proc_nice(h, &v->vp_handle, nice, flag, INarray,  |
                               OUTarray, &execnode);
        if (successful vproc operation) {
            setuerror(ret);
            break;
        }
        ret = check_failure(v, rpc_tries, &first_try, &origin_node_contacted,
                            &h, execnode);
        if (ret == FAILURE) {
            VPROC_RELEASE(v, "GENERIC");                            |
```

```
            return(FAILURE);
        }
    }

    used_buckets = 0;
    if (flag specified process group operation) {
        for (max = 0; max < MAX_RPC_BUCKETS &&
                    INarray[max] != PID_INVALID; max++) {
            if (OUTarray[max] == TRUE) {
                NICE_CACHE_ADD_TO_LIST(INarray[max], v->pv_pgrp, retval);
                used_buckets++;
            }
        }
        cache_free(MAX_RPC_BUCKETS - used_buckets);
    }
    VPROC_RELEASE(v, "GENERIC");
```

2.2.1.4.5.3 vproc_sigproc()

Sending a signal to a process may be a subset of sending a signal to a process group. The client function is modified to used the cache mechanism described earlier.

```
int
vproc_sigproc(
    vproc_t *v,     /* vproc being sent a signal */
    int signo,      /* signal number */
    uid_t effuid,   /* effective id of calling process */
    uid_t realuid,  /* real id of calling process */
    pid_t sid,      /* session id of calling process */
    int *has_priv,  /* used for determining signal privilege */
    int flag        /* is this part of a pgsignal() operation ? */
    )
        pid_t  INarray[MAX_RPC_ELEMENTS];
        bool_t OUTarray[MAX_RPC_ELEMENTS];

    VPROC_HOLD(v, "GENERIC");
    max = 0;
    if (flag specifies process group operation) {
        SIG_CACHE_REMOVE_IF_PRESENT(pid, sig, retval)
        if (retval == TRUE) {
            VPROC_RELEASE(v, "GENERIC");
            return(SUCCESS);
        }
        cache_reserve(MAX_RPC_BUCKETS);
        for (m = v->pv_pgrp; max < MAX_RPC_BUCKETS; m = m->pv_pgrp) {
```

```
            if (execution node of m == execution node of v) {
                INarray[max] = m->vp_pid;
                OUTarray[max++] = FALSE;
            }
        }
    }
    while (max < MAX_RPC_BUCKETS) {
        INarray[max] = PID_INVALID;
        OUTarray[max++] = FALSE;
    }
    for (rpc_tries = 1;; rpc_tries++) {
        h = handle for node where vproc is executing;
        ret = vprocs_sigproc(h, &v->vp_handle, effuid, realuid, sid, flag, sig,|
                        INarray, OUTarray, &execnode);                          |
        if (successful vproc operation) {
            setuerror(ret);
            break;
        }
        ret = check_failure(v, rpc_tries, &first_try, &origin_node_contacted,
                            &h, execnode);
        if (ret == FAILURE) {
            VPROC_RELEASE(v, "GENERIC");                                        |
            return(FAILURE);
        }
    }

    used_buckets = 0;
    if (flag specifies process group operation) {
        for (max = 0; max < MAX_RPC_BUCKETS &&
                        INarray[max] != PID_INVALID; max++) {
            if (OUTarray[max] == TRUE) {
                SIG_CACHE_ADD_TO_LIST(INarray[max], sig, retval);
                used_buckets++;
            }
        }
        cache_free(MAX_RPC_BUCKETS - used_buckets);
    }
    VPROC_RELEASE(v, "GENERIC");                                                |
```

2.2.1.4.5.4 vproc_sigpgrp()

Invocation of this routine implies a signal is being sent to a process group for which the process group leader is executing on a remote node. If this operation has been called at interrupt time, add the signal request to the signet queue using the appropriate routine as specified in the design specification for the signet daemon. |

2.2.1.4.5.5 vproc_proc_flag()

If the flag value being set or cleared is SORPHAN_PGRP then this is part of a process group operation and the cache algorithms are used.

```
int
vproc_proc_flag(
    vproc_t *v,         /* vproc for which nice value is being set */
    u_long flag,        /* flag value to be set or cleared */
    int set_clear       /* do we set or clear the flag value */
    )
        pid_t  INarray[MAX_RPC_ELEMENTS];
        bool_t OUTarray[MAX_RPC_ELEMENTS];

    VPROC_HOLD(v, "GENERIC");
    max = 0;
    if (flag == SORPHAN_PGRP then this is a process group operation) {
        ORPHAN_CACHE_REMOVE_IF_PRESENT(v->vp_pid, v->pv_pgrp, retval)
        if (retval == TRUE) {
            VPROC_RELEASE(v, "GENERIC");
            return(SUCCESS);
        }
        cache_reserve(MAX_RPC_BUCKETS);
        for (m = v->pv_pgrp; max < MAX_RPC_BUCKETS; m = m->pv_pgrp) {
            if (execution node of m == execution node of v) {
                INarray[max] = m->vp_pid;
                OUTarray[max++] = FALSE;
            }
        }
    }
    while (max > MAX_RPC_BUCKETS) {
        INarray[max] = PID_INVALID;
        OUTarray[max++] = FALSE;
    }
    for (rpc_tries = 1;; rpc_tries++) {
        h = handle for node where vproc is executing;
        ret = vprocs_proc_flag(h, &v->vp_handle, flag, set_clear,
                               INarray, OUTarray, &execnode);
        if (successful vproc operation) {
            setuerror(ret);
            break;
        }
        ret = check_failure(v, rpc_tries, &first_try, &origin_node_contacted,
                            &h, execnode);
```

```
        if (ret == FAILURE) {
            VPROC_RELEASE(v, "GENERIC");
            return(FAILURE);
        }
    }

    used_buckets = 0;
    if (flag == SORPHAN_PGRP then this is a process group operation) {
        for (max = 0; max < MAX_RPC_BUCKETS &&
                        INarray[max] != PID_INVALID; max++) {
            if (OUTarray[max] == TRUE) {
                ORPHAN_CACHE_ADD_TO_LIST(INarray[max], v->pv_pgrp, retval);
                used_buckets++;
            }
        }
        cache_free(MAX_RPC_BUCKETS - used_buckets);
    }
    VPROC_RELEASE(v, "GENERIC");
```

2.2.1.4.6 Process Transparency Server

The server routines handle vproc operation requests that were initiated by client vproc operations from remote nodes. As with the client routines, the server routines follow a basic pattern.

Obtain the vproc given the PID from the handle passed in to the server and call the appropriate vproc operation to fulfill this request; if the return values states this is not the execution node, set the error value to PROC_NOT_HERE and update the execution node value passed to this server routine to the execution node value that this vproc thinks is correct.

The example below describes vprocs_set_flag(), the server routine which ultimately satisfies a remote request for calls to PVPOP_SET_FLAG().

```
int
vprocs_set_flag(
    handle_t        h,
    pt_vproch_t     *vh,
    int             flag,
    int             set_clear,
    pt_node_t       *execnode
    )
    v = vproc specified by pid vh->vh_pid;
    if (v is not NULL) {
        VPROC_MARK_TEMPORARY(v);
        setuerror(0);
```

```
        ret = (*v->vp_vops->vpop_set_flag)(v, flag, set_clear);
        if (ret == PROC_NOT_HERE)
            *execnode = v->pv_exec_node_hint;
        else
            ret = getuerror();
        VPROC_UNMARK_TEMPORARY(v);
        return(ret);
    }
    else {
        *execnode = UNKNOWN_NODE;
        return(PROC_NOT_HERE);
    }
```

The vproc is located using the process id specified within the pt_vproch_t argument to the server routine and the local operation is called to complete the operation. If the return value is PROC_NOT_HERE then update the execution node "guess" as to where the vproc is actually executing. Whether the client will use this new information depends upon the state of the client and how many attempts to locate the process have been made.

If the vproc is not found on this node, set the execution node guess to UNKNOWN_NODE and return PROC_NOT_HERE. The server node has no idea as to the location of the vproc and the client should go directly to the origin node.

If a server routine needs access to a vproc that may not be executing on this node, **VPROCLOC()** should be used. If the vproc cannot be found, the error value ENOSPC should be returned. The remainder of this section details those operations which deviate from this general format.

2.2.1.4.6.1 vprocs_proc_nice()

The nice value operation may be participating in an operation which involves lowest nice value for a process group. The flag value passed as an argument determines if this is part of a process group operation.

```
int
vprocs_proc_nice(
    handle_t        h,
    pt_vproch_t     *vh,
    nice_t          *nice,
    int             flag,
    pid_t           INarray[],
    bool_t          OUTarray[],
    pt_node_t       *execnode
    )
    v = vproc of pid specified by vh->vh_pid;
```

```
if (v is not NULL) {
    VPROC_HOLD(v, "GENERIC");                                          |
    setuerror(0);
    ret = (*v->vp_vops->vpop_proc_nice)(v, nice, flag, NOPGRPOPERATION);|
    if (ret == PROC_NOT_HERE)
        *execnode = v->pv_exec_node_hint;
    else
        ret = getuerror();
    VPROC_RELEASE(v, "GENERIC");                                       |
}
else {
    *execnode = UNKNOWN_NODE;
    return(PROC_NOT_HERE);
}

if (flag == NOPGRPOPERATION)
    return(ret);

if (ret != PROC_NOT_HERE) {
    low = *nice;
    for (max = 0; INarray[max] != INVALID_PID; max++) {
        m = LOCATE_VPROC_PID(INarray[max]);
        if (m && VPROC_EXEC_NODE(m)) {
            ret = (*m->vp_vops->vpop_proc_nice)(m, nice, flag,         |
                                                NOPGRPOPERATION);      |
        }
        if (ret != PROC_NOT_HERE) {
            OUTarray[max] = TRUE;
            low = MIN(low, *nice);
        }
        if (m)
            VPROC_RELEASE(m, "GENERIC");                               |
    }
    *nice = low;
}
return;
```

2.2.1.4.6.2 vprocs_sigproc()

The sigproc server may be required to initiate a sigproc() operation for various other vprocs provided this is part of a pgsignal() operation. This is determined from the flag value passed as an argument to the server routine. Note that the flag value is turned off when calling the local operations on this node; this would result in a recursive operation.

```
int
vprocs_sigproc(
    handle_t        h,
    pt_vproch_t     *vh,
    sig_t           sig,
    uid_t           effuid,                                          |
    uid_t           realuid,                                         |
    pid_t           sid,                                             |
    int             flag,
    pid_t           INarray[],
    bool_t          OUTarray[],
    pt_node_t       *execnode
    )
    v = vproc of pid specified by vh->vh_pid;
    if (v is not NULL) {
        VPROC_HOLD(v, "GENERIC");                                    |
        setuerror(0);
        ret = (*v->vp_vops->vpop_sigproc)(v, sig, effuid, realuid, sid, flag,|
                                          NOPGRPOPERATION);          |
        if (ret == PROC_NOT_HERE)
            *execnode = v->pv_exec_node_hint;
        else
            ret = getuerror();
        VPROC_RELEASE(v, "GENERIC");                                 |
    }
    else {
        *execnode = UNKNOWN_NODE;
        return(PROC_NOT_HERE);
    }

    if (flag == NOPGRPOPERATION)
        return(ret);

    if (ret != PROC_NOT_HERE) {
        for (max = 0; INarray[max] != INVALID_PID; max++) {
            setuerror(0);
            m = LOCATE_VPROC_PID(INarray[max]);
            if (m && VPROC_EXEC_NODE(m)) {
                ret = (*m->vp_vops->vpop_sigproc)(m, sig, effuid, realuid, sid,|
                                                  flag, NOPGRPOPERATION);|
            }
            if (ret != PROC_NOT_HERE)
                OUTarray[max] = TRUE;
```

```
            if (m)
                VPROC_RELEASE(m, "GENERIC");
        }
    }
    return;
```

2.2.1.4.6.3 vprocs_proc_flag()

The server routine must unmap the network mapped value passed by the client routine before invoking the local vproc operation. In addition, this may be part of a process group operation to set or clear the orphan process group flag.

```
int
vprocs_set_flag(
    handle_t        h,
    pt_vproch_t     *vh,
    int             flag,
    int             set_clear,
    pid_t           INarray[],
    bool_t          OUTarray[],
    pt_node_t       *execnode
    )
    v = vproc of pid specified by vh->vh_pid;
    if (v is not NULL) {
        VPROC_HOLD(v, "GENERIC");
        setuerror(0);
        ret = (*v->vp_pvproc->vp_vops->vpop_proc_flag)(v, flag, set_clear);
        if (ret == PROC_NOT_HERE)
            *execnode = v->pv_exec_node_hint;
        else
            ret = getuerror();
        VPROC_RELEASE(v, "GENERIC");
    }
    else {
        *execnode = UNKNOWN_NODE;
        return(PROC_NOT_HERE);
    }

    if (flag == NOPGRPOPERATION)
        return(ret);

    if (ret != PROC_NOT_HERE) {
        for (max = 0; INarray[max] != INVALID_PID; max++) {
            m = LOCATE_VPROC_PID(INarray[max]);
            if (m && VPROC_EXEC_NODE(m)) {
```

```
            ret = (*m->vp_pvproc->vp_vops->vpop_proc_flag)(m, flag,|
                                                           set_clear);|
        }
        if (ret != PROC_NOT_HERE)
            OUTarray[max] = TRUE;
        if (m)
            VPROC_RELEASE(m, "GENERIC");                          |
    }
}
return;
```

2.2.2 Signet Daemon

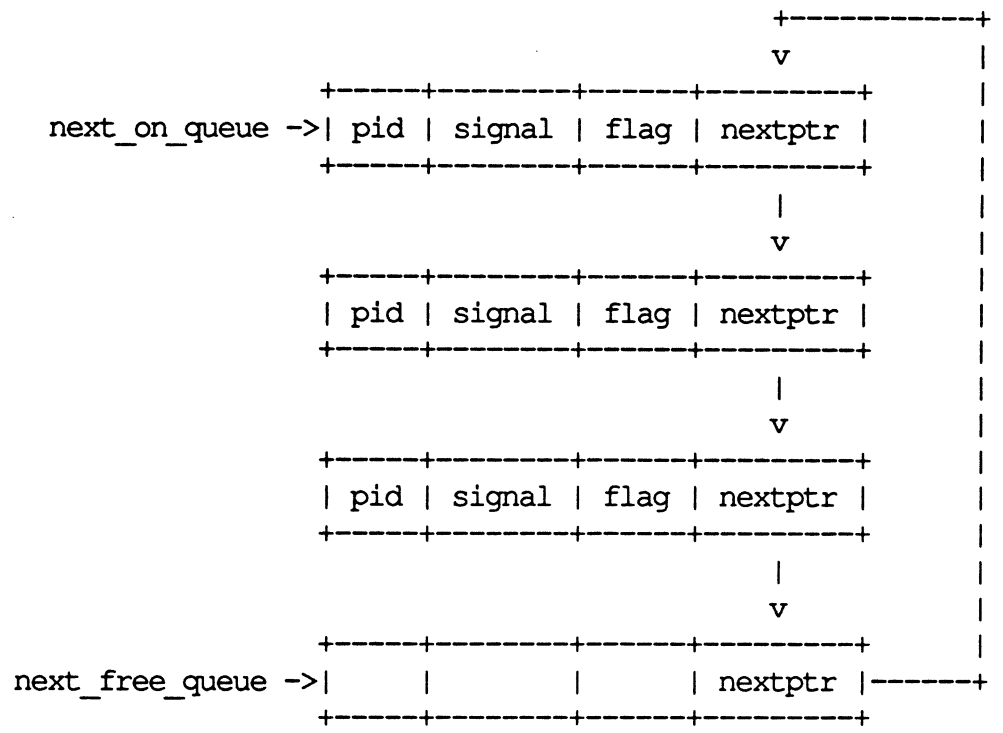
The signet daemon handles requests to send signals to remote processes and process groups when the kernel is at an interrupt state. The signet daemon consists of a kernel thread known as the *signet master* and an unspecified number of *signet slaves* which actually handle signal requests. The *signet master* has the responsibility of allocating elements for the signet queue and invoking *signet slaves* as they are needed to handle the signal requests. The signet daemon is part of remote processing and is started as a kernel thread as remote processing support for a node is installed.

In a single site UNIX implementation a signal sent to a process group would be handled by pgsignal(), which would traverse the list of processes in the group and send signals to each process. In a multiple site implementation however, processes within a process group may be executing on a remote site, which would require an RPC message to that site to signal the process. Rather than attempting to send a signal to a process group at the time of the interrupt, the signal request is put on a signal request queue. The signet daemon is awakened and satisfies as many requests as possible until the request queue is empty.

A signal request to a process group when the process group leader is a remote process will initially be handled by a client routine. The client routine is one of the remote operations which replace the local vproc operations when a process executes on a remote node. This client routine, which handles remote requests to signal a process group, will first determine if the kernel is at interrupt level, and if so the request will be put on the signal queue.

A signal request to a process group when the process group leader is a local process will be satisfied by going through the list of process group members, sending the signal to each process. The same functionality exists for sending signals to individual processes as for sending a signal to a process group. If any of the process group members are executing on a remote node, a client routine which has replaced the local signal routine in the vproc operations table will handle the signal request. If the client routine determines the kernel is executing at interrupt level the request will be added to the signal queue. Otherwise the client routine will make an RPC request to the appropriate server routine on the remote node to complete the signal request.

The signet queue is defined to be a number of elements linked together in a circular queue. The queue is depicted in the following manner:



The queue contains a sequence of elements, each of which may hold a value describing a process group or process which to signal and the specific signal to be sent. The flag field is used to specify whether the pid is for a process group or an individual process. <next_on_queue> points to the next queue element for which a signal will be delivered. <next_free_queue> points to the first available queue element to store a request for a signal to be sent.

When <next_on_queue> == <next_free_queue>, the queue is empty. This condition can occur only at startup when the linked circular queue is created, or when the signet daemon satisfies all requests on the queue.

If <next_free_queue> is the list element before <next_on_queue> and a signal request fills this element, the queue becomes full and <next_free_queue> can't be pointed to the next element on the list until more allocated elements are created for the list.

Rather than wait until this condition occurs, a high water mark is set to notify the *signet master* to add more queue elements. This value is set to 70% in this implementation, though the high water mark should be a tunable parameter that can be changed at run-time. The high water mark represents the percentage of the queue which contain valid requests to send signals. This value is based upon the total number of queue elements which comprise the signet queue, therefore as the queue grows larger, the number of queue elements which need to be full increases, but the percentage stays the same. The high water mark is implemented as a counter which contains an integer value of the percentage of queue elements available. As more

elements are added to the queue, the percentage specified is added to the high water mark. Requests put on the queue decrement the high water mark; requests handled by a *signet slave* increment the count. When the count goes to zero, the flag specifying more queue elements is set.

At this point additional queue elements are allocated for the signet queue by the *signet master* thread, the number of elements being equal to the initial amount of elements on the queue at system startup. The new elements are initialized and inserted into the circular queue at the pointer after <next_free_queue> and before <next_queue_element>. This allows free elements to be added to the queue without having the overhead of copying the current queue to a new queue object.

In addition to adding more elements on the queue, another *signet slave* must be invoked to help handle the expanded load. The extra *signet slave* will scan the list as described to satisfy requests. Determining when the queue is full will happen only when adding requests to the queue, an operation called by client routines when the kernel is at an interrupt level and a signal cannot be delivered at this time.

Duplicate signals being sent to the same process group are not put on the queue. Duplicate signal requests are checked before the signal request is added to the queue.

2.2.2.1 Assumptions

Sending signals to processes which execute on remote nodes are handled by the appropriate client routine which is referenced through the vproc operations table of the process. The client routine is responsible for determining if the kernel is executing at an interrupt level such that sending an RPC request (and subsequently waiting for a response) would not be a good idea at this time. In this situation the signal request is put on the signal queue to be handled by a regularly scheduled kernel thread (*signet slave*).

A mechanism must exist on the base system to determine whether or not the the signal request is being handled at interrupt time. There must also be a method to create additional kerel threads as signet slaves while the system is operating, or have a predetermined number of processes preallocated for the signet slaves as they are needed.

2.2.2.2 Detailed design

2.2.2.2.1 signet_master()

The *signet master* is a kernel thread which is invoked when remote processing is enabled on a system. Its responsibility is to determine if more queue elements are needed and to create and invoke *signet slaves* as they are needed.

If <signet_flag> contains the value SIGNET_MORE_ELEMENTS, the *signet master* must first expand the current circular list to accommodate more signal requests. A specified amount of queue elements are allocated, the fields <dlvrto> and <sig> are initialized to the value −1 and the pointer field <nextptr> is set to point at the next

element. The last element in this new queue has its <nextptr> field point at <next_on_queue>; <next_free_queue->nextptr> points at the beginning element of the newly allocated list of elements.

At this point, another *signet slave* thread is created to help handle the extra number of requests to send signals to remote processes.

This design removes the extra complexity of having the slave servers allocate extra elements. In addition, future design decisions such as performing an analysis of the signal queue can be implemented within the master server without impacting the current design.

2.2.2.2.2 signet_slave()

The signet_slave() is a kernel thread which handles requests to send signals to a process group or an individual process. This process runs at fixed priority PRI_SIGNETD, which allows it to handle signal requests in an expedient manner. The *signet slave* checks the signet_queue for signal requests. As many requests as possible will be serviced by the *signet slave* until no more requests are on the queue. As requests are handled, the high water mark is incremented. When there are no more requests, the *signet slave* goes to sleep. As requests are satisfied, the queue element fields are set to zero.

2.2.2.2.3 service_queue_request(struct signet_queue *elem)

Signal requests found on the signet queue by a *signet slave* are passed to the service_queue_request() function to handle the request. The <flag> field of each queue element specifies whether the signal specified is sent to a process group (**SIGPGRP**) or a process (**SIGPROC**). If the signal is for a process group, the function pgsignal() is invoked; if the signal is for a process group, the function pidsig() is invoked.

2.2.2.2.4 sig_add_queue(pid_t pid, int signo, int flag)

Signal requests which cannot be delivered at the current time are added to the signal queue using the routine sig_add_queue(). This function puts the signal request at the next free queue element and updates the queue pointers according to the rules specified by the signet queue data structure. The flag argument specifies whether the signal is to be sent to a process group (**SIGPGRP**) or to a single process (**SIGPROC**). Before putting the request on the queue, duplicate signals are checked. The list of valid queue elements are from <next_on_queue> up to <next_free_queue>. If the same request is found in these valid entries, the new request is thrown away. Note that the signal, process id, and flag value must all match since it is possible that a signal request is on the queue which specifies the pid of a process group leader and the request is to be sent to the entire process group, then a request is to be added which is for the same process, same signal, but we are really just sending a signal to that process.

After successfully adding a request to the queue, a *signet slave* is awakened. The high water mark is decremented only after the request is put on the queue. If the high water mark goes to zero, the circular queue is considered full and more queue elements must be added, along with another *signet slave* to handle the additional requests. This operation cannot be performed at interrupt time so <signet_flag> is set to SIGNET_MORE_ELEMENTS to notify the *signet master* that more elements must be added to the queue when it is scheduled to run.

2.2.2.2.5 signetstrtdisp()

Initializes a kernel thread to become a *signet master* or *signet slave*. Call creatp() and initp() to allocate a vproc object and initialize a physical process slot. Allocate enough memory for the queue elements and initialize the pointers to allow free elements in the signet queue. Set the priority of the thread to PRI_SIGNETD which will allow signals to be handled in a timely fashion.

2.2.2.2.6 Data Structure Definitions

```
#define SIG_Q_SIZE      100     /* initial number of queue elements */
#define PRI_SIGNETD     10      /* signet daemon priority value */

#define SIGNET_MORE_ELEMENTS    0x10

#define SIGPGRP         0x01    /* element specifies signal for process group */
#define SIGPROC         0x10    /* element specifies signal for process */

struct signet_queue
{
        pid_t  dlvrto;                  /* pgrp to receive signal */
        int    sig;                     /* the signal */
        int    flag;                    /* for process group or process */
        struct signet_queue *nextptr;   /* pointer to next queue element */
};

struct  signet_queue    *signetq,               /* the signet queue */
                        *next_on_queue,         /* next q element to service */
                        *next_free_queue,           /* next free q element */
int                     signetq_sleep = EVENT_NULL;  /* signet sleep queue */

long                    signet_flag;    /* specifies when to add q elements  */
long                    signet_hwm;     /* high water mark to add q elements */
```

2.2.2.2.7 Error Recovery Procedures

The queue initially contains a fixed number of elements. As the queue becomes full, a determinate number of elements is added to the queue and another *signet slave* is invoked.

2.2.3 Remote Processing Primitives

2.2.3.1 Introduction

FUSION provides several new system calls for remote processing and adds some functionality to some existing system calls. Rfork, rexec, and migrate are new, and fork, exec, and exit are enhanced in an upward compatible manner.

The design goals for the implementation of the remote processing primitives are

- to maintain exact POSIX semantics while allowing processes to move around the network,
- to allow these primitives to function as an add-on product together with vprocs,
- to allow processes to move around without excessive network delays,
- to optimize overall system and network performance,
- to optimize user perceived performance by allowing migrated process to resume execution at the earliest possible time,
- to take advantage of the AIX V3 architecture without sacrificing the ability to port this to other systems,
- and to keep code simple and avoid duplication of code.

This design interacts with the vproc design in that it deals with building and rebuilding the process relationship aspects of the vproc structure. The vproc lock mechanism is used to make sure that single system semantics are maintained. The pvproc structure is utilized for flags and other needed state information.

The following sections describe

1. Migrate. This includes a section on the use of vproc locking by migrate and a section on SIGMIGRATE. This is followed by a discussion of the clients and servers which make up the migrate design:
    a. Migrate Client
    b. Migrate Server
    c. Page Transmission Server
    d. Page Faults
    e. Page Fault Server
2. Exec. The Exec Client and Exec Server are described in terms of how they differ from the very similar Migrate Client and Migrate Server.
3. Rfork. A design using local fork and migrate of the child is described.

4. Exit. The changes to exit to perform necessary vproc operations are described. This is not really a remote processing primitive, but is described here in the context of the "quick-exit" used in the migrate and remote exec designs.

2.2.3.2 Migrate

The migration of a process is broken into two distinct phases. The first phase copies everything except the process data to the new node. At the completion of this phase, the new process is completely viable except that it will page fault as soon as it begins execution since no data pages have been created. In the second phase of migration, all of the data pages are sent over.

On systems where it is possible to enhance the memory management system to handle these page faults, the new process can begin execution immediately after the completion of the first phase. The migration is considered successful at that point, and the old copy of the process only exists as a "stub process" to transmit the data pages to the new process. Vproc operations can proceed during phase two, and the execution node of the process is now the destination node. No vproc operations on the migrated process will be handled by the stub process — the stub process is now considered to be distinct from the process which migrated.

The design for migrate has two sets of server code running on the destination node, one for each phase of the migration. The client code running on the source node talks to both of these servers. In addition, page faults by the new process on the destination node communicate to a server running on the source node.

The first server is the Migrate Server. It receives the request to begin the first phase of migration, and it does most of the work of moving the process to the new node. Although called a server, it can also act as a client with respect to other specific RPCs, such as when reopening files. There is no further protocol nesting and no protocol recursion in this design.

The second server is the Page Transmission Server (PTS). It receives pages of data asynchronously sent by the Migrate Client in the second phase of migration, and the PTS inserts those pages into the appropriate data segment. The Client talks to the PTS via an RPC pipe. The Client also can receive asynchronous requests to put specific pages into the pipe to the PTS as soon as possible.

An additional set of server code exists on the source node called the Page Fault Server. This receives requests from a process which previously migrated to get a specific data page. These requests are forwarded to the Migrate Client so that it may send the requested page to the PTS.

The following sections provide details of the different pieces of the migrate design. Figure 12 illustrates the interaction between these pieces. The highlights are:

1. The Migrate Client sends the Migrate RPC to the destination node.

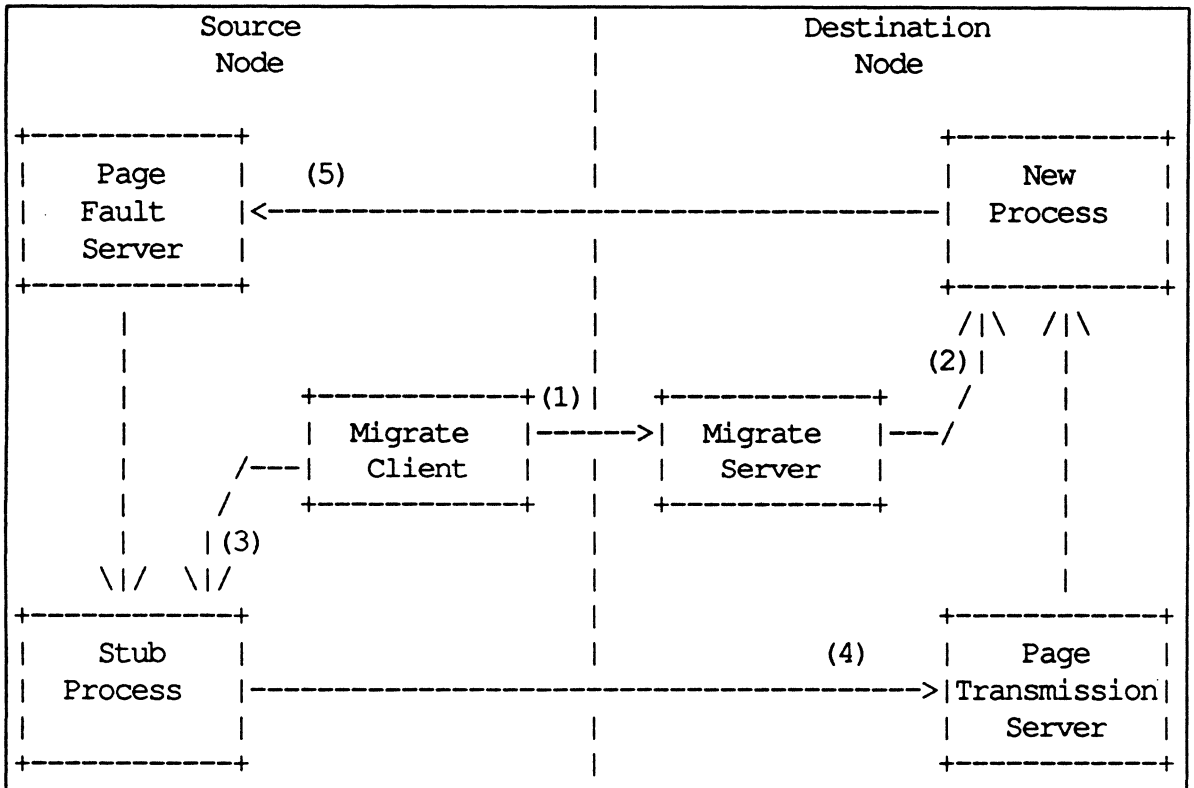


Figure 12. Interaction of clients and servers in the Migrate design

2. The Migrate Server creates the new process. That new process uses the data in the RPC to open files, set up the address space, etc. Phase one of the migration ends when the new process tells the Server to return back to the Client. The new process then can begin user mode execution.

3. The Migrate Client now begins phase two. It now acts as the stub process, not the original process.

4. The stub process sends data pages to the destination node of the migration. The Page Transmission Server on the destination node uses the machine dependent memory management system interfaces to put those pages into the memory segments of the new process.

5. If the new process page faults, it sends a request to the source node for the page. The Page Fault Server passes the request via the pvproc structure to the stub process. The Stub process then gives high priority to sending that page if it has not been sent already.

2.2.3.2.1 Vproc operations during migrate and remote exec

For a period of time during migrate and remote exec the process exists on both the source node and the destination node. Vproc operations acting during this time must either act on both incarnations of the process or the actions of the vproc operations on

one of the processes must be repeated on the other process once it is decided which process will become the real one (this isn't always the process on the destination node because of the possibility of error conditions). If repeating the actions later is chosen, it is necessary to also consider the implications of operations being performed out of order.

The fix is to say that no vproc operations can occur while the same process exists on two nodes. They all have to wait in line until it is known which process is going to become the one-and-only incarnation. By the time the migrate reaches phase two, this lock can be released. Since the speed of phase one is independent of the size of the process, the duration of the lock will not increase for big processes.

A pair of special locks are support by the vproc code to handle this for migrate and remote exec. These are the Migrate Shared Lock and the Migrate Exclusive Lock. The Migrate and Exec Servers obtain the Migrate Exclusive Lock because no vproc operation on the process can be valid while the process is still being created. The Migrate and Exec Clients obtain the Migrate Shared Lock. This only blocks operations which try to obtain the Migrate Exclusive Lock. Any vproc operation which tries to modify the process in any way must first obtain the Migrate Exclusive Lock. All other vproc operation must obtain the Migrate Shared Lock to make sure that a Server has not blocked them with its Exclusive Lock.

2.2.3.2.2 SIGMIGRATE

SIGMIGRATE is the mechanism used to all one process to cause another to migrate. An advantage of doing this as a signal is that this mechanism automatically picks up the capability to migrate whole process groups or just single processes. It also allows the process to ignore the signal or to catch the signal and perform special actions. Finally, it is one way to avoid the implementation problem of having to have one process access the user structure and data of another (possibly remote) process in order to move the data to another node.

First, we will present a quick review of signal delivery in a typical Unix system. When signals are delivered, all that the process sending the signal does is mark a bit in the process being signaled. It is the responsibility of the recipient to process the signal as appropriate. This is traditionally done in the routine psignal(). Psignal and routines it calls are responsible for causing the process to exit, create a core dump, stop, continue, or trap to a signal handling function.

The first step in implementing SIGMIGRATE is to put a hook into psignal (or equivalent) to call migrate() rather than core(), exit(), etc. Combined with the vproc changes which support having a node name passed with a signal, this is sufficient to handle most of what is needed for SIGMIGRATE. The remainder of this section discusses special considerations to make the implementation complete. These issues are

- arguments to signal handlers,
- preserving registers when SIGMIGRATE interrupts user mode execution,
- distinguishing between completed versus interrupted system calls,
- and dealing with partially completed interruptable system calls.

Signal handlers are defined to accept one argument which is the signal number. On many Unix systems, some or all signals are also defined to pass a second, signal specific argument. SIGFPE generated by arithmetic exceptions, for example, may be defined to pass an argument which specifies the type of exception. In **FUSION**, SIGMIGRATE signal handlers will be passed an argument indicating the node to which the process is supposed to migrate. The difficulty with this is that the node is specified by a string. We need to put the string somewhere in user space, and then we must pass the pointer to that string. The logical place to put the string is on the stack immediately before the arguments to the signal handler. These will look, in a sense, like a bunch of extra garbage bytes passed as arguments to the function. Since the function won't be declared to use all of those arguments, the extra bytes won't hurt anything. Those bytes will be popped off the stack in the same manner as any other function arguments. Depending on the calling convention on the machine, they will either be removed automatically by the return of the function (as with any other function return) or by the trampoline code which cleans up after the signal handler.

When SIGMIGRATE interrupts a process running user code, it is important that all registers are preserved. Normally, traps save registers on the kernel stack and restore registers in the process of returning to user mode. In the obvious implementation of just calling migrate() and letting it do the same thing it would do with a migrate system call, the destination node would end up setting the user registers the same way it does upon return from a successful system call. Machine dependent code is necessary to extract the registers from the kernel stack on the source node, and other machine dependent code is needed on the destination node to restore those registers when returning to user mode after a SIGMIGRATE which interrupts user mode while still setting the registers to indicate successful system call completion in the case of a migrate() system call.

When SIGMIGRATE interrupts an interruptable system call, it is necessary to restart the system call on the destination node. It is important for SIGMIGRATE to distinguish between interrupted systems calls and signal detection at the end of a completed system call. In the latter case it is important that the system call is not restarted. It is also important to not decide too early that a system call should be restarted. If another signal is processed effectively at the same time and that signal should interrupt the system call with restarting it, then SIGMIGRATE must not overrule that. The final decision to restart should be made only at the destination node after checking for all other signals.

The most difficult area of all is dealing with partially completed system calls, that is system calls which have already performed actions such as partial I/O and which cannot be restarted from scratch. The only alternative here is delay the SIGMIGRATE action until the system call completes. On systems which allow partially completed system calls to be interrupted, special care must be taken to not allow SIGMIGRATE to interrupt the system call. Special case code must be added to places, such as sleep() or equivalent, which which test for pending signals. Other signals should still be allowed to interrupt the system call, but SIGMIGRATE must not. This may require removing SIGMIGRATE from the list of pending signals and saving a flag indicating to the code which handles returning from system calls that there is a delayed SIGMIGRATE pending. **NEEDSWORK:** How do we do this without being very intrusive in the base code?

2.2.3.2.3 Migrate Client (source node)

The basic structure of the Migrate Client is outlined in Figure 13.

The first phase of migration is a simple RPC request for the Client. The RPC is called, and you wait for it to return.

In the second phase of migration the Client becomes a "stub process." The stub process is the same physical process as the original process. Only the process identification and process relationship information in the vproc structure is changed. This prevents confusion with the new physical process on the destination node.

The stub process creates an RPC pipe to pass pages over to the destination node. The server end of that pipe is the Page Transmission Server.

NEEDSWORK: The advantage of using an RPC pipe over individual requests is in the reduced overhead of not having to set up a large number of RPC requests. The disadvantage is that one server thread will be tied up for a long time (about a second for a several megabyte process) while the migration completes. While this design refers to the RPC pipe in a number of places, it is a trivial change to switch to having the stub process make separate RPC calls for each page to be transmitted.

When the stub process is sending the "random" chunks of data, it would be desirable but not strictly necessary to send recently referenced pages first since they are more likely to be needed soon. The stub process must keep track of which pages have been sent to avoid sending the same page multiple times. This is best handled by freeing the page and invalidating the page table entry as each page is sent. This also has the advantage of providing free memory as soon as possible on the source node. We may even need that memory ourselves for pages we need to bring in from the paging device. By the time phase two is ending, all of the data pages of the process in memory and on the page device will have been freed.

When the stub process is sending the very last page of data, it should include a flag in the RPC indicating that transmission is completed. This flag is used by the PTS to mark the target process as complete.

```
If not all data has arrived from a previous migrate wait for it to arrive (The
PV_MIG_INCOMPLETE flag in the pvproc is used for this).

Lock local vproc to prevent any changes (e.g., signals, setpriority, etc.)

send Migrate RPC — this includes all information necessary to create the process
except the contents of the writable data segments

wait for success/failure indication

Become a new, orphaned process, but hang around to send data.  Call this a "stub"
process.

Unlock local vproc.

if successful {
        open RPC pipe to the PTS on the destination node

        while not all data has been sent {
                if the "real" process on the destination node exited or exec'ed
                        break;
                else if any specific data has been requested
                        send it
                else
                        send "random" chunk of data
        }

        close RPC pipe
}

quick-exit (exit without the implications of orphaning, etc.)
```

Figure 13. Pseudo-code for the Migrate Client

Phase two ends with the stub process performing a "quick-exit." This is an abbreviated form of exit which does not do many of the vproc related operations such as SIGCLD notification to the parent and doesn't require creation of a zombie process. It does still do the vproc operations related to the fact that this is no longer an execution node of the process, such as notification of the process group leader. As explained in the section on exit, the vproc related pieces will be made separate so that this quick-exit can be separated from the true exit.

A final note concerns the case where a migration begins while the second phase of a previous migration is still occurring asynchronously. We could allow a situation arise

Create a new process, with a new proc, vproc, and pvproc. The local vproc should be created already locked to prevent any changes (e.g., signals, setpriority, etc.)

The following is executed in the new process:

Make yourself look like the original process (as with exec())

Open files, etc.

Use ld_getinfo() to create an appropriate address space.

Get the (shared) text segment.

Handle shared libraries:

```
If we can find the right library at the right origin
    Use it.
else {
    Create a new segment. This appears and behaves like a shared library
    segment, but is not really shared. If the process that we create this for
    fork()s, it will be shared, and so must be setup correctly. If the
    process is ptrace()d, a r/w private copy of this library segment will be
    constructed by ld_ptrace(), which must function correctly. When the
    process exits, the segment must be cleaned up in the same manner as
    private shared library segments are taken care of currently.
}
```

Set up data to be paged remotely.
Mark the pvproc as "incomplete" (the PTS will clear this when transmission of pages is complete).

Unlock local vproc (Actually, do this as early as possible. This must be done after we assume the identity of the new process. More important, this must be done after we are certain that the migrate will not fail. This unlock is, essentially, a point of no return. There shouldn't need to be very many checks, such as CPU type, node execution permission, and ability to reopen files, before we know the migrate will succeed).

Return to user mode execution.

Figure 14. Pseudo-code for the Migrate Server

whereby one stub process is sending data to another stub process which then send the data on to the latest execution node of the wandering process, but this has problems with complexity of the code and with risky failure conditions involving problems at

various points along the chain of nodes. Instead, the migrate client checks whether it is still in phase two of a previous migration before it begins phase one of the new migration. If there is still a previous phase two in progress, the client waits. This avoids the compounding the failure modes, but it means that successive migrates will not get the performance advantage of the parallelism of this two phase design.

2.2.3.2.4 Migrate Server (destination node)

The basic structure of the Migrate Server is outlined in Figure 14.

The first job of the Migrate Server is to create a new process. The vproc, pvproc, and proc structures must all be set up. This is essentially a local fork, except that the vproc is initialized with the PID of the original process, and it has its parent process, etc., set up to point to the migrating process's parent rather than to the server. |

As noted above in the vproc section, there is a small possibility that system or network failures may lead to having 2 processes with the same PID. The Migrate and Exec Servers will reduce the possible problems which this may cause by testing for an existing vproc which already claims to be on the execution node of the process. Migrate and remote exec will fail in this case.

The new process uses the information provided in the Migrate RPC to reopen all of its * files including current directory, reestablish any file locks, set up any signal handlers, set any remaining alarm time, set up its address space, open the text file and any shared libraries and map those into the address space, and set up vproc relationships such as parent, children, and process group information. The controlling tty | information (major, minor, channel, and node) is also passed. This is sufficient data | for the /dev/tty driver described in Section 2.1.1, Remote Devices, and for other | operations which need to know if the process has a controlling tty. If the migrate was | initiated by a migrate system call which requested that the mount context should be | preserved, then the old mount context is also established.

At this point, the new process is a perfectly valid copy of the original process except that it does not have any data pages yet. On systems where it is possible to extend the paging system, we will set up remote paging of these pages. The implementations memory management functions will be called to set this up, and hooks will be added to those functions to allow a new paging type.

The new process can then notify the Migrate Server that the migration was successful, and the Server can return control back to the Client. This completes phase one of the migration. The new process will continue execution, taking page faults as necessary which will be satisfied by the Page Fault Server and Page Transmission Server mechanism described below.

A flag in the pvproc, PV_MIG_INCOMPLETE is set in the new process to indicate that it is still waiting for pages to arrive from the Client. The Migrate Client will check this flag to make sure that a new migration cannot begin until the last one completes. As described below, exit, exec, and rexec will check this flag as well. If

they detect that the flag is on, they will notify the stub process that transmission of pages can cease. Fork, both local and remote, will wait until transmission is complete just as migrate does (on systems which support copy-on-write or copy-on-reference, local fork operation could copy the PV_MIG_INCOMPLETE flag into the child process, but this would create great complications when exec or exit try to determine if they can safely tell the stub process to cease transmission).

The concept of paging data remotely while the Page Transmission Server collects data asynchronously is really optional. If this is difficult to do in a particular port of FUSION, all that is needed is changing the point where we would otherwise set up remote paging to instead wait for the stub process to complete transmission of all pages. The code called to wait for the pages is the same code the Migrate Client calls before starting a new migrate. The Page Fault Server can then be dropped from that port and there is no need to deal with special page faults.

```
Put the data into a properly aligned page.

Get swap space to back it up (if that's what the memory management system
requires).

Attach and lock the segment, region, vseg, whatever.

Insert this page.

Detach and unlock the segment.

— section above is architecture dependent —
— section below is architecture independent —

If this is the last page for the process {
        clear the PV_MIG_INCOMPLETE flag in the process's pvproc.
        If the process is waiting for transmission to be complete
                wake it up
}

If there is a process waiting in a page fault on this page
        wake it up
```

Figure 15. Pseudo-code for the Page Transmission Server

2.2.3.2.5 Page Transmission Server (destination node)

This server is somewhat architecture dependent, but the basic outline is fairly general. It is described in Figure 15. The architecture dependent part deals with allocating the memory and associating that memory with the appropriate segment, region, vseg, page tables, etc. The architecture independent part deals with the communication with the

migrated process and the detection of the completion of phase two of the migration.

2.2.3.2.6 Page Faults

When a page fault is taken on a page to be brought in from a remote "stub" process, send a request to the stub via the Page Fault Server to send that page as soon as possible, then wait for the page to arrive. The stub process needs to be able to recognize a request for a page which it just managed to get off a short time before, and it can ignore such a request. This is explained under Migrate Client.

When the page arrives, it will be handled by the Page Transmission Server like any other page.

NEEDSWORK: How do we tell that server to wake us up when it has the page? Since page faults to disk with shared segments have a similar problem, this should not be a big obstacle.

```
Get a request for a particular "stub" process to send a particular page.

While the stub process has not yet picked up a previous page request
        sleep

Forward the page number to that process.
```

Figure 16. Pseudo-code for the Page Fault Server

2.2.3.2.7 Page Fault Server (source node)

The basic structure of the Page Fault Server is outlined in Figure 16.

The page number contained in the RPC message is forwarded to the stub process via the pvproc. The stub process will check for requests stored there whenever it needs to pick a page to send.

Normally, when a page fault has occurred, the process will have to sleep until it gets the requested page. It would, therefore, be impossible for the PFS to get a new page request before the process has received any previously requested page. It would then be safe for the PFS to blindly pass the page number to the stub process. This is not true, however, if the process consists of multiple threads and if the threads implementation allows some threads to continue after one thread is stopped by a page fault. To handle this case, it is necessary for the PFS to check to make sure that the stub process has picked up any previous page request before forwarding the new one.

2.2.3.3 Exec and Rexec

The structure of the Exec Client and Server is similar to that of the Migrate Client and Server, though there is no need for the Page Transmission Server.

2.2.3.3.1 Exec/Rexec Client (source node)

The basic structure of the Exec and Rexec Client is outlined in Figure 17. The Exec RPC is distinct from the Migrate RPC, although some the same actions are taken by

Select the destination node if not explicitly specified.

If not all data has arrived from a previous migrate (as indicated by the PV_MIG_INCOMPLETE flag) notify the stub process via the PTS that it can cease transmission.

Lock local vproc to prevent any changes (e.g., signals, setpriority, etc.)

Send Exec RPC — this includes all information necessary to create the process including the arguments and environment

Wait for success/failure indication

Mark the vproc as not being the execution node of the physical process. This may just mean changing identities as the stub process does in the Migrate Client.

Unlock local vproc.

Quick-exit (exit without the implications of orphaning, etc.)

Figure 17. Pseudo-code for the Exec/Rexec Client

the server.

If data from a previous migrate is still being transmitted, we can tell the stub process to cease transmission. This can be handled by sending a fake page fault RPC to the "stub" process which indicates that no more pages are needed. This will cause the stub to close its RPC pipe to the PTS.

The reasons for the locking are the same as described in the section on migrate. The same Shared and Exclusive Migrate Lock are used by exec.

See the Migrate Client and Exit sections for information regarding the quick-exit.

The main difference from the migrate design are that the RPC includes information on the exec arguments and there is no need for a Page Transmission Server and Page Fault Server to deal with bringing over data pages. The new process uses the exec arguments to start up a new program, and no data pages are needed from the old program at all.

The Exec Client is also responsible for selecting a node on which to execute. This is discussed in Section 3.2, Invocation Load Balancing.

2.2.3.3.2 Exec/Rexec Server (destination node)

The basic structure of the Exec and Rexec Server is outlined in Figure 18.

Create a new process, with a new proc, vproc, and pvproc. The local vproc should be created already locked to prevent any changes (e.g., signals, setpriority, etc.)

The following is executed in the new process:

Make yourself look like the original process.

Open files, etc.

Do the local exec.

Unlock local vproc (Actually, do this as early as possible. This must be done after we assume the identity of the new process. More important, this must be done after we are certain that the exec will not fail. This unlock is, essentially, a point of no return. There shouldn't need to be very many checks, such as node execution permission, before we know the exec will succeed).

Figure 18. Pseudo-code for the Exec/Rexec Server

The creation of the new process is the same as in the Migrate Server.

The new process performs the same actions as does migrate to reopen files. Like migrate, it sets up signal actions, though unlike migrate, standard exec semantics imply that all signal handlers will be replaced by SIG_DFL.

Once the process is set up, local exec is invoked to start running the new program. The new process lets the Exec Server know whether or not the exec was successful, and the Server returns the result to the Exec Client.

2.2.3.4 Rfork

The rfork function could be simulated fairly well at user level by doing a local fork and then having the child migrate to the new node. The problem with this implementation would be error handling. If the migrate fails, there would be no way to notify the parent and have the rfork fail. The correct behavior of rfork when a process cannot be created remotely is to return an appropriate errno to the parent. No child should be created, and the parent shouldn't get any SIGCLDs. This is difficult to arrange if the child really is created locally and it somehow has to go away without the parent thinking it was ever there. We desire that when a rfork fails our actual design should completely hide from the user whether it ever really created a child process. This can only be accomplished by putting some of the code into the kernel.

It is still possible to implement rfork as local fork followed by migrate provided that rfork is in the kernel. The basic idea is to have the parent wait on a semaphore until it is sure that the migration is complete. In case of failure, the child needs a mechanism by which it can pass the error number back to the parent. The semaphore will be implemented using sleep on the address of a pvproc field together with setting

```
Call fork.

In the parent:
        Sleep on a semaphore.

        If child successfully migrated, return PID.

        Otherwise, set errno and return -1.

In the child:
        Call migrate. Pass a flag indicating that the semaphore must be kicked upon
        completion. The semaphore will be kicked right before the stub process
        starts up at the end of a successful migration.

        If the migrate returns, it failed. Pass the errno to the parent, kick the
        semaphore, and call quick-exit.
```

Figure 19. Pseudo-code for Rfork

a flag in the pvproc. The child will turn on PV_RFORK_DONE in the parent's pvproc and wake up the parent. The errno will also be passed via the pvproc. The algorithm is outlined in Figure 19.

This strategy will perform as well as one which would use a special rfork RPC. Such a strategy would look very much like migrate, except that it would most likely do a procdup on the source node just before the stub process is created. The parent process could then go off and return to the user while the stub takes care of pushing the remaining data across. The strategy we have chosen here is essentially the same except that the procdup on the source node occurs at the very beginning and the parent sleeps until the stub process gets going. This way we get the same performance with greatly simplified code.

A side effect of this design is that the PID of the new process is selected by the source node, whereas in AIX 1.2 the PID is selected on the destination node. The fact that the origin node is remote is not expected to impact performance since it is necessary regardless to keep in contact with the parent process which is most likely to remain on that node.

2.2.3.5 Exit

The exit() function in **FUSION** requires only minor modification. The vproc needs to be cleaned up, and in the process the vprocs (possibly remote) of children and of the parent process need to be "adjusted" to correspond to this process having exited. Similar changes need to be made for process groups and sessions. Here we consider the vproc operations to be called, not the details of those operations. The details of the operations are left up to the vproc layer.

The common exit code for process termination, kexit(), is modified to call VPOP_PVPROC_REASSIGN_CHILDREN to orphan children and to send SIGCHLD to the parent process. If the parent process is ignoring SIGCHLD, this process will be orphaned to the INIT process to let it clean up.

In the case of quick-exit, all of the work to clean up the physical process is unchanged, but we do not want to orphan children or send SIGCHLD to the parent process. Quick-exit will set a flag in the vproc before it calls kexit to indicate that VPOP_PVPROC_REASSIGN_CHILDREN should not be called.

Other cleanup actions, such as those relating to remote files, devices, sockets, etc., also need to be dealt with by exit. The exit code itself just sees these cleanup actions as the normal close operations, and those cleanup actions will be covered in their respective designs. No changes need to be made to the exit code to make this work.

In addition, if this process had recently migrated and it still has a remote "stub" process sending data pages and a Page Transmission Server (PTS) receiving them, then the stub process needs to be notified that it no longer needs to transmit pages. As with exec, this can be handled by sending a fake page fault RPC to the "stub" process which indicates that no more pages are needed. This will cause the stub to close its RPC pipe to the PTS. *

2.2.4 Shell Enhancements

2.2.4.1 Overview

AIX V3 sh (Bourne shell) program source code and AIX/TCF sh source code are very similar. Both of them are derived from System V source code. To provide Process Transparency function in AIX V3, we mostly just need to port the AIX/TCF code to **FUSION**. Below we describe the changes to AIX V3 code which this would require.

The source code for ksh (Korn shell) in AIX V3 is fairly similar to the code for the Bourne shell. Most function names have been renamed, and a moderate number of new functions have been added to support ksh's additional functionality. The algorithms from sh, however, are mostly unaltered. While the exact lines of code used to add Process Transparency enhancements to sh will not work for ksh, the ideas will be very similar and the port will be a fairly straight forward translation.

The sh program is consist of 27 modules (.c files) and 11 include headers. They are: args.c, blok.c, cmd.c, ctype.c, defs.c, echo.c, error.c, expand.c, fault.c, func.c, hash.c, hashserv.c, io.c, macro.c, main.c msg.c, name.c, nls.c, print.c, pwd.c, service.c, setbrk.c, stak.c, string.c, test.c, word.c, xec.c, brkincr.h, ctype.h, defs.h, dup.h, hash.h, mac.h, mode.h, name.h, stak.h, sym.h, timeout.h. Those modules and headers in AIX V3 which have to be modified or require new functions and definitions are described in detail in the following sections.

2.2.4.2 Isolation of code changes

New code for Process Transparency will mostly appear in a new file, proctrans.c. Hooks will be added to the existing code which make calls to routines in proctrans.c. Slightly different versions of proctrans.c may be needed for the Bourne shell and the Korn shell. The code in proctrans.c will include ports of routines which were added to the AIX 1.2 version for TCF as well as new routines to replace blocks of code which TCF had inserted into existing routines.

2.2.4.3 Nodeinfo

One change will be made from the way Process Transparency was added for the AIX 1.2 version of TCF. In that implementation, many functions had a new argument added to point to a structure indicating the node on which execution would take place (if explicitly given by an "onsite" command). As the shell recursively descends the parse tree, it uses these arguments to keep track of the node. This approach has the disadvantage that it is highly intrusive to the code. Many functions have to be changed to accept this argument even though they do nothing with it but pass it on to other routines.

In **FUSION**, we will replace this with a global variable (though "static" to proctrans.c) which specifies the current node, combined with saving the previous node value on the local stack of any routine which needs to change the value. This will isolate the change and reduce by an order of magnitude the number of lines of code which need to be modified in the shell. The new global variable, "nodeinfo", indicates the node

on which the command currently being parsed should be executed. Nested commands are handled in this approach by having various routines save copies of nodeinfo on the stack before changing the node value and calling other routines.

In particular, the TON code (in xec.c in the existing TCF code, but in proctrans.c in FUSION) should save the existing "nodeinfo" on its stack. It should then put the new node information into "nodeinfo" and recursively call execute(). When execute() returns, the previous node data should be restored to "nodeinfo".

The execs() code in service.c is the one place where the "nodeinfo" is actually used. The call to execve() needs to be replaced with a call to rexecve(). Since execs() is a small routine, and it also needs to be changed to handle remote execution of shell scripts, it should be replaced entirely with a new version which will be in proctrans.c.

The shell uses longjmp in several places for error recovery. The above method for saving and restoring "nodeinfo" will not work is the recursive call to execute doesn't ever pass back through the TON code do to a longjmp. Each routine which calls setjmp must save away its own copy of "nodeinfo" on the local stack before the call to setjmp. When setjmp return via longjmp, the global "nodeinfo" variable needs to be refreshed with the value from the stack. To localize "nodeinfo" as a static variable in proctrans.c, this saving and restoring of "nodeinfo" will be performed by routines in that file. The code added to the caller of setjmp is just a call to the save routine, a call to the restore routine, and a declaration of the local variable.

2.2.4.4 Summary of changes

- proctrans.c

  new file with the bulk of new code
  functions to be pulled with only minor changes from existing TCF include

  - onsetup
  - restLocal
  - validSite
  - migrateMe
  - dosetpath
  - scType
  - scNum

  some new functions will be created to replace blocks of code added in TCF, allowing just a hook to be inserted for FUSION

- defs.c defs.h mode.h sym.h

  misc definitions added as needed

- cmd.c

  parse onnode command — call into proctrans.c

- fault.c

  handle SIGPWR and SIGMIGRATE

- func.c

  freetree(): add code to free nodes for onnode command
  prf(): add code to print nodes for onnode command
  (code is small enough that moving to proctrans.c is unnecessary)

- hashserv.c

  Don't use hash table with "onnode"
  Don't put commands executed via "onnode" into the hash table

- main.c

  Save nodeinfo before setjmp. Restore it after returning here via longjmps.

- msg.c

  new error messages
  new signals
  new builtins

- service.c

  execs must be able to remote exec shell scripts — no longjmp shortcut (this requires scan() to allocate extra space for an extra arg)

- xec.c

  add cases SYSMIG and SYSSPHERE for migrate and sphere builtins — put bulk of code in proctrans.c
  add case TON for onnode syntax — put bulk of code in proctrans.c

2.2.4.5 Detailed description of changes

This section describes the details of the major changes to be made. The code fragments illustrate how this code worked in AIX 1.2. Some small changes will be necessary in any particular port of this code to handle issues such as differing schemes for dealing with national languages.

2.2.4.5.1 Module xec.c

The function "execute" will be modified to handle new builtin commands, migrate, and setspath.. Builtin commands are handled by cases in the "switch (internal)" code. This, in turn, is in the code to handle TCOM parse tree nodes. For each of these builtins we will call new routines in proctrans.c. For example, we will add the following for the migrate builtin:

```
        case SYSMIG:
                sysmig(com);
                break;
```

In proctrans.c we will add code comparable to what appears in TCF in the SYSMIG case above. If no changes are made to that code, it would appear as:

```
sysmig(com)
char **com;
{
        char *a1;
        pid_t    pid;
        siteno_t rSite;
        PCchar   *pidStr;
        char     *end;

        if (sysconf(_SC_TCF) == -1)
            failed(notcf, "migrate");

        a1 = com[1];
        if (a1) {
            if (*a1 == '-') {
                if ((rSite = sfntonum(a1+1)) == -1)
                    rSite = strtol(a1+1, &end, 0);
                com++;  /* skip a1 */
            }
            else
                rSite = site((pid_t)0);

            if (*++com) /* any args?    */
                while (pidStr = *com++) {
                    pid = strtol(pidStr, &end, 0);
                    /*
                        Check to see if shell or shell's
                        process group are being migrated
                        If so, we always do a setlocal
                        to avoid confusing the user by
                        having the local on a different
                        site.
                     */
                     if (pid == shellpid ||
                          (pid < 0 && -pid == shellpgrp)
                          || pid == 0 || pid == -1)
```

```
                    {
                        migrateMe(rSite);
                        /*
                           If this is a process group
                           migration (or pid == 0 or
                           pid == -1), go on to send
                           the signal to the group.
                        */
                        if (pid > 0)
                           continue;
                    }
                    if (kill3(pid, SIGMIGRATE, rSite))
                         failed(nomigrate, pidStr);
                }
            else
                migrateMe(rSite);
        } else
            failed(badopt, migargs);
}
```

For the built-in command onnode,[3] we need to put a new type "TON" to the switch cases. The body of the case "TON" in existing TCF is provided as follow:

```
        case TON:
                onsetup(t,macro(onptr(t)->onsite->argval, sitet));
                execute(onptr(t)->ontre, exec_link, errorflg, t);
                restLocal(t);
                break;
```

This will be changed as described above to manipulate a new variable, "nodeinfo", rather than passing the node information as the extra argument, "t", to execute(). This code will be moved to proctrans.c.

Note: in the existing TCF code, the case body of the command parsing tree type "TFORK" was modified in order to improve error handling when rfork is used in place of fork. Although the code to handle EBADST, EPERM, ESITEDN1, and ESITEDN2 still exist in the existing TCF code, it is not necessary to port this change

3. Actually onnode is a reserved word. This allows greater syntactic flexibility. For example, "onnode nodename (command1; command2)" is allowed.

to FUSION since onnode will exclusively use rexec rather than rfork to move processes.

2.2.4.5.2 Module cmd.c

The module cmd.c just needs to have a new case, "ONSYM" added to the "switch(wdval)" statement. The function item() will have the declaration

```
int onnode_done = FALSE;
```

added. The body of the case will be:

```
case ONSYM:
        r = onsym();
        onnode_done = TRUE;
        break;
```

At the end of the switch, the call to word() needs to be changed to

```
if (! onnode_done)
        word();
```

The following code will be included in proctrans.c:

```
struct trenod *
onsym(){
        register struct onnod *t;
        char *ap;

        t = (struct onnod *)getstor(sizeof(struct onnod));

        /*
         * Fail if TCF isn't installed on this site.
         */
        if (sysconf(_SC_TCF) == -1)
                failed(notcf, "on");

        skipnl();
        if (wdval == EOFSYM)
                synbad();
        if ((int)wdval <= 0)
        {
                t->ontyp = TON;
                t->onflags = 0;
                t->onsitenum = (siteno_t) 0;
                t->onlocsav = (char *) 0;
                if (wdval != 0)
                {       /* site name is a reserved word */
                        extern struct sysnod reserved[];
                        short i;
```

```
                        for (i=0; reserved[i].sysval != wdval; i++);
                        ap = reserved[i].sysnam;
                } else
                        ap = wdarg->argval;
                if (cf(ap, "-v") == 0)
                {
                        if (skipnl())
                                break;
                        ap = wdarg->argval;
                        t->onflags |= ONV;
                }

                if (fndef || (wdval != 0))
                {
                        t->onsite = (struct argnod *)
                                        alloc(length(ap) + BYTESPERWORD);
                        if (t->onsite == (struct argnod *) 0)
                                return(0);
                        movstr(ap, t->onsite->argval );
                } else
                        t->onsite = wdarg;

                skipnl();
                t->ontre = item(TRUE);
        }
        /* NEEDSWORK:
        else
                synbad();
        ???? */
        return (struct trenod *) t;
}
```

2.2.4.5.3 Module service.c

The routine execs needs 2 types of changes made to it. The call to execve needs to be changed to rexecve using the node information saved in "nodeinfo". Also, execs has code in it to optimize the execution of shell scripts which do not begin with "#!". Rather than exec'ing a new shell, a longjmp is used to go back to main and start over. This trick will not work if the exec is supposed to go to another node. These changes significantly change the routine, so most of the code in the routine will end up being moved into proctrans.c.

2.2.4.5.4 Module hashserv.c

The context dependent nature of path names in **FUSION** (due to the change in mount context in the remote exec) means that hashing path names will not always work as

might be desired. As a result, a change needs to be made to avoid looking in the hash table if the current command is onnode (that is, if anything is saved in "hashinfo"). Another change needs to be made to avoid saving hash information related to commands executed with onnode.

2.2.4.5.5 Module func.c

In func.c, there are two subroutines which need to be changed, one named freetree, the other named prf. Both of these two subroutine have switch statements with cases for every type of parse tree node. We need to add case TON to consider if the command symbol is "onnode". The code to be added to freetree is:

```
case TON:
        free(onptr(t)->onsite);
        freetree(onptr(t)->ontre);
        break;
```

The code to be added to prf is:

```
case TON:
        prs_buff("onnode ");
        prs_buff(onptr(t)->onsite->argval);
        prc_buff(SP);
        prf(onptr(t)->ontre);
        break;
```

2.2.4.5.6 Header defs.h

For each new builtin, the header defs.h has to add one constant symbol such as:

```
# define        SYSMIG  34  /* current values have been used up to 33*/
```

Also add a definition for a new parse tree node type, TON.

2.2.4.5.7 Header mode.h

Header mode.h needs to add one new structure:

```
struct onnod
{
        int             ontyp;
        struct trenod   *ontre;
        struct argnod   *onsite;
        siteno_t        onsitenum;
        char            *onlocsav;
        int             onflags;
        char            *onmachtyp;
};
```

Meanwhile, a pointer to struct onnod

```
struct onnod  *_onptr;
```

needs to be added into a union structure

```
typedef union
{
        struct forknod  *_forkptr;
```

```
        struct comnod   *_comptr;
        struct fndnod   *_fndptr;
        struct parnod   *_parptr;
        struct ifnod    *_ifptr;
        struct whnod    *_whptr;
        struct fornod   *_forptr;
        struct lstnod   *_lstptr;
        struct blk      *_blkptr;
        struct namnod   *_namptr;
        char    *_bytptr;
} address;
```

We also need to add one macro

```
#define         onptr(x)        ((struct onnod *)(x))
```

2.2.4.5.8 Header sym.h

The symbol ONSYM needs to be added to the list of symbols for parsing, and the symbol ONV needs to be added for the onflags field of struct onnod.

2.3 Node Status Service

2.3.1 Description of the Product

This section covers interface and design features of the Node Status services. The Node Status Service provides the system with:

- information about nodes participating in the cell and what their uptime is.
- information about nodes not participating in the cell and how long they have been out of the cell.
- information about attributes of the nodes (load average, type of processor, version of operating system, scaling factor, preference level, cluster membership, PID ranges)
- a set of facilities for determining the least loaded node in the cell using CPU load, network traffic and other dynamic data.
- information about signed on users.
- provide storage and retrieval interface for /etc/mount information used by the cluster mount services

Node Status Services deal with all nodes within a cell. The most common client services will be modules, of the Process Transparency facility, requesting information about other nodes within the same cluster or group.

The Node Status Service will be robust so no single machine failure should bring the service down.

2.3.2 Description of Features

The Node Status Service provides a variety of features to support the services of FUSION. Commands and the parts of the kernel extension that require information about nodes in a cell or cluster obtain that information from the Node Status Service. The use of the Node Status Service reduces retry and other delays that would be experienced by the user or applications if they communicated directly with those nodes each time.

The Node User Service uses the Node Status Service to obtain the current set of signed on users. The Load Leveling function of the Cluster Environment uses the Node Status Service to determine which nodes are active and to obtain load average information for automatic node selection. The Cluster Mount Service makes extensive use of the Node Status Service to store information about file systems that are mounted in the cluster.

Cells and clusters may be large so users may wish to narrow the scope of nodes under consideration. For this reason, queries to the Node Status Service may be restricted to a set of nodes within the current user's "sphere of interest."

2.3.3 Components of the Service

The Node Status Services is comprised of two major components. These are the Local Node Status Service (LNSS) and the Group Node Status Service (GNSS). All client activity occurs between the local server and the client, application clients do not communicate with the group server. Group servers may communicate with each other and local servers.

2.3.3.1 The Local Node Status Server

The LNSS is responsible for collecting information about the local node, and for caching information about other nodes. It also pushes the information about the local node to the Group Node Status Server (GNSS) periodically so that information is available to other nodes.

The LNSS pushes static node information to its GNSS after LNSS is first initialized, and to a new GNSS if it does not have that information cached. The LNSS also periodically collects information such as the set of active users and CPU load data from the local node, and pushes it to the GNSS. The local component will be responsible for maintaining a cache of information about other nodes retrieved from the group server, thus eliminating redundant communications to other nodes. During network transients (start, stop, partition recover, etc.), the local component stores local user/system information for transmittal to the group server when it becomes available. The group server will maintain information for all nodes within its service area. All client requests for service will go through their LNSS. All requests made of this service default to a user's sphere of interest.

2.3.3.1.1 Configuring the Local Node Status Server

For each node, a node info file, in the FUSION cell wide name space, will represent membership in a group of the cell/group directory hierarchy. The basic node record, as would appear in node files, is shown below. This is a hard storage version of the node static data pushed to the Group Node Status Server. Data in /cell/group/node represents both tuneable and machine specific information about nodes in a named group. Node information is acquired by calling machine specific routines to determine machine configurations. For each supported platform calls specific to a particular hardware architecture must be made in order to determine the correct information to store and pass to the Group Server. Tuneable parameters are modified through user command interfaces, i.e. command line or system adminstration tools.

```
typedef struct static_node_data {
        int     node_number;
        char    node_name[MAXNAMELEN];
        char    cpu[20];
        char    coproc[20];
        char    os_version[20];
        char    os_vendor[20];
        float   load_scale;
```

```
        long    cluster_id;
        int     load_interval;
} nod_rec;
```

The LNSS does a lookup into the FUSION cell wide directory name space for a named file entry that denote the instance of the LNSS and associated group servers. These name files are used to help form the partial binding strings for contacting a group server. With the formed group string the LNSS calls the DCE CDS (Cell Directory Service) requesting a list of Group Node Status Server (GNSS) entries. The partial handle, which represents the preferred server, is used to make the first call to the rpcd running on the node of the group server. Complete handle information is returned from the call to the rpcd using the partial handle. If the rpcd replies that the service cannot be found, the LNSS then uses the next partial handle in the list to call another designated group server. Once connection is established with the remote server, the server may respond with NOT_A_SERVER and returns the handle to the active GNSS. On the first call to the GNSS, the LNSS passes the handle for the call back interface along with static node information. Dynamic loads data starts to be pushed from the LNSS to the GNSS at some tuneable interval, this essential represents a keepalive between the GNSS and LNSS interfaces.

2.3.3.1.2 Local Node Status and the Cluster Mount server

Passed back to the LNSS on the first call to the GNSS is the handle associated with the Group Cluster Mount Server (GCMS). The Cluster Mount Kernel Extension (CMKE) contains a system call allowing the LNSS to set the current Group Cluster Mount Server handle. At this time, and every time the LNSS contacts a new GNSS, the handle to the GCMS is passed into the CMKE by the LNSS using the system call setGCMShandle. The LNSS will then fork a Local Cluster Mount Server (LCMS) that will use the handle passed out through the getGCMShandle system call. This eliminates the need for a RPC interface between the LNSS and LCMS.

2.3.3.2 The Group Node Status Server

The Group Status Server (GNSS) stores information about other nodes within that GNSS's service area. Static node and dynamic load information is cached by the GNSS for each LNSS in the service area. Requests for information outside the service area are fulfilled and cached by the GNSS for clients within the service area. Along with the requested information, the requesting GNSS caches the handle used to contact the GNSS which provides information about nodes outside the services area. Group servers may be queried by the LNSS server components within its service area and by other group servers. All nodes that are candidates for group service operations will start a GNSS, only one GNSS in a designated group will respond as the group active server. Periodically the active group server will push group data to other non-active group servers. The GNSS also supports an interface between the GNSS and the Group Cluster Mount Server (GCMS). This interface permits the GCMS to find other participating group cluster mount server within the cluster. The GNSS will be responsible for maintaining all handle data for the GNSS and GCMS components. All

LNSS will be notified of a either a pending shutdown of non-preferred GNSS or a merge of GNSSs during partition recovery.

2.3.3.2.1 Configuration of the Group Node Status Server

Configuration of the group nodes FUSION services are maintained in the cell wide name space. Entries in the cell/named_group directory denote group servers, and for each group directory there are multiple group server configuration files. Files, in preferential order, indicate which nodes will operate as a GNSS. Lookup strings are constructed from GNSS1_nodename, GNSS2_nodename, etc. The first entry denotes the preferred GNSS, subsequent entries define what nodes will operate in the absence of the preferred group server.

2.3.3.2.2 Group Status Server Start-up

As with the Local Node Status Server, the GNSS will observe the same rules for contacting a compatible server through rpcds when initially started. Group servers will start on each node in which that node has been defined as a group server in the FUSION cell wide name space. If, during start-up, the preferred GNSS component starts up it will become the active GNSS.

There are three cases to consider in start-up:

- absence of any group servers — experienced during first time start-up
- presense of group servers — commonly encountered during multiple group service start-up
- intra-group partitioning — not likely to happen with good RPC routing protocols, but possible

2.3.3.2.2.1 Initial Start-up in a Cell

On the first invocation of a GNSS, the GNSS will be the only group server that other nodes or group servers can contact. Using the node configuration file, the GNSS determines how to represent the GNSS in the cell and creates an entry in the CDS directory that reflects this positioning. Each GNSS is responsible for maintaining its entry in the CDS directory namespace. Moving a GNSS server to another group or expanding groups is easily accomplished without shutting down any node. Since automatic handles require a partial portion of the handle to generate a binding handle, the first node to service the cell will register a single handle to the CDS in a well known directory. This directory containing the handle allows other servers to find When another group server starts it will find that group server. *

2.3.4 Service Area

The GNSS service area can span multiple clusters. The service area of a GNSS server does not share the domain of a cluster. As a practical matter, the service area is not restricted to a single cluster. In large clusters, service areas may be serviced by multiple group servers. GNSSs that can service a group will be reflected in an order of the GNSS4_Group list. This list contains all the primary candidates for GNSS

should a new GNSS need to be started.

The GNSS will contact other GNSSs to satisfy requests from clients whos SOI spans more then one group service area.

Each node is assigned to a GNSS's service area. Group may span serveral clusters. Nodes may dynamically join clusters and groups if the node can resign from the group and cluster.

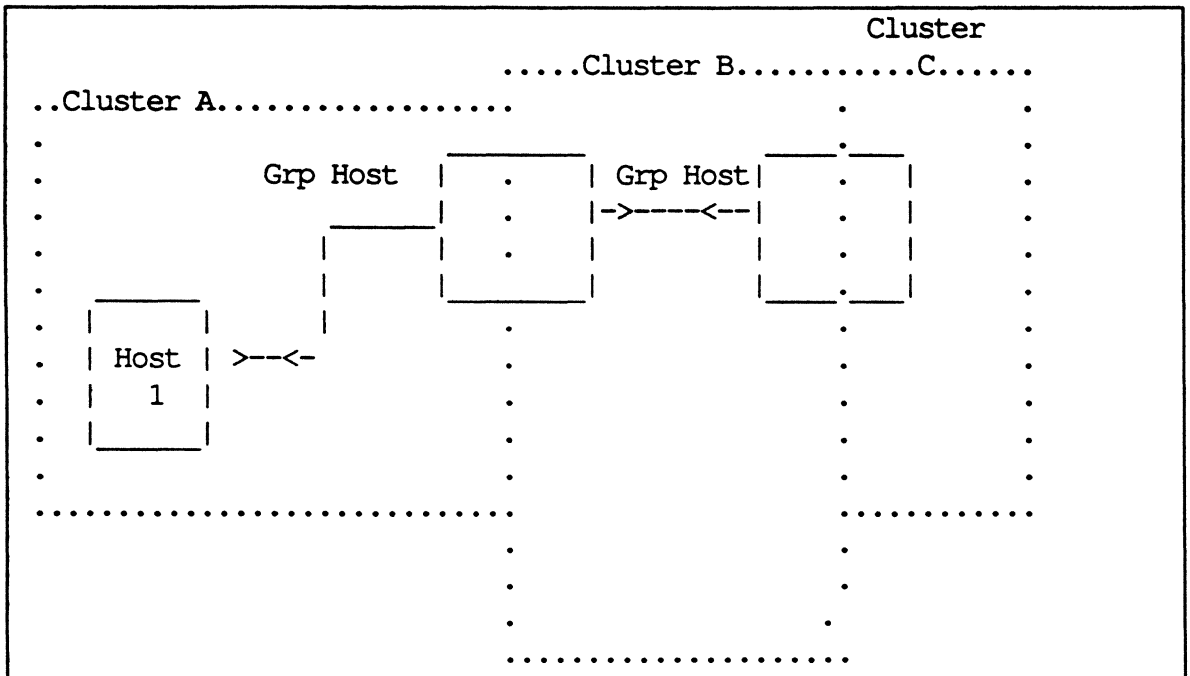


Figure 20. Example of GSS Service Area

An example is illustrated in Figure 20.

2.3.5 Service Initialization

2.3.6 GNSS and LNSS Protocols

The GNSS and LNSS share two distinct interfaces, the first interface supports data and control functions while the second interface provides GNSS to LNSS call-back mechanisms. Load, active user, and static node data are controlled through the IDL interface described in Programming Interfaces. A second interface, the GNSS to LNSS call-back interface allows the GNSS to determine if a node has actual gone down when the load interval timer has expired.

2.3.6.1 GNSS Election Protocol

All GNSS within a cell communcation through a peer to peer interface. This interface is used to pass tables representing the participating GNSSs within the cell. Tables are pushed to newly participating GNSSs and periodically pushed to non-servicing GNSS allowing these nodes to quickly find other servicing GNSSs in the cell when a GNSS

servicing a group becomes inaccessible. Inactive GNSS running within the group become active when a LNSS LNSS communication fails. Using the cached CDS registry entries the LNSS attempts to contact alternate group servers. When an inactive group server receives a message from a LNSS, the inactive group server then requests to become the group server for the group. Inactive group servers then wait for a message from an active group server indicating that should become the group server, or is informed that a group server has been selected and the handle for that group server. Active group servers query all other group servers to verify that the requesting server can indeed become the group server. This closely follows the two-phase commit paradigm. The newly elected group server continues to attempt contacting the designated group server tuneble interval. Once the designated group server is contacted the group server will attempt to resign to that group server.

2.3.6.2 Locating Other Group Servers

Everytime a GNSS starts it queries existing GNSSs by acquiring an automatic binding handle to registered GNSSs. On contacting a running GNSS the newly started GNSS determines if it must service a group of go into an inactive mode of operation. A merging GNSS then pushes its instance to all other GNSSs.

2.3.6.3 Partition Operation

Groups may operate in partition when communications between group servers outside a group cannot be established. When

2.3.6.4 Partition Recovery

The active group server periodically attempts to contact the primary group server. At contacting the primary group server, the parititioned server attempts to merge with the primary group server. An interface between the GNSS and GCMS permits the GCMS to determine if the group can be joined. If mount conflicts exist within the scope of the group, the group will continue to operate partitioned.

2.3.6.5 Election Deadlock Reconciliation

GNSS cell wide will support an election queue, each entry reflects the group membership and its priority within the group and the its status within the queue. States on the queue are REQ_GRP_SERVER, GRP_RESPONE, GRP_ELECTED. When another group server pushes a request to this queue, the node processing the requests determines if the incoming request should be pushed on the queue or if a response GRP_ELECTED should be returned.

2.3.7 GNSS and GCMS Protocol

The GNSS is resposible for starting the GCMS, the GCMS inherits a global structure that contains the uuid of the bound gnss_2_gcms interface. The GCMS then passes a handle to the gcms_2_gnss interface. Then GNSS then pushes the instance of the GCMS servicing cluster x to all GNSS within the scope of the cell.

2.3.8 Data Services Interface

2.3.8.1 Internal/External Data Services

The Node Status Server will be responsible for storing several well defined data elements for each type of service provided. These representation are done through a data service interface. Since IDL code must be recompiled to support new versions of a data interface, a facility in the NSS provides node client applications with the ability to define and register new data storage requirements. An example of this would be when a load leveling interface pushing data to a LNSS requires a new type or element. Typically this requires writing a new IDL interface for the LNSS and GNSS component and providing this new interface to all nodes in the cell. At the same time the NSS uses the data services, client node applications may use this data interface to define external runtime data storage elements.

Data interfaces, defined as DCE IDL code, can be converted from NIDL to schema definitions.

2.3.8.1.1 Internal Data Interfaces

Services used by the LNSS may register IDL independent versions of interfaces, this avoids tightly coupling the IDL interface to data. Version control can then be the responsibility of the host application when calling a registered interface.

2.3.8.1.1.1 Data Type Support

For each data type represented in the client storage requirements a type value is used to expand data transmitted or received from storage entities.

```
/*
 * Data types supported for storage and retrieval. Types are provided in the
 * case data must be sent across the wire via IDL. A non-negative value
 * returned or passed into the data structure indicates the length of a
 * character array.
 */
#define SCH_TYPE_INT_2          -1
#define SCH_TYPE_INT_4          -2
#define SCH_TYPE_FLOAT_4        -3
#define SCH_TYPE_FLOAT_8        -4
#define SCH_TYPE_UCHAR                  -5
#define SCH_TYPE_CHAR                   -6
#define SCH_TYPE_UUID           -7
```

2.3.8.1.1.2 Internal Data Representation

Database interfaces support different types of hashing/b-tree specifications registered by each client interface. Support for registering schemas and the associated data interfaces are detailed in the following pseudo-code. This is representative pseudo code of the NIDL database interface.

```
/*
 * The hash and keys supported for building tables and trees.
 */
#define SCH_KEY_HASH    1 /* Hash the key and perform hash operations */|
#define SCH_KEY_DUP     2 /* Used to form btree keys, allowing dup keys */|
#define SCH_KEY_UNIQ    3 /* Do not allow duplicate keys */            |
#define SCH_KEY_NONE    4 /* No keys for this field */                 |

/*
 * The match_list_t structure defines the query data syntax for performing a
 * query on a data set. These are operators supported for query.       |
 */

#define Q_AND   0x01
#define Q_OR    0x02
#define Q_XOR   0x04
#define Q_NOT   0x08
#define Q_GT    0x10
#define Q_LT    0x20
#define Q_EQ    0x40

type struct match_list {                                              |
                    char                        *m_field;             |
                    int                         m_operator;           |
                    char                        *m_comparator         |
} match_list_t;                                                       |

typedef struct db_query {
        char    *q_fldname; /* Matches field listed in registered schema */
        char    *q_value;       /* Value to perform logical comparison */
        syntax_t q_flag;          /* Logical comparator */
        struct db_query *q_next;
} db_query_t;

/*
 * The following schema data structure allows service interfaces to register
 * individual database storage types for other services.
 */
typedef struct db_schema {
        char            *sc_fldname;
        int             sc_type;
        int             sc_key;
        struct db_schema *sc_next;
```

```
} db_schema_t;
```

One of the reserved fields could be used to denote versioning of a data interface, it then falls back on the client to deal with modifications to versions.

```
/*
 * This structure provides interfaces for both the initialization and
 * lookup functions. The dt_schema provides the pointer to the specific
 * data registered under the name dt_tblname. Clients need not understand
 * the data specifics of an interface at runtime, a query may be performed
 * to retrieve a data dictionary from the database server. The database
 * interface will provide data representation for the client.
 * The reserved entries are for future enhancements such as:
 * mapping storage strategies, service selection and typing, versioning,
 * or replication masks. These are only suggestions to their use.
 */
typedef struct db_dict {
        char            *dt_tblname;
        unsigned32      reserved_1;
        unsigned32      reserved_2;
        unsigned32      reserved_3;
        unsigned32      reserved_4;
        db_schema_t     *dt_schema;
} db_dict_t;

/*
 * data wrapper used to pass variable length data of differing data types
 * between the caller and the server.
 */
typedef struct db_data {
        void            *db_data;
        int             db_size;
        struct db_data  *db_next;
} rec_data_t;

/*
 * On requests that contain multiple match sets, a list of these matches|
 * will be returned to the caller. Storage interfaces will not need to
 * understand the data representation in order for the requester
 * to unwrap the response.
 */
typedef struct sel_data {
```

```
        rec_data_t      *sel_data;
        struct sel_data *sel_next;
} sel_data_t;

/*
 * It would be useful to have a virtual/file/rpc descriptor that allows
 * access to local virtualized data, back store to disk, and distributed
 * storage and retrieval.
 */
typedef unsigned int    db_descriptor_t;
typedef unsigned int    status_t;
```

2.3.8.1.2 External Database Service Interfaces

Client routines are provided calls to register, lookup, store, and retrieve information managed by the LNSS/GNSS.

```
/*
 * Initialize database based on the service specified requirements. db_init
 * verify that the request is well formed and create an empty slot in the
 * data dictionary. A descriptor will point to the actual storage module.
 * Subsequent calls using the other database functions will modify data
 * stored at the descriptor.
 */
db_descriptor_t
db_init(db_dict_t *)

/*                                                                    *
 * Perform lookups into the data dictionary based on the visible name entry.
 * This may be modified later to include other selective lookup criteria.
 * On successful completion of the lookup, db_lookup returns the actual
 * descriptor to the data record definition.
 */
db_descriptor_t
db_lookup(db_dict_t *)

/*
 * Used in conjunction with db_lookup, allows the request service to attach
 * to a registered database.
 */
db_descriptor_t
db_attach(db_dict *)

/*
```

```
 * Retrieves data from a specified database using keys and hashes derived
 * from the registered schema.
 */
(sel_data_t *)
db_retrieve(db_descriptor_t, db_query_t *)

/*                                                                    *
 * Adds an entry, rec_data_t, into the database pointed to by db_descriptor_t
 */
status_t
db_add(db_descriptor_t, rec_data_t *)

/*                                                                    *
 * Deletes entries from the database pointed to by db_descriptor_t matching
 * the record or records supplied.                                    |
 */
status_t
db_del(db_descriptor_t, rec_data_t *)

/*                                                                    |
 * Modifies entries in the database pointed to by db_descriptor_t using|
 * the recored or records supplied                                    |
 */                                                                   |
status_t                                                              |
db_mod(db_descriptor_t, rec_data_t *)                                 |

/*
 * Since the server will allocate match_list data pointed to by sel_data_t
 * the client application must clean-up the space when it is no longer needed.
 */
status_t
db_ml_release(db_descriptor_t, sel_data_t *)


/*
 * Before exiting from a database service routine, this call is provided
 * to release all resources allocated to the caller.
 */
status_t
db_release(db_descriptor_t)
```

Before data may be passed to or from an interface the data will be wrapped in a variable length data structure to insure compact storage. The IDL interface will still | perform marshaling but the client must be able to wrap and unwrap the opaque data.

2.3.9 Data Maintained by the Service

Node Number

An integral identifier that identifies the node uniquely within the cell.

Node Name

The "commonly used" node name.

CPUType

The CPU type for the current CPU.

Coproc

name of coprocessor

OSversion

The OS version of the node.

OSvendor/type

Name of vendor and OS type

Scaling Factor A scaling factor. This scaling factor is used to determine the least loaded node. A higher scaling factor indicates a more powerful node. This could correspond to a relative MIPs rating.

Node State

Whether the node is currently up, down, or not a good load level candidate. An arbitrary node is not permitted to change a different node's state. If a node attempts to do this, then the Node Status Server will itself attempt to determine if the node status and return it to the calling node. A node may mark itself as being down. The values for this field are NODE_UP, NODE_DOWN, NODE_TRANSITIVE.

Node Time Stat

The time the node came up (if the node's current status is up) or the time the node went down (if the node's current status is down). P. Node In Cluster

Whether the node is currently participating in it's cluster.

Load Average

LoadAverage1Min LoadAverage5Min LoadAverage15Min *

The one-minute, five-minute, and fifteen minute load averages for the node. These measure the average length of the queue of runnable processes at a given node, averaged over periods of 1 minute, 5 minutes and 15 minutes.

MaxLoadTime

The maximum time expected between load updates from a node.

MaxLoadDelta

Load which causes reporting to be done ahead of schedule.

2.3.9.1 Programming Interfaces

2.3.9.1.1 Node Status

The Node Status Server will publish several interfaces each of which is specific for the type of data which is being stored and or requested. Interfaces presently supported are prefixed with their respective category name i.e. node_, user_, load_, and clmnt_ (cluster mount).

The return value for all of these interfaces is uniformly defined.

```
GNSS_PENDING                    server is trying to declare itself
GNSS_PENDING_MERGE              server conversion in process
NOT_A_SERVER                    this binding handle is not useful
OK                              successful completion
```

2.3.9.1.1.1 Node Interface

node_

this interface supports the organized storage and retrieval of information about nodes. This information is referred to as static data and consists of information about each node which will change infrequently if at all.

Terms/structs

node_rec

a fixed structure as described below

```
typedef unsigned32 node_id              /* A unique node identifier */

typedef struct node_rec {
        unsigned32      version         /* internal version number */
        node_id_t       node_id;        /* Unique node number */
        uuid_t          callback_hnd;   /* Handle used by GNSS */
        unsigned8       node_name[20];  /* fixed length string */
        unsigned8       cpu[20];        /* fixed length string */
        unsigned8       coproc[20];     /* fixed length string */
        unsigned8       os_version[20]; /* fixed length string */
        unsigned8       os_vendor[20];  /* fixed length string */
        float           scale;          /* an arbitrary numbering scheme */
        unsigned32      time_started;   /* when the system came up, or went */
                                        /* down */
        unsigned8       stats;          /* nodestat, datastat, grpstat, */
                                        /* cluststat) */
        unsigned32      load_interval_1; /* load intervals, can be modified */
```

```
        unsigned32      load_interval_2; /* for start-up */
        unsigned32      load_interval_3;
};

definitions for the status member of the node_info structure          |

#define NODEUP          0x01        /* node is up */                  |
#define NODEX           0x02        /* node is inaccessible */        |
#define DATAVALID       0x04        /* data is current */             |
#define CLUSTIN         0x08        /* I am in a cluster */           |
#define GRPMINE         0x10        /* I am servicing this node */    |
```

The following variables are used in the RPC calls and have the same meaning.

handle:	The RPC handle associated with this operation
rec_index:	The largest used index of records
records:	A variable length array of node_recs
return_status:	Return status of the server
rec_id:	A single node identification number
a_index:	The largest used index of the a_list array
a_list:	A variable length string array
soi:	A UUID (or something like it) which represents the users sphere of interest. It expands to a list of node_id's. If it is present the next two fields are ignored.
max_index:	The largest index supported in the record array
cursor:	An I/O parameter used by the server which is processing this request. Must be initialized to zero on first call. If the request cannot be satisfied in a single call, this value must be returned unchanged with each subsequent call.
in_record:	A single node_rec used as a match template. Since 0 is not valid for data in any field, it is a wild card. The rec_id must be set to 0. Matching will occur only on valid data.
out_rec_index:	The largest index used which contains valid data.
out_records:	A variable length of nodes_recs. No partial rec_ids will be returned.

The node_ functions are:

node_add	If node does not exist, adds to Node database, else modifies data.
node_add_alist	Add/modify the attribute list of a given node.
node_del	Delete a node from the Node servers database.
node_get_data	Retrieves node_recs for specified nodes.
node_get_alist	Retrieves a single nodes attribute list.
node_match_data	Retrieves node_recs of nodes which match the supplied data.

node_add - If node does not exist, adds to Node database, else modifies data.

This service will add node(s) to the Node database if it is not already there. If the node already exists in the database, its data is replaced by the new data.

```
[idempotent]
int     node_add(
        handle_t [in]   handle;
        signed32 [in]   rec_index;
        node_rec [in, last_is(rec_index)]       records[];
        status_t [out]  return_status;
        )
```

node_add_alist - Adds/modifies an attribute list to a node

This service will add a nodes attribute list to the database, or modify an existing one.

```
[idempotent]
int     node_add_alist(
        handle_t        [in]    handle;
        node_id         [in]    rec_id
        signed32        [in]    a_index;
        unsigned8       [in, last_is(a_index)]  a_list[];
        status_t        [out]   return_status;
        )
```

node_del - Delete a node(s) from the Host servers database.

This service will delete a node(s) record(s) from the Node Status Server database.

```
[idempotent]
int     node_del(
        handle_t        [in]    handle;
        signed32        [in]    rec_index;
        node_id         [in, last_is(rec_index)]        rec_ids[];
        status_t        [out]   return_status;
        )
```

node_get_data - This function retrieves node_recs for the specified nodes

This function retrieves node_recs for the specified nodes

```
[idempotent]
int     node_get_data(
        handle_t        [in]    handle;
        soi_t           [in]    soi;
        signed32        [in]    rec_index;
        node_id         [in, last_is(rec_index)]        rec_ids[]
        signed32        [in]    max_index;
        signed32        [in, out]       g_rec_index;                    |
        node_rec        [in, out, last_is(g_rec_index), max_is(g_max_index)]|
                        records[];
        unsigned32      [in,out]        cursor
        status_t        [out]   return_status;
        )
```

node_get_alist - Retrieves an attribute list

This service will retrieve a single nodes attributes list.

```
[idempotent]
        int     node_get_alist(
        handle_t        [in]    handle;
        node_id         [in]    rec_id
        signed32        [in]    max_index;
        signed32        [in, out]       a_index
        unsigned8       [in, out, last_is(a_index), max_is(max_index]
                        a_list[];
```

```
        status_t        [out]   return_status;
        )
```

node_match_data - Retrieves record ids of records which contain the supplied field types

This function retrieves record ids of records which contain the supplied field types

```
[idempotent]
        int     node_match_data(
        handle_t        [in]    handle;
        soi_t           [in]    soi;
        signed32        [in]    rec_index;
        node_id         [in, last_is(rec_index)]        rec_ids[]
        node_rec        [in]    in_record;
        signed32        [in]    max_index;
        signed32        [in, out]       out_rec_index;
        node_rec        [in, out, last_is(out_rec_index),
                        max_is(max_index)]      out_records[];
        unsigned32      [in,out]        cursor
        status_t        [out]   return_status;
        )
```

node_match_alist - Retrieves node_recs of records which contain alist supersets of elements supplied by the caller

This function retrieves node_recs of records which contain at least the elements supplied by the caller in the stored alist.

```
[idempotent]
int     node_match_data(
        handle_t        [in]    handle;
        soi_t           [in]    soi;
        signed32        [in]    rec_index;
        node_id         [in, last_is(rec_index)]        rec_ids[]
        signed32        [in]    a_index;
        unsigned8       [in, last_is(a_index)]  a_list[];
        signed32        [in]    max_index;
        signed32        [in, out]       out_rec_index;
        node_rec        [in, out, last_is(out_rec_index), max_is(max_index)]
```

```
                    out_records[];
        unsigned32  [in,out]        cursor
        status_t    [out]   return_status;
        )
```

node_nomatch_alist - Retrieves node_recs of records which do not contain any of the elements supplied by the caller in their stored alists.

This function retrieves node_recs of records which do not contain a single element in stored a_lists which match those supplied by the caller.

```
[idempotent]
int     node_nomatch_data(
        handle_t        [in]    handle;
        soi_t           [in]    soi;
        signed32        [in]    rec_index;
        node_id         [in, last_is(rec_index)]        rec_ids[]
        signed32        [in]    a_index;
        unsigned8       [in, last_is(a_index)]  a_list[];
        signed32        [in]    max_index;
        signed32        [in, out]       out_rec_index;
        node_rec        [in, out, last_is(out_rec_index),
                        max_is(max_index)]      out_records[];
        unsigned32      [in,out]        cursor
        status_t        [out]   return_status;
        )
```

2.3.9.1.1.2 Load Interface

load_
this interface supports the organized storage and retrieval of CPU load information for nodes. The storage method is a non-shared database.

Terms/structs

```
typedef struct load_rec{
        unsigned32      last_time_sec;  /* seconds at time of last update */*
        unsigned32      last_time_msec; /* microsecs at time of last update */
        unsigned32      avail_mem;              /* available memory in pages */|
        unsigned32      page_rate;              /* current paging rate in pages */|
        unsigned32      io_rate;                /* current I/O rate */|
```

```
        unsigned32      swap_space              /* available virtual swap space
        unsigned32      undefined_1;    /* User defined fields*/
        unsigned32      undefined_2;
        unsigned32      undefined_3;
}

typedef struct lrecord{
        node_id         rec_id;
        load_rec        load;
}

typedef struct lm_record{
        node_id         rec_id;
        unsigned8       node_name[20]
}

typedef struct load_max{
        node_id         rec_id;
        set_load        max_load;
}

typedef struct set_load{
        unsigned32      max_time_sec;   /* microseconds between updates */
        unsigned32      max_time_msec;  /* microseconds between updates */
        float           max_load_delta; /* delta which causes load reporting */
}
```

lrec: An lrecord which is used to update the servers load information

max_load: All the data necessary to change the time and delta of reporting loads.

num_recs: The index of the last record in the following records list

rec_ids: The node_ids of the target nodes.

record: A single node_rec used as a match template. Since 0 is not valid data in any field, it is a wild card. The rec_id must be set to 0. Matching will occur only on valid data.

max_lrecs: The maximum number of lrec spaces available in the buffer provided.

num_lrecs: The index of the number of lrecords returned in the output buffer. i.e. number of lrecords-1.

max_lm_recs: The maximum index of lm_records(s) which fit in the buffer provided. This is used to determine how many nodes will be returned. i.e. if 3, it will give back the three least loaded nodes in that order.

lm_recs_used: The index of the number of lm_records returned in the output buffer. i.e. number of lrecords-1.

lm_recs: The output buffer. No partial records will be returned.

The load service functions are:

load_put	Updates node load information
load_get_data	Retrieves load information about specified nodes(s)
load_set_max	Sets the max reporting time and max load delta
load_get_max	Gets the max reporting time and max load delta
load_fastnode	Uses one set of fields to select records and returns a specific node identifier

load_put Updates node load information

This service will update load information in the node database for the specified node.

```
[idempotent]
int     load_put(
        handle_t        [in]    handle;
        lrecord         [in]    lrec;
        status_t        [out]   return_status;
        )
```

load_set_max Sets the max reporting time, and max load delta

This service changes the max reporting time and the max reporting delta.

```
[idempotent]
int     load_set_max(
        handle_t        [in]    handle;
```

```
        load_max        [in]    max_load;
        status_t        [out]   return_status;
        )
```

load_get_max — Gets the max reporting time, and max load delta

This service changes the max reporting time and the max reporting delta.

```
[idempotent]
int     load_get_max(
        handle_t        [in]    handle;
        load_max        [in, out]       max_load;
        status_t        [out]   return_status;
        )
```

load_get_data — Retrieves load information about specified nodes(s)

This function retrieves load information about specified nodes

```
[idempotent]
int     load_get_data(
        handle_t        [in]    handle;
        soi_t           [in]    soi;
        unsigned32      [in]    num_recs;
        node_id         [in, last_is(num_recs)] rec_ids[]
        node_rec        record;
        unsigned32      [in]    max_lrecs;
        unsigned32      [in, out]       num_lrecs;
        lrecord         [in, out, last_is(num_lrecs), max_is(max_lrecs)]
                        lrecs[];
        unsigned32      [in,out]        cursor
        status_t        [out]   return_status;
        )
```

load_fastnode — Uses one set of fields to select records and returns an lm_record(s)

This function processes the load information about nodes which match the specifications set in the field_ids parameter and returns an lm_record(s).

```
[idempotent]
```

```
int     load_fastnode(
        handle_t        [in]    handle;
        soi_t           [in]    soi;
        unsigned32      [in]    num_recs;
        node_id         [in, last_is(num_recs)] rec_ids[]
        node_rec        [in]    record
        unsigned32      [in]    max_lm_recs;
        unsigned32      [in, out]       lm_recs_used;
        lm_record       [in, out, last_is(lm_recs_used), max_is(max_recs)]
                                lm_recs[];
        status_t        [out]   return_status;
        )
```

2.3.9.1.2 Active User

The C library entries documented on the getutent manual page (i.e. getutent(), getutid(), getutline(), pututline(), setutent(), and endutent()) are inappropriate for accessing the active user database. This is because the output of any particular call depends upon what calls have previously been done (e.g. a getutid() call that immediately follows a getutline() call would be very difficult to emulate).

New functions are provided that interface with the active user server. It is inappropriate for the calls that manipulate the active user data base to replace the getutent calls. This is because the getutent calls are still needed to:

- add utmp entries when there is no need to manipulate the active user data base. Such as, adding utmp entries prior to the user logging in, adding utmp entries for date changes and node status changes, etc.
- search through the utmp entries either when only information about the local node is desired or if the active user server is inaccessible.
- provide source and binary compatibility for applications not yet coded to interface with the active user service.

Therefore, these functions will remain unaltered. Additional routines for accessing the active user service are given below.

2.3.9.2 addactuser Routine

```
#include <utmp.h>
int addactuser(utmp)
struct utmp *utmp;
```

This function adds the specified utmp entry utmp to the active user database.

2.3.9.3 delactuser Routine

```
#include <utmp.h>
int delactuser(utmp)
struct utmp *utmp;
```

This function deletes the specified utmp entry utmp from the active user database.

2.3.9.4 selactuser Routine

```
#include <utmp.h>

int selactuser(user,line,nodes,token)
char *user;
char *line;
char *nodes[];
caddr_t *token;
```

This function is used to select utmp records from the active user database. The utmp records are actually returned by subsequent getactuser() calls.

The selection criteria for selecting utmp records are as follows:

- If user is a NULL pointer, then utmp entries with any user names are selected. Otherwise, only utmp entries where ut_user is equal to user are selected.
- If line is a NULL pointer, then utmp entries with any device names are selected. Otherwise, only utmp entries where ut_line is equal to line are selected.
- If nodes is a NULL pointer, then utmp entries for all nodes within the current user's sphere of interest are selected. Otherwise, nodes is an array of pointers to node names, with the array terminated by a NULL pointer. Only utmp entries where ut_node is equal to one of the node names specified by nodes are selected.
-

The "token" parameter is a generic pointer that is an output parameter of selactuser. The memory to which it points is filled in by selactuser, and "token" is passed in to subsequent getactuser calls.

2.3.9.5 getactuser Routine

```
#include <utmp.h>
struct utmp *getactuser(token)
caddr_t *token;
```

This function is used to fetch a utmp entry that was selected by a prior call to selactuser(). When all utmp entries have been fetched, this function returns the constant EOF. The "token" argument is a generic pointer that is both an input and an output parameter. The initial value of token is filled in by selactuser, and the value is changed on subsequent getactuser calls.

2.3.9.6 purgeactuser Routine

```
int purgeactuser()
```

This function purges the active user data base of all users for the current node (i.e. deletes all users for the current node from the active user database).

2.3.9.7 Internal Interfaces and Design

The fields ut_user, ut_lnode, and ut_line are key fields in the Active User database.

The library programming interfaces translate their requests into RPC calls. If a server cannot (for some reason) be contacted by the library routines, the transactions that cannot be transmitted are stored in a local file. They are replayed once the server is again available. This is done as part of the addactuser() and delactuser() library routines and is not externalized to the caller of these routines.

2.3.9.7.1 RPC Interface

The user functions are:

user_add	If the user does not exist, adds to User database, else add given fields to the user.
user_del	Deletes the specified user from the user database.
user_get_data	Retrieves specified field data from supplied records.
node_rel_data	Uses one set of fields to select records and returns a different set of fields.

2.3.9.7.2 user_add

If the user does not exist, adds to User database, else add given fields to the user.

This service will add a use to the User database if it is not already there. If the user already exists in the database, the fields are added to that user record. If a field exists, its data is replaced by the new data.
[idempotent]

```
void user_add(
     handle_t    [in] handle;
     node_id     [in] rec_id;
     struct utmp [in] utmp_entry;
     status_t    [out] return_status;
     )
```

handle	The RPC handle associated with this operation.
rec_id	A single node identification number.
utmp_entry	A single users utmp data.
return_status	Returned status from the server.

2.3.9.7.3 user_del

Deletes the specified user from the user database.

This service will delete a user record from the user database.
[idempotent]

```
void user_del(
     handle_t  [in] handle;
     node_id   [in] rec_id;
     unsigned8 [in] user_name[20];
     status_t  [out] return_status;
     )
```

handle The RPC handle associated with this operation.

rec_id A single node identification number.

user_name A buffer containing the user name.

return_status Returned status from the server.

2.3.9.7.4 user_get_data

Retrieves specified field data from supplied records.

This function retrieves specified field data from supplied records.
[idempotent]

```
void user_get_data(
     handle_t    [in] handle;
     node_id     [in] rec_ids[];
     unsigned32  [in] name_size;
     unsigned8   [in, length_is(name_size)]  user_names[];
     unsigned32  [in] num_fields;
     field_id    [in, length_is(num_fields)] field_ids[];
     unsigned32  [in] max_out_bytes;
     unsigned32  [out] bytes_used;
     unsigned8   [out] buffer[];
     unsigned32  [in,out] cursor;
     status_t    [out] return_status;
     )
```

handle The RPC handle associated with this operation.

rec_ids The UUIDs of the kernel-servers of the target nodes. The UUID of this server generally identifies the node.

name_size The size of the input name buffer.

user_names A buffer containing a field. Each field consists of a UUID which represents USERNAME, a length and the name string.

num_fields The number of fields in the following field list.

field_id An array of UUIDs structures containing the UUIDs of fields to be extracted from the nodes identified by the rec_ids provided.

max_out_bytes	The maximum number of bytes available in the buffer provided.
bytes_used	The number of bytes of the output buffer which contain valid data.
buffer	The output buffer. No partial records will be returned, the format of the output buffer is as follows: {record_id (a uuid, fixed length) number of fields (an unsigned16) field_id(a uuid, fixed length) length of data (an unsigned 16) data (string of bytes)} repeat.....
cursor	An I/O parameter used by the server which is processing this request. Must be initialized to zero on first call. If the request cannot be satisfied in a single call, this value must be returned unchanged with each subsequent call.
return_status	Returned status from the server.

2.3.9.7.5 Sphere of Interest

Associated with each process is a list of nodes called the Sphere of Interest (SOI). The SOI is purely advisory. The presence or absence of a node from the list in no way allows or prohibits any kind of access to the node. The list is used to narrow down the list of nodes which would be considered under certain default conditions (remote "who", automatic node selection, etc). The SOI is arbitrarily changeable by any process, though the most common case would be for a login shell to set the Sphere and leave it alone after that.

All requests made of this service are defaulted to a user's sphere of interest. The sphere of interest is really just a hint about the part of the network which is interesting to that user. A users sphere of interest should be some logical collection of nodes to which he has access privileges and with which he has routine correspondence. Of course, reasonable choices should be made based on network topology, frequency of access, etc.

2.3.9.7.5.1 User Interface

As described in the FUSION Functional Specification, a builtin command in the shells would allow the user to add or delete nodes to/from the SOI.

Some programs will check for the existence of a file called "~/.sphere.PROGNAME". If it is found, its contents will be used in place of the SOI for that command. This could be used to allow different SOIs to be used for who and ps, for example.

2.3.9.7.5.2 System Call Interface

The getsphere and setsphere system calls would manipulate the SOI. Nodes will be specified using fully qualified names, allowing any accessible node to be in the SOI.

2.3.9.7.5.3 Kernel Storage and Caching

The setsphere system call will save away the SOI in a cache in kernel memory. A UUID will be associated with the cache entry, and the UUID will be saved by the process. If an identical list already exists in the kernel, the existing cache entry and UUID will be used (and a use count is incremented). Otherwise, our NSS will be asked to supply a UUID (Soi_To_UUID()).

When a process forks, the use count on the shared SOI will be incremented. On exit, the count will be decremented, and the SOI will be freed if it is now unused. Similarly, the setsphere call will decrement the use count on the OLD SOI (if any) and free if necessary.

If the SOI groups are not expanded yet, the list will be small enough that there wouldn't be much penalty if we didn't do caching.

2.3.9.7.5.4 NSS Storage and Caching

The NSS will also cache SOIs (in this case, in user memory). When the Soi_To_UUID request is received, the NSS will check the cache and create a new entry and UUID only if necessary.

By having each kernel check with its NSS when executing setsphere, we increase the chance that two kernels talking to each other will already know the UUIDs for each other's SOIs (in the most common case, two kernels talking to each other will have the same GNSS). If we did not do this, remote exec would always have to send the whole list just to compare and see if it is already known on the new node.

2.3.9.7.5.5 Process Movement

Send the UUID in place of the SOI. Only send the whole list if the UUID is unknown on the destination node. If the SOI groups are not expanded yet, the list will be small enough that there wouldn't be much penalty if we had to send the whole list all the time.

2.3.9.7.5.6 Auto Node Selection

Choose from among the intersection of the set of cluster members, SOI, the set of nodes on which the user is allowed to execute, and the set of nodes on which the program can execute.

2.3.9.7.5.7 Fast

The "fast" and "fastnode" programs default to choosing from among the intersection of the set of cluster members, nodes of the "same type", Xperm, and SOI. Options may be desirable to leave out cluster membership or SOI from consideration (as existing TCF already has an option to leave out the sameness criterion).

2.3.9.7.5.8 Who, loads, etc.

The basic interface to NSS would be to send the UUID for the SOI to request that the search by limited to these nodes.

If the NSS didn't know that UUID (perhaps you switched NSSs at some time), it needs to be able to send a response requesting that you download that SOI list.

2.3.9.7.5.9 RPCs

The following RPCs will be needed to directly manipulate Spheres of Interest:

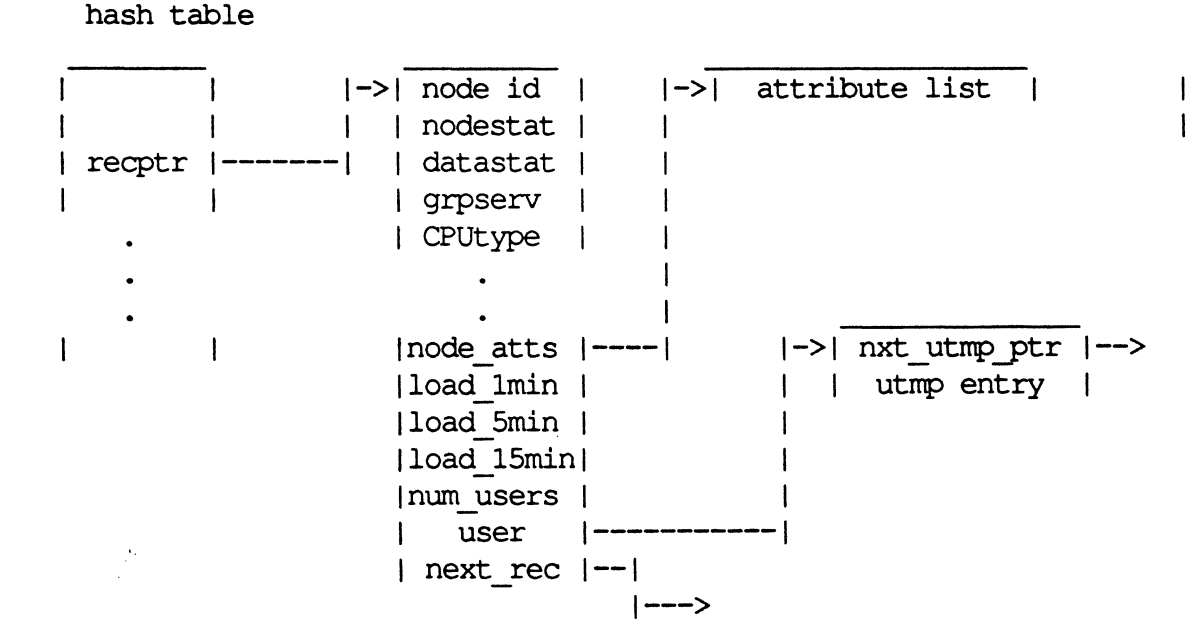
Soi_To_UUID: sent to NSS to obtain a UUID for an SOI. NSS returns either a new UUID or one obtained from its SOI cache.

Get_Soi: sent by the destination node of a process movement operation (migrate, exec, rexec, rfork) to the source node. The UUID obtained from the Process_Movement RPC is passed and the complete SOI is returned. The caller is then expected to add the new SOI/UUID to its cache. This call can also be used by the NSS when a request is sent to it with the UUID of an unknown SOI.

The UUID for the SOI will also be passed to the NSS for any operation such as "fast" or auto-node-selection if the SOI is to be considered among the selection criteria.

2.3.9.8 Internal Design

In its first incarnation each server will maintain a hash table (the key of which is the node identifier) which points to a fixed size record. This record contains several pieces of quick reference information and pointers to all other data. See diagram:




```
typedef struct node_info {
        node_rec        node_data;      /* node up info */
        set_load        max_load;       /* max reporting data */
        load_rec        load_data;      /* All the load info */
```

```
        unsigned16      num_users;      /* the number of users */
                                        /* logged in to a single node */
        node_var_rec    *node_atts;     /* pointer to variable length */
                                        /* string data */
        n_utmp          *users;         /* pointer to n_utmp entry */
        node_info       *next_node;     /* pointer to next */
};
```

All data will be aged and discarded based on tuneable parameters.

PLEASE NOTE: In the design for determining stale data (and node inaccessibility) there is a requirement that an extended time function exist. The present time function only reports in seconds which is (probably) not fine enough granularity.

A thread will run in each server which will periodically collect load information about its node. (NOTE: a group server acts as a local server for the node on which it is being run). The frequency for this collection will be a tuneable parameter specified in a file (indicated in microseconds). It will cache this information in the local server's database and transmit it to the group server only when the Max time or Max delta have been exceeded for the load. At such time it will transmit all the load data, i.e. load_1min, load_5min, load_15min, to the group server.

A companion thread will run in all servers which has two purposes 1) it checks to see if the load data it has is out of date. It decides if this is the case by examining the time of the last update it received (for each node it has information on) and comparing that against the Max time interval allowed. 2) it checks the status of nodes it has marked as suspect/down.

The actions taken with respect to these two conditions differs significantly if the server is functioning as a group or local server.

If it is functioning as a local server: and its load data is out of date, it marks its data as stale. When a subsequent request is made for data about this node the local server queries the group server for the node information and updates its database.

If a node is marked suspect/down, it periodically (at intervals based on a timer/request mechanism) requests the status of those nodes from the group server.

If it is functioning as a group server and the load data is state, it also checks to see if it is the primary for the node. If it is, it attempts to solicit new data from the node. If it receives no response it marks the node as suspect.

When it is functioning as a group server and a node is marked suspect/down, if it receives no message within a finite interval (from any other group server announcing it has taken over the role of being that nodes server), it marks the node as down. It then waits to be updated by a message from another group server about that node.

2.3.9.9 Command Interfaces

2.3.9.9.1 Node Status

2.3.9.9.1.1 nodeup

The nodeup command is the command that is used to inform the Host Server of the current node's status. It also transmits the non-volatile information about its node. The previously existing values for the node are discarded. This command is typically invoked at boot time. A sample invocation of the command is:

```
nodeup CPUtype=i386 NodeName=FredStation
       NodeNumber=7864 NodeIsUp=yes
       NodeInCluster=yes ScalingFactor=2.0
```

2.3.9.9.1.2 fastnode, fast

The fastnode command selects and then displays the node name of the least loaded node that has a compatible CPU type as the current node. fastnode has the following options:

Broadens the selection beyond those nodes with the same CPU type as the current node. attributes Limits the selection to nodes with specific attributes. Attribute specification is done in a manner similar to the nodeup command.

In all cases, the selection is limited to nodes within the current user's sphere of interest (and of course, to nodes that are up). The fast command is similar to fastnode, except that a specified command is run upon the least loaded node.

```
For example:    fast cc foo.c
```

2.3.9.9.1.3 loads

The loads command displays average load information about all nodes in the current user's sphere of interest.

The loads command has the following options:

- Display load average information only about the current node.
- Display load average information about a specific node.
- Limit the selection to nodes with specific attributes.

Attribute specification is similar to the nodeup command.

2.3.9.9.1.4 loadserver

The loadserver command allows the user to specify a (load) delta for retransmission of load information, as well as a max time interval should no change occur within the normal reporting times.

2.3.9.9.1.5 node

The node command is used to report on information about nodes within a sphere of interest (often a cluster). This information is obtained from the Host Service. The command line options and various names by which the command is known control

which nodes are reported and what information about them is displayed. Any of the data recorded for nodes will be obtainable with this command.

2.3.9.9.2 Active User

To take advantage of the extended service offered by this technology, small modifications to a few system utilities are required. Exiting binaries will work as they do now.

Modifications are available in the following areas:

- The auinit command has been added. Auinit notifies the active user server to purge its database of all users for the current node each time the system goes up or down. It is invoked by /etc/init in a startup script.
- Commands which modify the local /etc/utmp file have been changed to also notify the active user server of the changes. This includes the commands login, rlogind, and telnetd.
- Extended versions of commands allowing the current user to work within his/her sphere of interest are included. These commands are altered to get information from the active user server rather than the /etc/utmp file. Some users may not desire the extensions made to these commands. To minimize the impact on existing users, the modified versions of the standard commands will be delivered with the names comsatx, fingerx, rwhox, talkx,usersx, wallx, whox, and writex. In this way the system administrator may choose to install these commands to replace the original commands. Individual users can use aliases to access these commands instead of the normal versions. Options are also provided for these commands to only get information about users logged onto the current node.

2.3.9.9.2.1 login, rlogind, and telnetd

There are no externally visible changes required for these commands. However, these commands do modify existing /etc/utmp entries. Therefore, these commands have been changed to use the new library routines that are described in the following section of this document when modifying existing utmp entries.

2.3.9.9.2.2 comsatx

The comsatx command is different from the comsat command in the following way:

- Comsatx notifies the user's login sessions at all nodes within the comsatx command's sphere of interest that mail has arrived. Not just the current node.

2.3.9.9.2.3 fingerx

The fingerx command is different from the finger command in the following ways:

- When fingerx lists the idle time and login time for a particular user name, the idle times and login times are listed for login sessions of the user at all nodes within the current user's sphere of interest. In addition, the node the user is logged in at is appended to the login information.

- A –L option has been added to the command. When the –L option is used, only login sessions on the current node are listed.

2.3.9.9.2.4 rwhox

There are no differences between the user interface of rwho and rwhox. The only difference is that rwhox does not use the rwhod interface to get at the list of remote users. Rwhox uses the active user service to get a list of users within the current user's sphere of interest.

2.3.9.9.2.5 talkx

The talkx command is different from the talk command in the following ways:

- When a user name is specified with no node name (i.e. the constructs user@node or node!user are not used), then the user to be contacted is searched for at all nodes within the current user's sphere of interest.
- An additional node name parameter, to be used with the line parameter has been added to the command. When the line parameter is used without the nodename parameter, then the user to be contacted is searched for on device line within all the nodes in the current user's sphere of interest. If the nodename parameter is specified, the user is searched for only at the nodename specified.
- If the nodename parameter is not specified, a user name has been specified with no node name, the user being written to is logged in at more than one node within the current user's sphere of interest, and one of the user's logins is on the current node, all the nodes the user is logged in at are displayed, but the user is contacted at the current node.
- If the node name parameter is not specified, a user name has been specified with no node name, the user being written to is logged in at more than one node within the current user's sphere of interest, and none of the user's logins is on the current node, all the nodes the user is logged in at are displayed, and the user is contacted at an arbitrary node within the list.

2.3.9.9.2.6 usersx

The usersx command is different from the users command in the following ways:

- If the usersx command is used with no options, it lists the login name of all users logged in on nodes within the current user's sphere of interest. If this form of the command is used, the node name the user is logged in on is prepended to the output of each line.
- A –L option has been added to the command. When the –L option is used, only users logged in to the current node are listed.
- A –s nodename option has been added to the command. When this option is used, only users logged in to the specified node name are listed.

2.3.9.9.2.7 wallx

The wallx command is different from the wall command in the following ways:

- By default, the wallx command sends the specified message to all users logged in at all nodes within the current user's sphere of interest.
- A –L option has been added to the command. When the –L option is used, the specified message is sent only to users logged onto the current user's node.

2.3.9.9.2.8 whox

The whox command is different from the who command in the following ways:

- By default, the whox command displays information about users logged in at all the nodes within the current user's sphere of interest.
- In addition to printing the normal whox output, the node name the user is logged in to is also printed.
- A –L option has been added to the command. When the –L option is used, only information about users on the local node is displayed. This option also suppresses the display of the node name from the output of the whox command.

2.3.9.9.2.9 writex

The writex command is different from the write command in the following ways:

- An additional optional nodename parameter has been added to the command line, so that a user at a particular node can be conversed with.
- If no node name parameter is specified, then all nodes within the current user's sphere of interest are searched for the specified user.
- If the node name parameter is not specified, the user being written to is logged in at more than one node within the current user's sphere of interest, and one of the user's logins is on the current node, all the nodes the user is logged in at are displayed, but the login session at the current node is used as the message delivery point.
- If the node name parameter is not specified, the user being written to is logged in at more than one node within the current user's sphere of interest, and none of the user's logins is on the current node, all the nodes the user is logged in at are displayed, and an arbitrary node (from the list that was displayed) is used as the message delivery point.

2.4 Keep-Alive Service

The Keep-Alive Service (KAS) is a kernel extension which may be used by other portions of the kernel to receive notification of loss of contact with another node or of regained contact. From the point of view of the KAS, those portions of the kernel calling the KAS constitute the "user".

The KAS provides a general interface which allows the user to specify a pointer to a function to be called when the status of a specified node changes. The user also specifies a minimum time interval for the KAS to check the status of that node. The KAS communicates with the KAS extension on the other node, and thus it does not verify that any other particular service on the other node is functioning.

2.4.1 Interfaces

The KAS provides interfaces to

1. register a keep-alive with the server
2. cancel a previously registered keep-alive
3. and request an immediate check of the status of a node

2.4.1.1 kas_register

The interface to register a keep-alive is:

```
struct kas_request *
kas_register(node_t nodename,
    time_t interval,
    void (*callback)(char *nodename),
    int flag)
```

The arguments are:

nodename — the node to be checked by the KAS.

interval — the minimum time interval for checking, in seconds. The KAS may check more frequently if other keep-alives are active or if kas_check is called. The total time to detect that node status has changed is "interval" plus the time it takes to timeout on any RPC messages (see Internal Design below).

callback — The function to be called when the keep-alive detect a change. This function must be written such that it returns quickly to the caller. If it does not, the KAS daemon may be tied up for extended periods. The call back function should perform actions such as wakeups or setting flags in data structures. If it needs to do something more complicated, such as sending RPCs, it should either pass that work to another daemon or it should create a thread which will perform the action. The call back function is allowed to call any of the kas_* functions described here.

flag — If flag is 0, KAS will call the call back function when it detects that it has lost contact with the specified node. If flag is KAS_UP, KAS will call the call back function when it detects that it has regained contact with the node.

The return value of kas_register is only useful for a later call to kas_cancel.

2.4.1.2 kas_cancel

The interface to cancel a previously registered keep-alive is:

```
(void)
kas_cancel(struct kas_request *request)
```

The request argument is the return value from a previous call to kas_register. If the request is not found on the current queue, kas_cancel will just return.

2.4.1.3 kas_check

The interface to force an immediate check is:

```
(void)
kas_check(node_t nodename)
```

kas_check forces the timeout on any pending keep-alives to zero causing the KAS to immediately send RPC messages to check the status of the node. kas_check does not itself verify the status of the node. If the status has changes, the pending call back functions will be invoked.

2.4.2 Internal Design

The KAS is made up of 3 pieces:

1. the routines called by users to interface with KAS,
2. the KAS daemon which sends keep-alive RPCs and calls the call back functions,
3. and the KAS server which receives the RPCs sent by the KAS daemon or another KAS server.

The data structures used by these pieces are described below, followed by the design of each of the pieces.

2.4.2.1 Data Structures

Figure 21 shows the links between the data structures used by the KAS. One node structure is kept for each node for which keep-alives are being maintained. Each node structure points to a doubly linked queue of keep-alive requests for that node. The keep-alive requests on each queue are kept sorted by the time interval specified in that request. The time interval used for the node is the time interval for the first request on the queue. The node structures contain:

```
struct kas_node {
    struct kas_node *kn_next;       /* link to next node */
    struct kas_request *kn_queue;   /* list of requests */
    node_t kn_nodename;             /* the node for these keep-alives */
    time_t kn_lastcontact;          /* the time of the last contact */
```

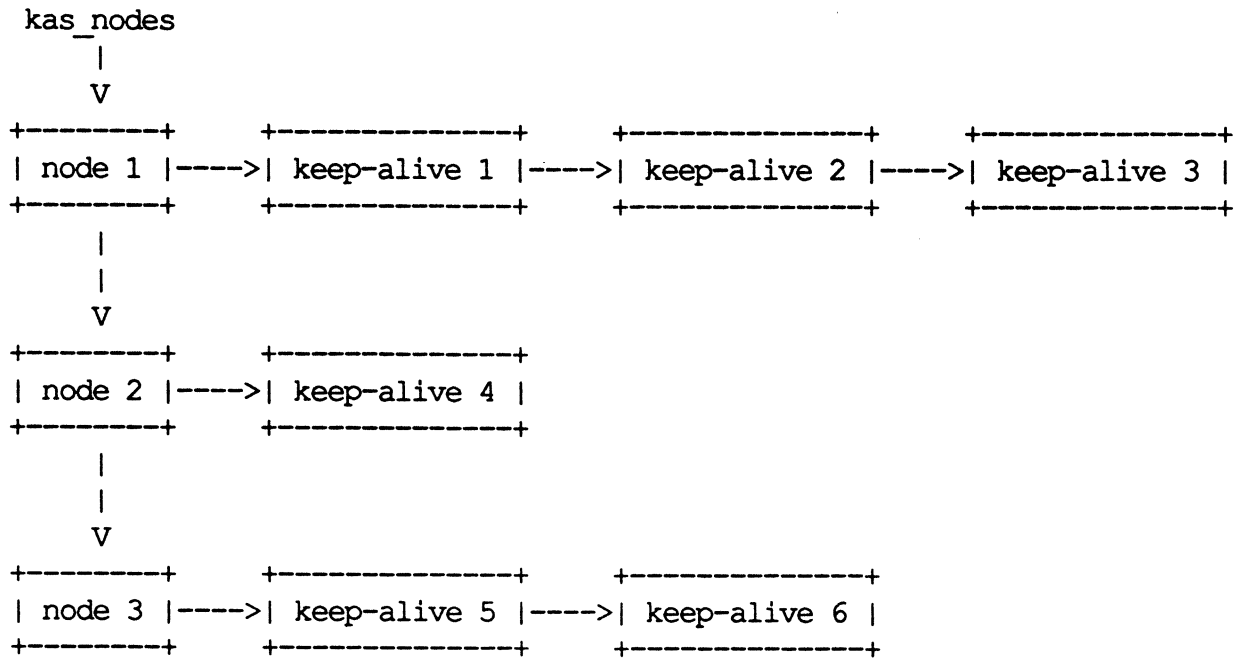


Figure 21. Keep-Alive data structure links

```
        int kn_flag;                    /* node status at last contact */
    };
```

The keep-alive structures contain:

```
    struct kas_request {
        struct kas_request *kr_prev;    /* pointers for doubly */
        struct kas_request *kr_next;    /*    linked list      */
        time_t kr_interval;             /* requested time interval */
        void (*kr_callback)();          /* requested call back function */
    };
```

2.4.2.2 Interfaces

NEEDSWORK: The implementation of the interfaces using the above structures is straight forward, but some pseudo-code should be included here. It should be noted that kas_register needs to wake up the KAS daemon in case the daemon will be sleeping for a period longer than the specified interval.

2.4.2.3 The KAS daemon

The KAS uses a daemon to check the status of nodes for which keep-alive requests have been made. This daemon normally runs as a single threaded job. It spawns off threads to perform retries when it appears that a node's status has actually changed.

To determine what work to perform next, the KAS daemon scans the node list, adds the time of last contact to the minimum keep-alive interval, and finds the node with the earliest sum. If that time has not yet arrived, the KAS daemon sleeps until that

time or until a call to kas_register or kas_check is made. If there is no work at all to be performed, the daemon will just sleep until kas_register or kas_check wakes it up.

When the KAS daemon wakes up, it send a RPC with the MAYBE flag to each node which has reached its next contact time. RPCs with the MAYBE flag do not wait for any response. This allows the daemon to send out any number of "pings" to different nodes without having to wait for responses. In addition, for each node whose time of last contact is more than KAS_TIMEOUT seconds ago, a thread is spawned to send a reliable RPC to verify if we have really lost contact with the node.

If a thread sends a reliable RPC and gets a failure, contact is assumed to be lost. Each call back function in the node's queue is called, and each keep-alive structure is freed. The node structure is also freed.

If the thread makes successful contact with the reliable RPC, it just updates the time of last contact and exits.

2.4.2.4 The KAS Server

When a MAYBE RPC is received from a KAS daemon on another node, a similar MAYBE RPC is sent back to that node except that this response RPC has a flag in it indicating that no further response is necessary. When an original MAYBE RPC or a response MAYBE RPC is received, that node's time of last contact in the node list is reset (if there is such a node in the local node list). Under normal conditions this means that the KAS daemon's MAYBE RPC will end up having the effect of updating the time of last contact on both nodes.

When a reliable RPC is received from the KAS daemon, the time of last contact is also updated. The KAS Server just returns, and it is the responsibility of the KAS daemon thread which sent the RPC to update its time of last contact.

3. The Cluster Environment

3.1 Clustering of Data

3.1.1 The Cluster Mount Service

3.1.1.1 Overview

While the DCE file system provides on-disk mounts for mounting file systems and volumes, such a mount becomes visible in the DCE name space and, as such, as visible to all members within the cell. Some users may desire to have mounted file systems which are not visible to the entire cell, but only to those within the cluster. The Cluster Mount Server (CMS) provides this ability with changes to /etc/mount, /etc/umount, lookup() and several AFS routines.

The CMS is responsible for maintaining consistent mount tables within the cluster. When a user attempts an /etc/mount, the CMS will grant permission for that mount by checking the mount tables for the entire cluster and determining if there is an inconsistency.

The CMS function consists of three parts, Cluster Mount Kernel Extension (CMKE) which caches mount information, the Local Cluster Mount Server (LCMS), which is part of the Local Node Status Server, and the Group Cluster Mount Server (GCMS) which caches all information about the cluster. All updates of mount information are handled by a central GCMS know as the Primary Group Cluster Mount Server (PGCMS). The LCMS uses the Group Service provided as part of the Node Status Server and caches and pushes information accordingly.

3.1.1.2 Who Tells What to Whom When...

3.1.1.2.1 Mounting a Non-Replicated Local PFS

1. On a locally stored non-replicated directory (inside the cluster)
   - get write token for local directory
   - get permission from GCMS
   - put mount vfs in local kernel
   - put mount hint in mounted-on vnode in local kernel memory
   - cache in local CMKE
2. On a remotely stored non-replicated directory
   - get write token for remote directory
   - contact GCMS for permission to mount on vnode
   - mount vfs in local kernel

- put mount hint in mounted-on vnode in remote kernel
- cache in local CMKE

3. On a remote non-replicated DCE directory outside the cluster
   - get write token for remote directory
   - contact GCMS for permission to mount on vnode
   - mount vfs in local kernel
   - cache in local CMKE
   - install mount hints in all node in the cluster
4. On a remote NFS directory outside the cluster
   - get NFS write token for remote directory
   - contact GCMS for permission to mount on vnode
   - mount vfs in local kernel
   - get list of nodes on which this NFS directory is mounted
   - inform the NFS Token Exporter of the mount
   - cache in local CMKE
5. On a DCE lazy replica mounted via /etc/mount
   - get the write token for the directory
   - contact GCMS for mount permission
   - get from LCMS the list of all storage nodes currently mounted via /etc/mount
   - mount vfs in local kernel with indicator to call the CMKE
   - determine where mount hints will be stored:

```
if replica is in cluster,
        store mount hints in each storage node's kernel
if replica is outside cluster,
        if other replicas are inside cluster,
                mark node in LCMS as "outside"
                install mount hints in all node in the cluster
        if no replicas are inside cluster,
                (see mounted on DCE directory outside cluster)
```

   - cache in local CMKE

- notify GCMS of mount hint information

6. On a DCE lazy replica mounted via AFS mounts
   - get write token for directory
   - contact VLDB for list of nodes storing replicas
   - contact GCMS for mount permission
   - mount vfs in local kernel with indicator to call the CMKE
   - determine where mount hints will be stored:
     if replica is in cluster,
       store mount hints in each storage node's kernel
     if replica is outside cluster,
       if other replicas are inside cluster,
         mark node in CMKE as "outside"
         store mounted on vnode in each node in cluster
       if no replicas are inside cluster,
         (see mounted on DCE directory outside cluster)
   - cache in local CMKE
   - notify GCMS of mount hint information
7. On a DCE r/w replica

   See mounting on a directory appropriate to the locale of the r/w replica since the r/w replica contains a different path name and will be treated like a non-replicated file system.
8. On a FRFS r/w replica mounted via /etc/mount
   - get write token for directory
   - contact GCMS for mount permission
   - get from LCMS a list of all storage nodes currently mounted via /etc/mount
   - mount vfs in local kernel with indicator to call the CMKE
   - determine where mounted on vnode will be stored:
     if replica is in cluster,
       store mounted on vnode in each storage node's kernel
     if replica is outside cluster,
       if other replicas are inside cluster,
         mark node in CMKE as "outside"
         store mounted on vnode in each node in cluster
       if no replicas are inside cluster,
         (see mounted on DCE directory outside cluster)

- cache in local CMKE
- notify GCMS with list of nodes to inform and appropriate flags to indicate where the mounted on vnode should be created in each kernel.

9. On a FRFS read-only replica mounted via /etc/mount

   See mounting on a directory appropriate to the locale of the r-o replica since the mount will only be visible on the r-o replica on which it is mounted.

   NOTE: In order to mount on a read-only replica of an FRFS the r-o replica either there is no primary or the r-o replica is mounted on a different mount point (see directory mounts).

10. On a FRFS r/w replica mounted via AFS mount

    See mounting on a DCE Lazy replica mounted via AFS mount.

11. On a FRFS read-only replica mounted via AFS mount

    See mounting on a DCE Lazy replica via AFS mount

    NOTE: Since mounting on r-o replica mounted via AFS can only occur because the primary is not up, the mount will be broadcast to all nodes as if the primary were available.

3.1.1.2.2 Mounting FRFS

3.1.1.2.2.1 Registering FRFS File services upon mounting

When using /etc/mount to manage replicated file systems, the replication information must be maintained by the GCMS. However, in the event that a GCMS goes down and returns, it will be necessary for the replication information to be written on disk so the GCMS can be updated. Each FUSION Replicated File System will have a hidden file, .replinfo, (See section X.X of the File System Replication Services (FSRS) for a description of the .replinfo file). which will contain information about the replicated file system.

Whenever a file volume is mounted using /etc/mount, it will be checked for .replinfo. If a valid .replinfo file exists, the default action is to preform mount the volume as part of the replicated file system. A mount option, -norepl, is provided to override this default action.

The contents of the .replinfo file are extracted and added to the the data held by the CMS on FRFS volumes. The replica will be added to the array of replicas listed in the fo_cms_entry. The CMS will store the address of the replica and its type. In addition, the fo_cms_entry (See Data Interfaces, section 3.1.1.11, for the contents of this structure) will be stored in the .replinfo file. For each additional replica, the .replinfo file on the read/write replica will be updated. This will cause the .replinfo file to propagate.

3.1.1.2.3 Lazy Replica

Lazy replicas will be registered with the GCMS which will have to maintain all information about the replicas. Since these are AFS volumes, they will be registered with the VLDB so no .replinfo file will be necessary. All this information is available through the VLDB. The GCMS will only know about those replicas which are currently mounted via /etc/mount within the cluster and will treat them as a single entity, ignoring all other replicas listed in the VLDB.

The Lazy read-only replica will be treated like all FRFS replicas and use the same mounting functions. The Lazy read/write replica will be treated like a local PFS since it represents a single volume.

3.1.1.2.4 NFS

To maintain NFS cache coherency, a NFS Token Exporter must be created for each NFS mount within the cell. The CMS checks the cell's name space for an existing NFS Token Exporter for the requested mount. If no NFS Token Exporter for the mount exists, a node is selected and it's location is stored in the cell's name space. The location of the NFS Token Exporter is also cached by the CMS. Mount hints are created by the same rules as mounting a non-replicated local PFS

3.1.1.2.5 Directories

Works like the PFS with the mounted on restrictions similar to those of a single UNIX site.

NEEDSWORK:

Other kernels may not support directory mounts as AIX 3.1 does. While we can disallow the mount on such a kernel, we will need to simulate the mount somehow so the other kernels can access the 3.1 directory mount.

3.1.1.3 Registering the Mount

3.1.1.3.1 Creating a Volume ID

When mounting a volume, CMS must give the volume a unique physical and mount ID. Currently AFS uses a double long to contain the physical volume ID. **FUSION** will reserve a range of the 64 bits available for its own use. This will guarantee that **FUSION** physical volume IDs will not conflict with AFS volume IDs. This range of physical volume IDs will be divided into two groups, permanent volume IDs and temporary volume IDs. Permanent physical volumes IDs are assigned to FRFS volumes.

If the /etc/mount is for a replicated file system, the mounting node looks to see if a valid volume ID exists. For replicated file systems, this is stored in the .replinfo file and the VLDB volume header. If no valid volume ID exists, a random volume ID based on the mounted text string is generated. The random generator will be one that creates numbers that are independent of node byte order. Conflicting volume IDs are detected and a different seed (based on the mounted text string) is tried. Mount IDs

will be created in a similar manner. A mount option is provided to allow the user to provide the volume ID. This would normally be used to resolve conflicting IDs that were generated during network partition.

3.1.1.3.2 Faking the VLDB Entry

Because AFS will need to fill in the volume registry for the cache manager and the protocol exporter, it will be necessary for the CMS to contain a volume structure similar in nature to struct vldbentry. This structure will be made up for each volume mounted using /etc/mount.

The AFS call VL_GetEntryByName() calls the VLDB to get information about the volume. This call will need to be modified to contact the CMS to get information about the volume. It will get a structure of type fo_cms_entry which contains all relevant information needed by the cache manager.

The LCMS will create this structure when the volume is mounted via /etc/mount. The structure will be set as follows:

- only one VolId will be set for each CMS mount = volume ID
- if a lazy or FRFS read-only replica, VolType = ROVOL else, VolType = RWVOL.
- name will be set to the volume name, if any
- volume_type will refer to the actual PFS type which /etc/mount must know in order to perform the mount
- frfsrepl will be set if this is an FRFS volume.
- nServers is the number of replicas registered with the CMS (see replication registry).
- siteAddr will be set to the network address of the additional servers per AFS.
- dotdot will be set to the mounted-on vnode.
- mounted will refer to the root vnode for this vfs.
- flags will be set for the type of volume, if read-only flags = VLF_ROEXISTS else, flags = VLF_RWEXISTS

3.1.1.4 Mount Context

The CMS provides access to the cluster-wide file system thru the use of pre-process mount context and a special CMS vfs. The pre-process mount context is composed of root relative and dot relative stored in the user structure. The "Modifications to AIX 3.1" section for details of the changes required to support mount context.

3.1.1.4.1 Using the CMS vfs to access all the data in the cluster

Access to the mount context of other nodes is provided by the CMS vfs mounted at "/.../cellname/clustername/CMS". This is a special vfs which supports only lookup,

readdir, getattr vnode operations. Lookups into "/.../cellname/clustername/CMS" access the CMS and shows the node names of active cluster members. Doing a

```
cd /.../cellname/clustername/CMS/nodex
```

results in the root file system of nodex being mounted at /.../cellname/clustername/CMS/nodex and the user's dot relative mount context being set to that of nodex. Since mount hints where installed on all local mounts on all nodes, nodex's mounts are detected by the path name lookup code and mounted. This gives the same view of nodex's file system name space as seen from nodex.

3.1.1.4.1.1 Changing root relative mount context

A process may change it's root relative mount context by use of the chmntcontext system call.

3.1.1.5 Startup, Shutdown, and Merging

3.1.1.5.1 Assumptions About Timeouts and Retries

1. Requests from a GCMS to an LCMS retry until the GNSS informs the GCMS that the node is down.
2. Requests from a LCMS to a GCMS will eventually succeed. This is due to the fact that the LNSS is always sending load information to the GNSS and locates a new group server when that operation fails. When the LNSS locates a group server, it registers the GCMS handle to the kernel through a system call. If the LCMS detects a timeout talking to the GCMS, it performs a system call to retrieve the new handle to the GCMS, rejoins the group, and retries the failed operation.
3. Requests from a PGCMS to a GCMS can fail. If all members of a group become unreachable, then the PGCMS loses contact with that GCMS. When this happens all nodes serviced by the unreachable GCMS are marked DOWN and the mount data as INVALID. The amount of time the PGCMS waits before declaring a group unreachable will be based on the time interval the GNSS uses for it's election of a new group server.
4. Requests from a GCMS to a PGCMS will eventually succeed. If a request from a GCMS to a PGCMS results in a timeout, the GCMS will enter into an election process to create a new PGCMS.

3.1.1.5.2 Starting a Group Cluster Mount Server

The GCMS is started by the GNSS forking an GCMS instance. As part of the startup operation, the GCMS registers two rpc interfaces with the GNSS. One is the interface that the GNSS will use to send merge and shutdown requests to the GCMS. The other is needed by the LCMS to talk to it's GCMS.

When the GCMS receives at cluster join request for a client node, it creates a instance of service for the cluster ID if it doesn't exist and informs the GNSS that it now servicing a new cluster. The GNSS returns to the GCMS a list of all other group

servers for the cluster ID. When the last node using a cluster ID leaves the group, the GCMS shutdowns service for the cluster ID and informs the GNSS and the PGCMS.

3.1.1.5.2.1 Locating the PGCMS

The GCMS locates it's PGCMS by checking the list returned by it's GNSS. This list contains both a RPC handle to all servers in the cluster and a flag indicating if the server is the PGCMS for that cluster. If the list has no valid entries for a PGCMS, then the GCMS attempt to select itself as PGCMS.

3.1.1.5.2.1.1 Selecting the PGCMS

The GCMS registers it's RPC handles with the GNSS with the flag set indicating that this is the PGCMS for this cluster. The GNSS passes this info to all active GNSSs. The GCMS pauses and then queries the GNSS for an updated list. The GCMS scans the list in numeric order starting at the lowest node number and attempts to join with any node indicating that it is the PGCMS. This continues until a PGCMS is located or it encounters it's node number. If a PGCMS is located, the GCMS registers it's RPC handles with the GNSS without the PGCMS flag set. Join requests that arrive during the time the GCMS is scanning the list receive an EBUSY reply. GCMSs receiving an EBUSY reply pause and retry the operation. Join requests which arrive after a PGCMS is located receive an TRYNODE reply that includes a handle to the PGCMS.

If the selection results in this node being elected PGCMS, the cluster mount table must be built. The PGCMS has two methods for acquiring an accurate cluster mount table.

1. get the cluster mount table from an active GCMS
2. get the mount information from all cluster members (through their GCMS)

Method 1 can be used only if there is an active GCMS and the GCMS has an accurate mount table. To avoid using inaccurate data, the PGCMS and all GCMS serialize all transactions which modify mount information. Each transaction that modifies mount data increments the start transaction count when the operation begins and end transaction count when the transaction ends. Before using method 1, the PGCMS checks to see if all the transaction counters match. If any of the counters are different, then the cluster mount table held in the GCMSs can not be trusted. In this case, method 2 is used.

3.1.1.5.2.2 Shutdown of an existing GCMS

The GNSS informs it's GCMS when it is shutting down as a result of a more preferred GNSS becoming available. Upon receiving the shutdown request, GCMS responds with TRYNODE to all client requests. Clients receiving a TRYNODE reply get a new GCMS handle from the kernel and retry the operation. The GCMS sends it's mount information for all clusters it is servicing to the more preferred GCMS. When the shutdown message arrives at the more preferred group server, it attempts to merge the cluster. If mount conflicts exist, the more preferred GCMS starts a new

database for the conflicting cluster data and allows the merging cluster to remain separate from any existing instance of the cluster. The conflicting mounts are logged and the preferred GCMS schedules an attempt to merge the cluster data. The more preferred GCMS returns EBUSY to all requests received during the merge operation. Clients receiving an EBUSY reply, pause and retry the operation.

3.1.1.5.2.3 Merging Group Mount Tables

The GNSS sends a merge request to the GCMS when groups discover each other. The GCMS contacts it's PGCMS who will attempt to merge the cluster mount tables. If the merge fails the mount conflicts are logged and the GCMS returns NOTOK to the GNSS. The PGCMS returns EBUSY to all update requests it receives when in the merge state. When the requesting GCMS receives an EBUSY reply, it pauses and then retries the operation.

3.1.1.5.3 Cluster Start

3.1.1.5.3.1 At the joining node

The clusterstart system call is handled by the CMKE. The CMKE builds the list of local and exported mounts and calls the LCMS. The LCMS prepares a list of of mount preferences, acquires volume IDs for all mounts, and determines the type of mount hint required. A cluster join request is sent to the GCMS which forwards the request to the PGCMS. The cluster join completes successfully if:

1. Exported mounts don't conflict with existing mounts.
2. All nodes in the cluster accept any exported mounts which have a locally stored mount point.
3. Joining node accepts all exported mounts from other nodes in the cluster which will be mounted on locally stored mount points.

If the join operation is successful, the node updates its mount information. This includes:

1. Install mount hints for all locally mounted file systems.
2. Export all locally mounted file systems through the AFS Protocol Exporter.
3. Install mount hints on storage sites for all mounts which have a hint type of STORAGE_NODES.
4. Have mount hints at all nodes in the cluster for all mounts which have a hint type of ALL_NODES.

3.1.1.5.3.2 At the PGCMS node

To insure that mounts are done without conflict, each mount (both the mounted and mounted on points) is locked for the duration of the join call.

If the joining node is exporting mounts which are stored on context dependent mount points, then all nodes in the cluster must accept the mount before the joins succeeds. The PGCMS has two ways of determining if a node accepts the mount. To avoid network traffic, nodes may have registered a list of mount points which they have a known preference. The PGCMS consults this list and queries all nodes which have not indicated a preference for the context-dependent mount points. Since join operations are allowed to happen in parallel, the following method is used to avoid race conditions.

The PGCMS maintains a list of nodes which have joined the cluster. Associated with this list is a transaction counter and a lock. Each time a node is added to the list, the lock is acquired and the transaction counter is incremented. When a join operation starts, the PGCMS acquires the lock, extracts a list of nodes to inform from the list and remembers the current transaction count. The PGCMS then sends the request to the GCMSs and waits for the replies. When all replies have been received, the transaction counter is checked. If the transaction counter has changed, a new list is built and the operation is repeated. This insures that nodes which are added to the cluster during the time PGCMS was waiting for replies from GCMSs get a chance to reject the mounts.

For each mount point that is accepted the PGCMS updates the accepting node's preference list. If all nodes accept the joining node's mounts and the joining nodes mounts don't conflict with existing mounts, then the PGCMS calls back to the joining node to get acceptance for existing context-depend mounts. The PGCMS holds the lock on the list of nodes in the cluster and defers processing any context dependent mounts for the duration of the call back to the joining node. If the joining node accepts the existing mounts, then the join is successful. All GCMSs are notified of the new mounts and the joining node is added to the nodes in the cluster list. If any NFS mounts where done, a check is made for a NFS Token Exporter for the mount. If a NFS Token Exporter is not located, one is started. If any mounts where done on an NFS mounted directory, then the NFS Token Exporter for that mount is informed of the mount.

3.1.1.5.4 Cluster Stop

3.1.1.5.4.1 At the LCMS

The departing node sends a leave request to it's GCMS. The GCMS forwards the leave request to the PGCMS who then informs all of the group servers for this cluster of the departure.

3.1.1.5.4.2 At the PGCMS

When a node leaves the cluster, all mounts that were exported to the cluster by the leaving node become unavailable. If any of the mount points provided by the departing node are mounted on, the leave request will return EBUSY and a list of busy mount points. (This action could be overridden by "force" type of switch).

NEEDSWORK: What does unavailable or umount really mean? /etc/mounts are expected to work like AFS volumes. What does AFS do when a volume is unattached and unmounted?

Ideally, umount would get a volume token which would cause all other tokens to be recalled. The token manager would not issue any new tokens a long as the volume token held. The problem with this solution it that a single physical volume can be accessed as many different logical volumes. Since the AFS token manager only knows about physical volumes, the CMS would have to track all using nodes of a mount and inform them of the unmount. This method is detailed below.

All nodes that have imported the mount are informed of the unmount (nodes importing a mount informed the GCMS of the event). The PGCMS forwards the departure message to all the GCMSs. The GCMSs marks the departing node as DOWN and send the departure message to all nodes within it's group which have non-persistent mounts sponsored by the departing node. The LCMS on the receiving node then forwards the message the CMKE. The CMKE then unmounts all non-persistent vfss that have the departing node as the mounting node. Both the CMKE and LCMS flush any cached mount data pertaining to the departing node. All non-persistent mount data pertaining to the departing node is marked as INVALID at the GCMS.

3.1.1.6 Mount Conflicts

The following tests are made to determine if a mount conflicts with the cluster mount image:

1. FRFS replicated file system with more than one R/W volume
2. physical volumes with the same replication information
3. conflicting physical or mount IDs
4. mount point is not a directory
5. mount point where a node has refused to allow a mount
6. multiple mounts on the same mount ID
7. mounting on a non-empty directory (optional)

3.1.1.7 Command Interfaces

3.1.1.7.1 /etc/mountx

The **FUSION** /etc/mountx extends /etc/mount by providing cluster-wide mounts and support for managing the **FUSION** replicas. The following additional operations are added:

1. Modes to display mount information by cluster, sphere of interest, node, storage nodes, and mount points.

2. Options for mounting and unmounting replicas.

3. Queries about the state of mount points.

NEEDSWORK: more detail

3.1.1.7.2 /etc/umountx

NEEDSWORK: Forced umounts.

NEEDSWORK: mount/umount permissions.

NEEDSWORK: more detail

3.1.1.8 Modifications to AIX 3.1

3.1.1.8.1 User structure changes

The user structure needs to be expanded to handle the pre process mount context.

```
struct mnt_context {
     u_short root;        /* root relative mount context */
     u_short dot;         /* dot relative mount context */
     u_short cms_dotdot; /* ".." mount context of CMS vfs */
}
 /*
  * somewhere in the user struct.....
  */
 struct mnt_context u_mntcontext;
```

3.1.1.8.2 Path name lookup

The CMS design allows /etc/mounts to be detected during path name lookup. This is accomplished with the use of mount hints. The path name lookup code in AIX 3.1 is modified to detect these hints and call the CMKE to have to mount performed.

```
if (vp->v_flag & V_CHECKMOUNT) {
        nvp = cmke_chkmnt(vp, u.u_mntcontext.lookup)
        if (nvp) {
                VNOP_HOLD(nvp);
                VNOP_RELE(vp);
                vp = nvp;
        }
}
```

".." evaluation must check to see if the vnode is marked as "mounted without mounted on vnode". If this is true and the vnode is marked as valid for ".." evaluation, then the mounted on vnode needs to be created.

```
if (vp->v_flag & V_NOMNTDON) {
        if (vp->v_flag & V_DOTDOTOK) {
                nvp = cmke_getmntdon(vp, u.u_mntcontext.lookup)
                if (nvp) {
                        VNOP_HOLD(nvp);
```

```
                        VNOP_RELE(vp);
                        vp = nvp;
                }
                else
                        goto eout;
        }
        else {
                u.u_error =  ENOENT;
                goto eout;
        }
}
```

lookuppn() sets the u.u_mntcontext.lookup at the start of the lookup operation.

```
if ( *pathname == '/')
    u.u_mntcontext.lookup = u.u_mntcontext.root;
else
    u.u_mntcontext.lookup = u.u_mntcontext.dot;
```

3.1.1.8.3 chdir() modifications

Calls which change the u_cdir of the user structure must also change the dot relative mount context.

```
/*
 * update the dot relative mount context
 * when u.u_cdir changes
 */
u.u_mntcontext.dot = u.u_mntcontext.lookup;
```

3.1.1.8.4 mount() and umount() system calls

All mount and umount operations done while the node is clusterstarted need to call the CMKE to make to mount points available cluster wide.

3.1.1.9 Modifications to AFS

Changes have been proposed that would provide the following functions:

1. Provide a means of installing and detecting mount hints using the AFS Protocol Exporter and Cache Manager.
2. The AFS Protocol Exporter will notify the CMS when a volume is moved. (needed to maintain the mount hints installed in #1 above).
3. Allow CMKE to perform volume lookups for CMS /etc/mounts.
4. Add logical volume IDs to the AFS cache manager and AFS Protocol Exporter operations.

3.1.1.10 Module Interfaces

3.1.1.10.1 Between LCMS and GCMS

3.1.1.10.1.1 LCMS -> GCMS

```
/*
 * join a cluster
 * state indicates if this join is due to
 * losing the previous group server
 * the following are only valid when joining
 * after losing the previous group server:
 *     prev_server
 *     imported_mounts
 */
lcms_joincluster (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* in */    int             state,
     /* in */    int             groupserver,
     /* in */    int             boot_sequence,
     /* in */    mount_list      local_mounts,
     /* in */    mount_list      exported_mounts,
     /* in */    mount_list      imported_mounts,
     /* in */    preference      mount_points,
     /* out */   conflict_list   list_of_mountconflicts,
     /* out */   unsigned32      *st
)

/*
 * leave a cluster
 */
lcms_leave (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* out */   unsigned32      *st
)

/*
 * export mounts to the cluster
 */
```

```
lcms_mount (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* in */    int             boot_sequence,
     /* in */    mount_list      list_of_mounts,
     /* out */   conflict_list   list_of_mountconflicts,
     /* out */   unsigned32      *st
 )

/*
 * umount a list of mounts
 */
lcms_umount (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* in */    int             boot_sequence,
     /* in */    mount_list      list_of_mounts,
     /* out */   conflict_list   list_of_mountconflicts,
     /* out */   unsigned32      *st
)

/*
 * get mount info for a mounted global
 * file handle
 */
lcms_getmntby_mntd_gfh (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* in */    int             mount_context,
     /* in */    int             boot_sequence,
     /* in */    gfh             mntdgfh,
     /* out */   mntinfo         mount,
     /* out */   unsigned32      *st
 )

 /*
  * get mount info for a mounted-on global
```

```
 * file handle
 */
lcms_getmntby_mntdon_gfh (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* in */    int             mount_context,
     /* in */    int             boot_sequence,
     /* in */    gfh             mntdgfh,
     /* out */   mntinfo         mount,
     /* out */   unsigned32      *st
)

/*
 * get mount info for a list of nodes
 */
 lcms_getmntby_nodes (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* in */    int             boot_sequence,
     /* in */    int             mount_context,
     /* in  */   int             nodes[],
     /* out */   mntinfo         mount,
     /* out */   unsigned32      *st
)

/*
 * get a sequence of temporary volume IDs
 * type_of_ID is either logical or physical
 */
lcms_get_volIDs (
     /* in */    int             cell,
     /* in */    int             group,
     /* in */    int             cluster,
     /* in */    int             node,
     /* in */    int             boot_sequence,
     /* in */    int             number_of_IDs,
     /* in */    int             type_of_ID,
     /* out */   int             *tempID,
     /* out */   unsigned32      *st
```

```
)

/*
 * add mount hints to all nodes in cluster
 */
lcms_addhints (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    hint_list       list_of_mounthints,
    /* out */   unsigned32      *st
)
```

3.1.1.10.1.2 GCMS -> LCMS

```
/*
 * see pgcms_ok2mnt for details
 */
gcms_ok2mnt (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    mount_list      list_of_mounts,
    /* out */   conflict_list   list_of_mountconflicts,
    /* out */   unsigned32      *st
)

/*
 * see pgcms_getmntinfo
 */
gcms_getmntinfo (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* out */   mntinfo         local_mounts,
    /* out */   mntinfo         exported_mounts,
    /* out */   mntinfo         imported_mounts,
    /* out */   preference      mount_points,
)
```

NEEDSWORK:

3.1.1.10.2 Between GCMS and PGCMS

3.1.1.10.2.1 PGCMS –> GCMS

```
/*
 * update the cluster mount table stored in group
 * servers
 */
pgcms_addmounts (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    mount_list      list_of_mounts,
    /* in */    hint_list       list_of_mounthints,
    /* out */   unsigned32      *st
)

/*
 * update the cluster mount table stored in group
 * servers
 */
pgcms_deletemounts (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    mount_list      list_of_mounts,
    /* out */   unsigned32      *st
)

/*
 * install mount hints
 */
pgcms_addhints (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    hint_list       list_of_mounthints,
    /* out */   unsigned32      *st
)

/*
```

```
 * PGCMS informs all group servers when any node goes down
 */
pgcms_nodedown (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             nodes2down[],
    /* out */   unsigned32      *st
)

/*
 * a join or mount operation is mounting over
 * a context-dependent mount point
 * the PGCMS requests all group servers to get acceptance
 * from all their clients
 */
pgcms_ok2mnt (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    mount_list      list_of_mounts,
    /* out */   conflict_list   list_of_mountconflicts,
    /* out */   unsigned32      *st
)

/*
 * PGCMS callback to GCMS to have the joining node (node2ask)
 * to accept the context-dependent mounts that
 * currently exist in the cluster
 */
pgcms_accept2join (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    int             node2ask,
    /* in */    mount_list      list_of_mounts,
    /* out */   conflict_list   list_of_mountconflicts,
    /* out */   unsigned32      *st
)

/*
```

```
 * Return a list of mounts for the indicated cluster
 */
pgcms_get_mntinfo (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* out */   mntinfo         local_mounts,
    /* out */   mntinfo         imported_mounts,
    /* out */   mntinfo         exported_mounts,
    /* out */   preference      mount_points,
    /* out */   unsigned32      *st
)

/*
 * Return number of transactions started and completed
 */
pgcms_get_transcnt (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* out */   u_int           db_trans_in;
    /* out */   u_int           db_trans_out;
    /* out */   unsigned32      *st
)

/*
 * reset the cluster mount table
 * issued when PGCMS starts up and
 * the cluster mount table had to be rebuilt
 */
pgcms_reset (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    cluster_list    cluster_mount_table,
    /* out */   unsigned32      *st
)
```

NEEDSWORK: more detail

3.1.1.10.2.2 GCMS -> PGCMS

```
/*
 * Join the cluster as a group server.
 * fails if a more perferred group server exists for this group.
 * PGCMS returns the cluster mount table if the
 * join succeeds.
 */
gcms_joincluster (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    int             boot_sequence,
    /* in */    int             fs_types_supported[],
    /* out */   cluster_list    cluster_mount_table,
    /* out */   handle_t        newserver,
    /* out */   unsigned32      *st
)

/*
 * Inform the PGCMS that this node no longer services
 * the indicated cluster.
 */
gcms_leavecluster (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    int             boot_sequence,
    /* out */   handle_t        newserver,
    /* out */   unsigned32      *st
)

/*
 * Inform the PGCMS of a inactive node.
 * Node down is detected by the GNSS and
 * passed to the GCMS.
 */
gcms_nodedown (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
```

```
    /* in */    int             node2down,
    /* out */   handle_t        newserver,
    /* out */   unsigned32      *st
)
```

3.1.1.10.2.3 PGCMS -> PGCMS/GCMS

```
/*
 * shutdown as group server
 * called when the GNSS has detected a more
 * preferred GNSS
 * mount conflicts are allowed to continue by creating
 * an instance of the cluster database in the
 * more preferred node
 */
pgcms_resign (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    cluster_list    cluster_mount_table,
    /* out */   unsigned32      *st
)

/*
 * merge groups
 * fails if mount conflicts exist
 */
pgcms_merge (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
    /* in */    cluster_list    cluster_mount_table,
    /* out */   conflict_list   list_of_mountconflicts,
    /* out */   unsigned32      *st
)
```

3.1.1.10.3 Between CMKE and LCMS

3.1.1.10.3.1 CMKE -> LCMS

```
/*
 * get mount info for a mounted global
```

```
 * file handle
 */
cmke_getmntby_mntd_gfh (
    /* in */    int             mount_context,
    /* in */    gfh             mntdgfh,
    /* out */   mntinfo         mount,
    /* out */   unsigned32      *st
)

/*
 * get mount info for a mounted-on global
 * file handle
 */
cmke_getmntby_mntdon_gfh (
    /* in */    int             mount_context,
    /* in */    gfh             mntdgfh,
    /* out */   mntinfo         mount,
    /* out */   unsigned32      *st
)

/*
 * get mount info for a list of nodes
 */
cmke_getmntby_nodes (
    /* in */    int             mount_context,
    /* in  */   list            nodes,
    /* out */   mntinfo         mount,
    /* out */   unsigned32      *st
)

/*
 * called at clusterstart
 */
cmke_joincluster (
    /* in */    int             boot_sequence,
    /* in */    int             cluster,
    /* in */    int             fs_types_supported[],
    /* in */    mount_list      local_mounts,
    /* in */    mount_list      exported_mounts,
    /* in */    mount_list      imported_mounts,
    /* in */    preference      mount_points,
    /* out */   conflict_list   list_of_mountconflicts,
    /* out */   unsigned32      *st
```

```
)


cmke_leavecluster (
    /* in */    int             boot_sequence,
    /* in */    int             cluster,
    /* out */   unsigned32      *st
)

/*
 * export a list of mounts
 */
cmke_mount (
    /* in */    int             boot_sequence,
    /* in */    mount_list      list_of_mounts,
    /* out */   conflict_list   list_of_mountconflicts,
    /* out */   unsigned32      *st
)

/*
 * umount a list of mounts
 * returns a list of conflicts if the operation fails
 */
cmke_umount (
    /* in */    int             boot_sequence,
    /* in */    mount_list      list_of_mounts,
    /* out */   conflict_list   list_of_mountconflicts,
    /* out */   unsigned32      *st
)

/*
 * locate NFS Token Exporter for mounted gfh
 * returns handle to NFS Token Exporter
 * called by the NFS client code when it is unable
 * to talk to it's NFS Token Exporter
 * this call is forwarded to the PGCMS (thru the LCMS,GCMS)
 * were a NFS Token Exporter is located/started
 */
cmke_get_NFSexporter (
    /* in */    int             cell,
    /* in */    int             group,
    /* in */    int             cluster,
    /* in */    int             node,
```

```
        /* in */    int             boot_sequence,
        /* in */    gfh             mntdgfh,
        /* out */   handle_t        NFSexporter,
        /* out */   unsigned32      *st
    )
```

NEEDSWORK:

3.1.1.10.3.2 LCMS -> CMKE

```
    /*
     * return information about all mounts
     * called when a new PGCMS is rebuilting the mount table
     */
    lcms_get_mntinfo (
        /* out */   mntinfo         local_mounts,
        /* out */   mntinfo         imported_mounts,
        /* out */   mntinfo         exported_mounts,
        /* out */   preference      mount_points,
        /* out */   unsigned32      *st
    )

    /*
     * install mount hints
     */
    lcms_set_mnthints (
        /* in  */   hintlist        hints,
        /* out */   unsigned32      *st
    )
```

3.1.1.10.4 Between GCMS and GNSS

3.1.1.10.4.1 GNSS -> GCMS

```
    /*
     * merge groups
     * GNSS provides a list of merging group servers
     * operation fails if conflicting mounts exist
     */
    gnss_merge(
        /* in  */   handle_t        gcms_servers[],
        /* out */   unsigned32      *st
    )

    /*
     * stop providing GCMS services and transfer
```

```
 * cluster mount data to a more preferred node
 */
gnss_shutdown(
     /* in  */   handle_t        preferred_gcms,
     /* out */   unsigned32      *st
)
```

3.1.1.10.4.2 GCMS -> GNSS

```
/*
 * register RPC handles of the GCMS
 */
gcms_registerhandle(
     /* in  */   handle_t        GNSS2GCMS_handle,
     /* in  */   handle_t        LCMS2GCMS_handle,
     /* out */   unsigned32      *st
)

/*
 * inform the GNSS of new cluster(s) instance
 * the GNSS returns a list of handles to other
 * group servers
 */
gcms_startcluster(
     /* in */    cluster_list    myserverarea[],
     /* out */   cluster_list    gcms_servers[],
     /* out */   unsigned32      *st
)

/*
 * inform the GNSS when the last node using a
 * cluster leaves
 */
gcms_stopcluster(
     /* in  */   int             cluster_ID,
     /* out */   unsigned32      *st
)
```

3.1.1.10.5 System Calls

```
/*
 * change root relative mount context
 * fails if:
 *    chroot has already been performed
 *    requested node is unavailable
 *
```

```
 * NEEDSWORK: error numbers
 */
chmntcontext(
    /* in */ char *nodename
)

/*
 * clusterstart
 * fails if mount conflicts exist
 */
clusterstart(
    /* in */ cluster_id
)

/*
 * clusterstop
 */
clusterstart()

/*
 * get RPC handle to GCMS
 */
getGCMShandle(
    /* out */ GCMS_handle
)

/*
 * set the RPC handle to the LCMS (used by the CMKE)
 */
setLGCMShandle(
    /* in */  LGCMS_handle
)

/*
 * get RPC handle to CMKE
 */
getCMKEhandle(
    /* out */ GCMS_handle
```

NEEDSWORK: more detail

3.1.1.11 Data Interfaces

3.1.1.11.1 Volume Structures

```
/*
 * One structure per volume, describing where the volume is located
 * and where its mount points are.
 */
struct cm_volume {
    struct cm_volume *next;      /* Next volume in hash list. */
    struct cm_cell *cellp;       /* this volume's cell */
    struct lock_data lock;       /* the lock for this structure */
    struct afsHyper volume;      /* This volume's ID number. */
    char *volnamep;              /* This volume's name, or 0 if unknown */
    struct cm_server *serverHost
        [AFS_MAXHOSTS];          /* servers serving this volume */
    struct afsFid dotdot;        /* dir to access as .. */
    struct afsFid mtpoint;       /* The mount point for this volume. */
    struct afsHyper roVol;       /* RO volume id associated with vol (if any) */
    struct afsHyper backVol;     /* BACKUP vol id associated with vol (if any) */
    struct afsHyper rwVol;       /* RW volume id for this volume */
    long accessTime;             /* last time we used it */
    long vtix;                   /* volume table index */
    long copyDate;               /* copyDate field, for tracking vol releases */
    short refCount;              /* reference count for allocation */
    char states;                 /* added here for alignment reasons */
};

/*
 * sections of vldbentry needed to create cm_volume for /etc/mount volumes
 */
struct fo_cms_entry {
        struct afsHyper VolIds[MAXVOLTYPES];
        unsigned long VolTypes[MAXVOLTYPES];
        char name[MAXNAMELEN];
        unsigned long volumeType;
        char frfsRepl;
        unsigned long nServers;
        struct fo_cms_replentry[MAXNTYPES];
        struct afsFid dotdot;
        struct afsFid mtpt;
        unsigned long flags;
}

struct fo_cms_replentry {
        struct afsNetAddr siteAddr;
```

```
        char    type;
}
```

3.1.1.11.2 GCMS Private data

```
/*
 * group server membership
 */
struct cms_id {
    u_short             cell;
    u_short             group;
    u_short             cluster;
};

/*
 * info kept for each node
 */
struct cms_members {
    u_short             nodenum;        /* internal node number */
    rpc_handle_t        node_h;         /* node's rpc handle */
    u_int               node_state;
    cms_members         *next;
};

/*
 * list of nodes which
 * require a lock to access
 */
struct cms_list {
    struct cms_members  *members;
    u_int               lock;           /* lock required to update list */
    u_int               cnt;            /* # of members in list */
};

/*
 * busy mount points
 * kept by mounted and mounted on mount IDs                              |
 */
struct cms_bsymnt {
    volume_t            mntd_ID;        /* mounted ID */                  |
    volume_t            mntdon_ID;      /* mounted on ID */               |
    cms_bsymnt          *next;
};

/*
```

```
 * data kept by each group server
 * entries marked with (PGCMS) are
 * only valid when the entry is also the primary group server
 */
struct gcms {
    struct cms_id       id;             /* service area where this data
                                         * applies
                                         */
    u_int               gcms_state;
    u_int               pgcms_state;    /* (PGCMS) */
    u_int               flags;
    struct cms_list     *group;         /* nodes in this group */
    struct cms_list     *cluster;       /* nodes in cluster (PGCMS) */
    struct cms_list     *servers;       /* group servers in cluster (PGCMS) */
    struct cms_bsymnt   *busymnts;      /* in transit mount points */
    mutex_t             bsymnts_lock;   /* lock for busy mounts list */
    rpc_handle_t        pgcms_h;        /* rpc handle to PGCMS */
    db_handle_t         db_mount_h;     /* mount data base handle */
    mutex_t             db_mount_lock;  /* lock for mount data base */
    db_handle_t         db_node_h;      /* node data base handle */
    mutex_t             db_node_lock;   /* lock for node data base */
    u_int               db_node_cnt;    /* # of additions to node db */
    u_int               db_trans_in;    /* data base mods started */
    u_int               db_trans_out;   /* mount data base mods completed */
    cms_gcms            *next;
};

static struct gcms *gcms;               /* service info for each
                                         * cluster this node services
                                         */

/*
 * states of the PGCMS / GCMS (pgcms_state / gcms_state)
 */
#define NOTASERVER      0x0001          /* shutdown */
#define RUN             0x0002          /* fully operational instance */
#define PENDING         0x0004          /* instance is starting up */
#define SHUTDOWN        0x0008          /* instance has encountered a
                                         * server with a higher priority
                                         * and is attempting to shutdown
                                         */
#define MERGING         0x0010          /* instance is involved with a
                                         * merger with an group
```

```
                                        */
#define PARTITION       0x0020          /* mount conflicts prevented
                                         * cluster join
                                         */

/*
 * state held by the GCMS for each of it's group members (node_state)
 */
#define NOTACTIVE       0x0001           /* not a cluster member */
#define WAITING         0x0002           /* node is waiting for a reply */
```

3.1.1.11.3 LCMS Private data

```
struct lcms {
    struct cms_id       id;
    u_int               lcms_state;
    rpc_handle_t        cmke_h;          /* rpc handle to CMKE */
    rpc_handle_t        gcms_h;          /* rpc handle to GCMS */
};
```

3.1.1.11.4 CMKE Private data

```
/*
 * cached info about mounts
 */
struct cmke_mount {
   struct vfs           *vfs;           /* mounted vfs */
   struct vnode         *vp_held;       /* list of vnodes being held to
                                         * provide mount hints
                                         * must be released when umounted
                                         */
  struct mntinfo        *info;          /* information needed to
                                         * recreate this mount
                                         */
  struct cms_mount      *next;
};

struct cmke {
    struct cms_id       id;
    u_int               cmke_state;
    rpc_handle_t        lcms_h;          /* rpc handle to LCMS */
    struct cmke_mount   *local_mounts;
    struct cmke_mount   *exported_mounts;
    struct cmke_mount   *imported_mounts;
    struct preference   *predefined;    /* list of predefined mount points
```

```
                                        * that this node will accept or
                                        * reject
                                        */
};

/*
 * state of node (cmke_state)
 */
#define CMKE_CLUSTERSTARTED     0x0001;
#define CMKE_MOUNTCONFLICTS     0x0002;     /* mount conflicts prevented node
                                             * from joining cluster
                                             */
```

3.1.1.11.5 Gobal Files

3.1.1.11.5.1 NFS Token Exporter

Each unique NFS mount requires a NFS Token Exporter. The node selection is handled by the PGCMS when a NFS mount is performed. Handles to the server are stored in the as:

```
name     : /.../cellname/clustername/CMS/NFSTokenExporter4_ROOTFH
contents: RPC handle to NFS TokenExporter
```

were ROOTFH is the root file handle (in ascii) that the NFS Token Exporter services. Note that a NfS Token Exporter service area is cell wide.

3.1.1.11.6 Between GCMS and GNSS database

The GCMS cluster mount information is managed by routines provided by the GNSS. The GCMS stores two types of data in this data base; cluster mount information and information about nodes in the cluster.

```
Mount Record

STRING          mounted on text
STRING          mounted text
OPAQUE          mount arguments
GFH             mounted on global file handle
GFH             mounted global file handle
INT             mounting node
INT             flags
INT             mount context
MOUNT_ID        mount ID of this mount                          |
LIST            list of mount hints for this volume
LIST            list of physical volumes storing the mount
                each entry contains:
                    storage node
```

```
                        type (RW, RO, REPL_TYPE, etc)
                        physical volume ID
                        server's network address
        LIST            list of nodes which have imported the mount
                        valid only at the group servers


        Node Record

        INT             node number
        INT             group number
        INT             flags
        INT             boot sequence number
        LIST            list of mount preferences
        LIST            list of file system types supported
```

NEEDSWORK: more detail

3.1.1.11.7 Between kernel extensions and CMS

NEEDSWORK: Other **FUSION** modules may need an interface to the CMS. These would (probably) include reopen, automatic load leveling, and remote devices. These details need to be worked out when more information is available.

3.1.1.11.7.1 Between replication user programs and CMKE

The FRFS replication service provides tools which allows the the storage locations and types of volumes to change while the replicas are mounted. This information needs to be sent to the CMKE when the changes are made. **NEEDSWORK:** define the interface

3.1.2 NFS Interoperability

3.1.2.1 Purpose

This design specification provides desing information for changes to the NFS extensions required for the FUSION project.

3.1.2.2 NFS Mount Model

3.1.2.2.1 Overview

Mounts must be consistent throughout the cluster with a high degree of availability. The enhancements to NFS to provide this break down into two major areas. This first is allowing multiple mounts of the same file system on different nodes while at the same time detecting conflicting mounts. The second area is providing hints that the cluster mount server should be consulted about the mounted on status of a vnode.

3.1.2.2.1.1 Mount support

In order to increase availability it is desirable to allow more than one node within a cluster perform a given NFS mount. In order to allow this and detect conflicts when different file systems are mounted on the same vnode a unique id is assigned to each mount. This unique id is referred to in this document as a mount id. The mount id can either be assigned automatically or be manually assigned and passed in as a mount option.

3.1.2.2.1.2 Automatic assignment of mount id

This is the procedure which would typically be used for a mount which is of a very temporary nature. This type of mount is referred to either as a normal mount or a non administered mount. The mount id for this mount will be composed of a flag which identifies the mount as normal and a uuid. It is therefore guaranteed that any other mount attempted on a vnode which has had automatic mount id assignment will conflict.

3.1.2.2.1.2.1 Advantages

The primary advantage of this type of mount is ease of use. The mount command can be used exactly as it would on any other system. It is not possible to make an error and end up with conflicting mounts on the same vnode.

3.1.2.2.1.2.2 Disadvantages

The primary disadvantage of this type of mount is availability. As only one node can mount a file system that file system will not be available until that node joins the cluster. Also if the node which did the mount crashes or otherwise leaves the cluster the file system must be force unmounted. The reason for this is on a crash or other reboot the mounting node will get a different uuid to create the mount id. The mounting node would then be unable to join the cluster because the mount ids would conflict.

3.1.2.2.1.3 Administered mounts

File systems which have mount ids assigned by hand, perhaps with automated help, are called administered mounts. The mount id is either gotten from the mount command line or from the local control file.

3.1.2.2.1.3.1 Advantages

The primary advantage of this type of mount is availability. Multiple nodes can have mount the same file system at the same place. As soon as the first node which does the mount joins the cluster the mount becomes available to all nodes within the cluster. When other nodes which have done the mount are allowed to join the cluster because the mount ids do not conflict. Administered mounts do not need to be unmounted when the mounting node goes away as presumably on reboot the mount will have the same mount id. In short administered mounts overcome the disadvantages of normal mounts.

3.1.2.2.1.3.2 Disadvantages

Human intervention is required to select the mount id. If the administrators setting up the mounts are not careful different file systems can be mounted on the same point causing random inconsistencies.

3.1.2.2.1.4 Mount hints

In a large cluster it becomes expensive to contact every node when a mount is done. To circumvent this hints will be supplied that tell a node that the cluster mount server should be contacted to see if the directory in question is mounted on.

3.1.2.2.1.4.1 Mount and Unmount Count

There is a mount and unmount count associated with directory vnodes and with the LOOKUP token. When a vnode is created on a client both the mount and unmount counts are initialized to zero. When tokens are created by the NFS token exporter the counts are also initialized to zero. The reason for both mount and unmount counts is to allow a NFS token exporter to serve multiple clusters. This is a future enhancement that is very easy to accommodate now.

3.1.2.2.1.4.2 Hint detection

XXX Possible base code change — this needs to be worked out with cluster mount server.

Hopefully by a vnode operation but otherwise by a direct call the code above the vnode layer will ask if a vnode is mounted on. The code below the vnode layer will get the LOOKUP token from the NFS token cache manager, which may have had to get the token from the NFS token exporter. The mount and unmount counts in the token and vnode are compared to see if the cluster mount server should be contacted.

3.1.2.2.1.4.3 Hint mount and unmount

Whenever a mount or unmount occurs on top of a NFS vnode the cluster mount sever will have to contact the NFS token exporter for the mounted/unmounted file system. Whenever the NFS token exporter is told of a mount the mounted count will be

incremented. Whenever the NFS token exporter is told of an unmount the unmount count will be incremented. The NFS token exporter will return to the cluster mount server the number of mounts and unmounts on the vnode to allow for more reliable operation when a NFS token exporter is moved to a new node.

3.1.2.2.2 New Functions

3.1.2.2.2.1 nfs_checkmount(vp)

3.1.2.2.2.1.1 Overview

Checks the mount and unmount counts in the vnode and it's associated token and determines if the cluster mount server should be contacted. The vnode mount and unmount counts are updated from the token. Returns TRUE if the cluster mount server should be contacted.

3.1.2.2.2.1.2 Detailed Design

```
nfs_checkmount(vp)
{
        flag = don't talk to cluster mount server
        if(the vnode is not already mounted on) {
                if(counts in token != counts in vnode) {
                        if(token mount count != token unmount count) {
                                flag = talk to cluster mount server
                        }
                }
        }
        else {
                if(counts in token != counts in vnode) {
                        flag = talk to cluster mount server
                }
        }
        counts in vnode = counts in token
        return(flag)
}
```

3.1.2.3 Migration of open NFS files

3.1.2.3.1 Overview

When a process migrates any vnodes which are being referenced by the process need also to be migrated. Two NFS vfs operations (nfs_prepare_for_export() and nfs_reopen()) are added to support process migration. The nfs_prepare_for_export() vfs operation returns a global file handle (gfh) for the vnode which can be used on any **FUSION** node. The nfs_reopen() vfs operation is passed the gfh and creates a vnode given a global file handle.

When migrating a vnode to a new node it will often be true that the file system that the vnode is part of has not yet been mounted on the migrated-to node. The file

system will have to be mounted when this occurs. For this reason vnode migration and file system mount are very intertwined. The following support is needed from the cluster mount server to implement this approach.

1. When a mount is done the cluster mount server will be given the global file handle (gfh) for both the mounted on and mounted vnode along with all the mount options.
2. The cluster mount server must be able to supply the root vnode global file handle given a mount id.
3. The cluster mount server must be able to supply the global file handle that a given mount id is mounted on.
4. The cluster mount server must contact the appropriate hint code for the mounted on file system.
5. The cluster mount server must be able to supply mount information given a mount id.

A new vnode operation , gfh2vnode(), is added. This vnode operation is given a global file handle and returns a vnode. gfh2vnode() handles any mount operations required to make the vfs referenced by the global file handle available.

3.1.2.3.2 New Functions

3.1.2.3.2.1 nfs_prepare_for_export(vp)

3.1.2.3.2.1.1 Overview

Additional vfs operation prepare_for_export() for preparing a vnode for migration. Returns a global file handle for the vnode.

3.1.2.3.2.1.2 Detailed Design

```
nfs_prepare_for_export(vp)
{
        /*
         * XXX
         */
}
```

3.1.2.3.2.2 nfs_prepare_for_export(gfh)

3.1.2.3.2.2.1 Overview

Additional vfs operation reopen() to reopen a vnode on a migrated-to node. Calls nfs_gfh2vnode() which will perform any necessary mount and create a vnode from the global file handle. If the vnode is created by nfs_gfh2vnode(), then a call to the NFS token cache manager is made to acquire the OPEN token for the vnode. If this vfs is mounted with the secure NFS option, a call is made back to the migrated-from node to get the login information required by secure NFS.

3.1.2.3.2.2.2 Detailed Design

```
nfs_reopen(reopen_data)
{
        /*
         * XXX
         */
}
```

3.1.2.3.2.3 nfs_vnode2gfh(vnode)

3.1.2.3.2.3.1 Overview

Create global file handle (gfh) for a given vnode. Called by nfs_prepare_for_export() to build the global file handle. The following is placed in the private data section of the global file handle:

1. NFS file handle
2. useful vnode fields
    a. v_type
    b. v_flags
    c. v_rdev

3.1.2.3.2.3.2 Detailed Design

```
gfh *
nfs_vnode2gfh(vnode)
{
        /*
         * XXX
         */
}
```

3.1.2.3.2.4 nfs_gfh2vnode(gfh, address of routine to get mount info)

3.1.2.3.2.4.1 Overview

Create a vnode for a given global file handle. If the vfs is not currently mounted, performs the necessary mount operation. Returns a pointer to the vnode or NULL if the vnode creation failed.

3.1.2.3.2.4.2 Detailed Design

```
nfs_gfh2vnode(gfh, address of routine to get mount info)
{
        pick the mount id out of the gfh                                    |
        if(this mount id is not already mounted on this node)               |
        {
            call passed in routine to get mount info
            /*
```

```
            *  if just migrated in it will be from migrate from node
            *  else if will be from cluster mount server
            */
           if (mounted on mount id is not already mounted on this node)|
           {
                   make fake vnode
                   do NFS mount vfs op to mount file system on fake vnode
                   if(mount error)
                   {
                       trash vnode
                       return(error)
                   }
                   trash fake vnode

                   mark root vnode as "mount point without a mounted
                   on vnode"

                   if(migrate from node is in another cluster)
                   {
                       mark root vnode as not eligible for ".." evaluation.
                   }
           }
           else
           {
                   do NFS mount vfs op to mount file system on mounted on
                   vfs

                   if(mount error)
                   {
                       return(error)
                   }
           }
       }
       create vnode using NFS file handle in gfh
       stat vnode
       if(stat failed)
           return(error)
       return(success)
}
```

3.1.2.3.3 Changes to existing functions

3.1.2.3.3.1 ".." evaluation

3.1.2.3.3.1.1 Overview

When a mount is done a part of process migration, a vnode is created without a mount-on vnode. When ".." evaluation detects these vnodes one of two actions are performed. If the migration was from outside the cluster, the vnode will be marked as not eligible for ".." evaluation. If this is the case ".." evaluation returns with an error. If the vnode is eligible for ".." evaluation, the new vnode operation gfh2vnode() is called to create the mount point. This code change is made in the AIX 3.1 base.

3.1.2.3.3.1.2 Detailed Design

```
if(vnode marked mount point without mounted on vnode)
{
    if(vnode marked not eligible for ".." evaluation)
    {
        go to ENOENT processing
    }

    give cluster mount server mounted mount id get back gfh
        for mounted on vnode

    call global file handle to vnode(mounted on gfh, address
        of routine to get info from cluster mount server)
    if(mount error)
    {
        go to ENOENT processing
    }
    clear vnode marked mount point without mounted on vnode
}
```

3.1.2.3.4 Data Structures

3.1.2.3.4.1 Additions

3.1.2.3.4.1.1 global file handle

```
/*
 * global file handle
 */
typedef struct
{
        u_short gfh_type;       /* identifier vfs type */
        u_int   gfh_mntid;      /* mount id */
        caddr_t *private_data;  /*
                                 *   For NFS:
                                 *    1) file handle
                                 *    2) useful vnode fields
```

```
                                        *       a) v_type
                                        *       b) v_flags
                                        *       c) v_rdev
                                        */
} nfs_gfh;
/*
 * types of global file handles
 */
#define GFH_TYPE_NFS    0x0001
... other file system types
```

3.1.2.3.4.2 NIDL Definitions

3.1.2.3.4.3 Changes

struct vnodeops in /usr/include/sys/vnode.h

Add the gfh2vnode() operation.

3.1.2.4 Migration of NFS file locks

3.1.2.4.1 Overview

NFS file locks are supported outside of the normal NFS protocol by two daemons — rpc.lockd and rpc.statd. This document outlines the changes required to support process migration of NFS file locks. These changes are based on the V3.1 NFS code, current Locus NFS code and Sun NFS code.

There will be two possible states for each file after process migration. If the NFS server is a **FUSION** node, the local node will access the NFS server directly. For non-**FUSION** NFS servers, the original locking node will provide access to the NFS server.

3.1.2.4.1.1 Lock Migration

Two additional vnode operations nfs_prepare_to_relock() and nfs_relock() are added to migrate locks. The nfs_prepare_to_relock() returns an opaque list of info used by nfs_relock() in the local node to recreate the locks.

A new XDR protocol, rlm (Remote Lock Manager), is used to perform lock migration and maintain remote locks. This operates like the nlm (Network Lock Manager) protocol — message passing RPCs with async replies.

3.1.2.4.1.1.1 Lock Migration — FUSION NFS server

Two changes are made to rpc.lockd to support lock migration. First, the **FUSION** server's rpc.lockd is modified to reply to the IP address of the requesting node. Second, the "sysid" field of the lock structure passed to the underlying physical file system will not be derived from the IP connection address. With these changes, the rpc.lockd on any node can directly control the migrated NFS file locks without having to use a surrogate locking node.

3.1.2.4.1.1.2 Lock Migration — non-FUSION NFS server

Generic Sun rpc.lockd uses the client node name that was provided as part of the lock request as a key for locking files and to determine the response path for rpc.lockd messages. To allow existing locks to persist after migration, the local node uses the original locking node as a surrogate locking node. After migration, the lock will exist in the original locking node and the local node. The in-memory tables in both nodes are updated to reflect the surrogate lock.

```
Before migration:                    Lock reference count
FUSION server                                 1<-\
original locking node                         1--/
new node                                      0

After relock:                        Lock reference count
FUSION server                                 2
original locking node                         1
new node                                      1

Close in original locking node:      Lock reference count
FUSION server                                 1<-\
original locking node                         0  |
new node                                      1--/
```

Figure 22. Original locking node with **FUSION** server

Figures 22 through 26 show the lock relationships for the possible node configurations.

Notes for Figures 22 through 26:

1. Arrows represent the path for the lock requests.
2. The new node informs the controlling node (**FUSION** server or original locking node) of the migrated lock. The controlling node is allowed to fail the request if it is involved in any lock reclaim operations.
3. Once the lock has been installed in the new node, either the migrated-from node or the new node will close the file.

3.1.2.4.1.2 Surrogate Locks

Lock operations for surrogate locks are identical to normal locks until the request goes over-the-wire. Before going over-the-wire, a check is make of the owner flag. If the requesting node is not the owner of the lock, the request is passed to the surrogate node using the rlm handler. The rlm handler in the surrogate node locates the existing lock entry in the in-memory table and places the request on the wire. When the reply arrives, the nlm response routine checks the rlm_lock field in the lock and passes control of the lock to either the nlm handler or to the rlm handler. If this is a

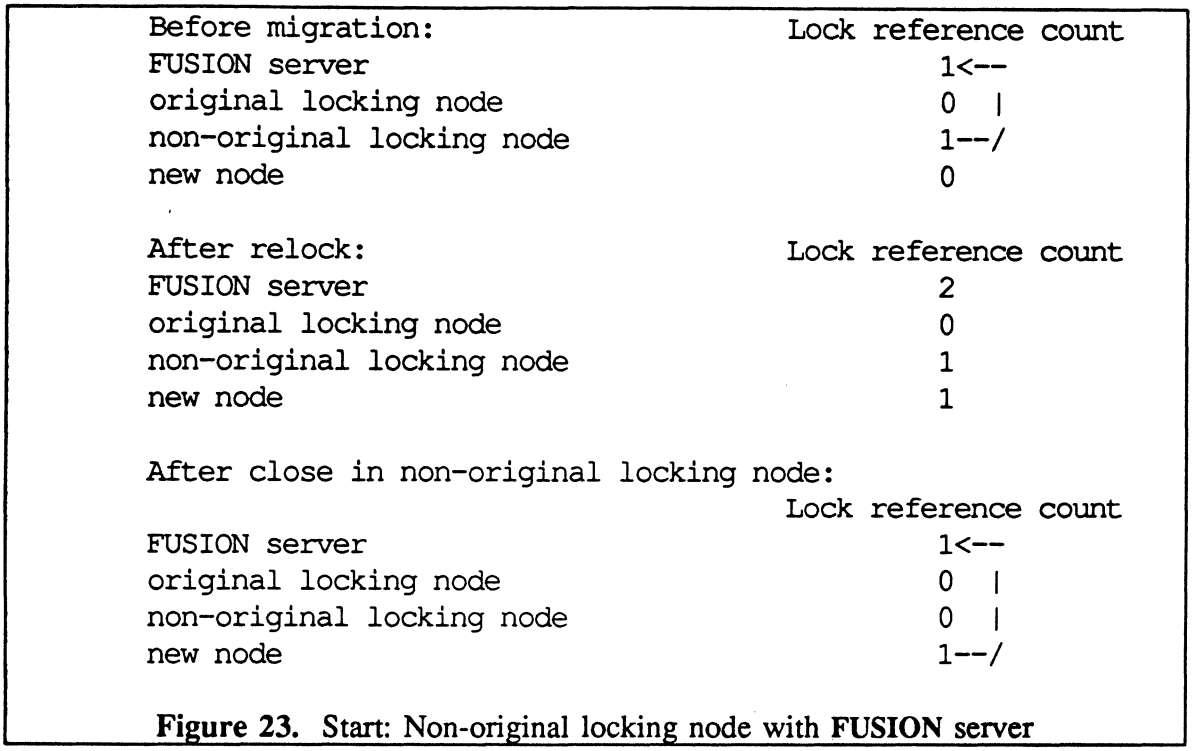


Figure 23. Start: Non-original locking node with **FUSION** server

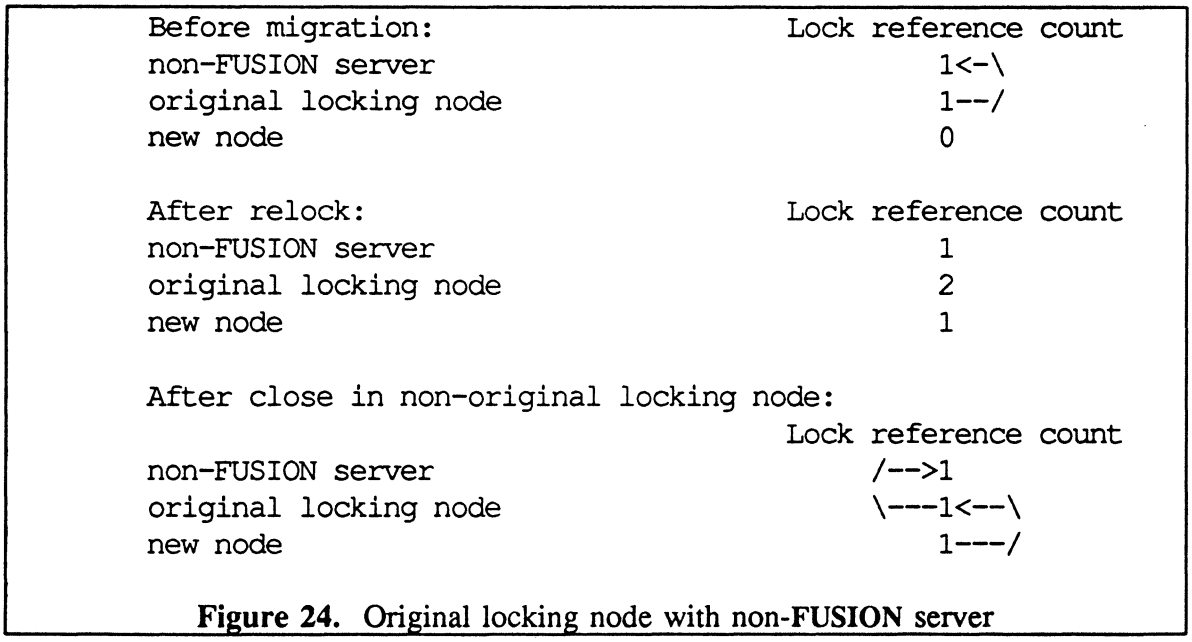


Figure 24. Original locking node with non-**FUSION** server

surrogate lock, control is passed to the rlm handler which sends the replies to the using node, and then continues to process the reply in the surrogate node. The using node's rlm handler passes the response on to the nlm handler for local processing. Processing the surrogate requests in both nodes is used to keep the in-memory lock tables in sync.

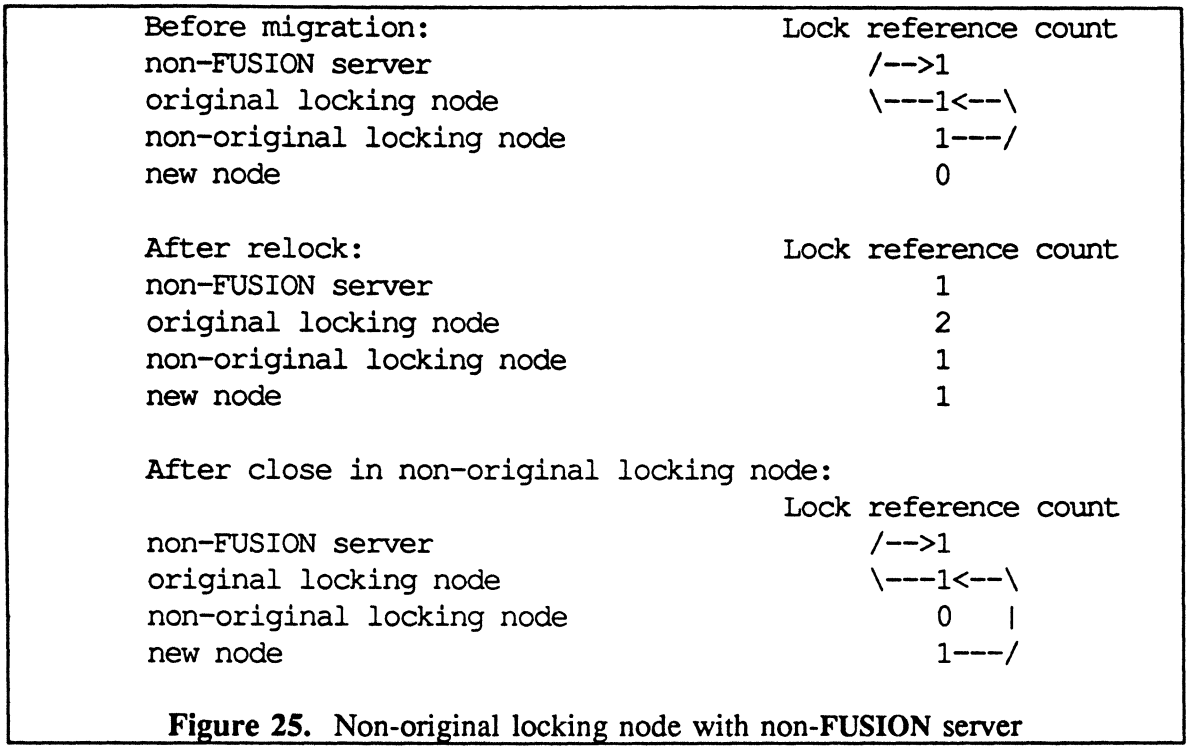


Figure 25. Non-original locking node with non-FUSION server

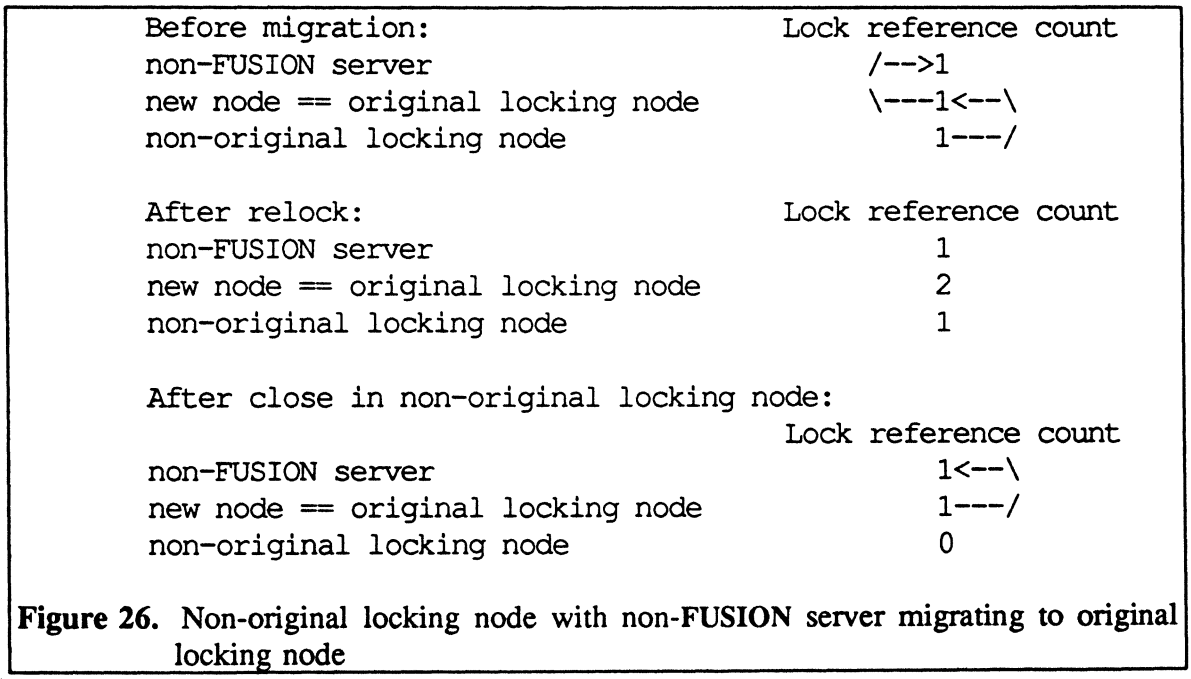


Figure 26. Non-original locking node with non-FUSION server migrating to original locking node

3.1.2.4.1.3 Crash Recovery

3.1.2.4.1.3.1 Current Version

rpc.statd provides crash recovery for NFS servers and clients. Nodes involved in locking operations are kept in /etc/sm directory by node name. On reboot, rpc.statd

notifies all nodes in this directory of the fact. The notified node performs the function indicated in the in-memory monitor tables. When a server reboots, clients are allowed a grace period in which to reclaim their locks. Servers destroy the client's locks when notified of the client's reboot.

3.1.2.4.1.3.2 Changes Required for Surrogate Locks

Surrogate locks require a different crash recovery than that provide in the current version. As part of the migrate process, both the surrogate and migrated-to node added monitor table and /etc/sm directories entries for the opposing node. The surrogate node unlocks on all locks held for the migrated-to node if the migrated-to node reboots. The migrated-to node signals all processes that held locks on the surrogate node if the surrogate node reboots.

3.1.2.4.2 New Functions

3.1.2.4.2.1 nfs_prepare_to_relock(filehandle, reopendata,...)

3.1.2.4.2.1.1 Overview

This the an addition vfs operation for process migration which is called in the migrated-from node to gather the locking information required to relock the vnode in the migrated-to node. Places the info needed to relock the vnode in the reopendata pointer. Lock information is acquired through rpc calls the remote lock manager (rlm) code. Returns:

NOLOCKS no locks held on the vnode

OK reopendata has lock info for vnode

FAIL server or client has lock reclaim is in progress

3.1.2.4.2.1.2 Detailed Design

```
nfs_prepare_to_relock(vp, relockdata, ,...)
{
  /*
   * avoid the rpc call if file has no locks
   */
  if (vnode_has_no_locks)
        return(NOLOCK)
  /*
   * call the local rlm rpc.lockd extension;
   * loop until all locks transferred
   */
  do
  {
      rlm_getlockres = talk2rlm(GETLOCKS, lockreq)

      /*
```

```
         * doing lock reclaim?
         */
        if(lockreply->status == denied)
             return(FAIL)

        /*
         * move data into relockdata buffer
         */
         move_lock_data(lockreq, reopendata)

    } while (!rlm_getlockres.eof)

    return(OK)
}
```

3.1.2.4.2.2 proc_rlm_getlocks(filehandle, pid)

3.1.2.4.2.2.1 Overview

Get the locks for the filehandle owned by pid. This function is called in the migrated-from node as part of the prepare_to_relock() vfs operation. Since a maximum of 8K of data can be transferred by a single rpc call, multiple calls may be required to transfer the entire list of locks. A offset in kept in the record lock struct to indicate the starting position of the next call. proc_rlm_getlocks() returns either a list of xdr's locks or a denied status if this node is involved in lock reclaim operations. This code resides in rpc.lockd.

3.1.2.4.2.2.2 Detailed Design

```
proc_rlm_getlocks(filehandle, pid)
{
    /*
     * skip forward to reclock.offset
     * copy locks into reclock struct
     * adjust offset for next call
     * set eof if last lock found
     */
}
```

3.1.2.4.2.3 proc_rlm_putlocks(lock_list)

3.1.2.4.2.3.1 Overview

This function is called in the migrated-to node to setup file locks as part of the relock vfs operation. The following operations are performed:

1. 1) updates in-memory lock tables
2. 2) checks resources (access to servers, locking nodes, etc)

3. 3) informs either the **FUSION** server or the original locking node of the migrated lock
4. 4) adds either the **FUSION** server or the original locking node to the monitor table

Results are returned by calling rlm_relock_reply(). This operation will fail if resources are unavailable or a lock reclaim operation is in progress. This code resides in rpc.lockd.

3.1.2.4.2.3.2 Detailed Design

```
proc_rlm_putlocks(lock_list)
{

     /*
      * add locks from remote node to in-memory lock tables
      */
     if ((reclock = rlm_relock_add(lock_list)) == FAIL)
     {
          results = FAIL
          rlm_relock_reply(results)
     }
     /*
      * check state of remote nodes
      */
     if ((results = rlm_relock_check(reclock)) == FAIL)
     {
          nfs_relock_cleanup(reclock)
          rlm_relock_reply(results)
     }
     /*
      * inform the controlling node (either the FUSION NFS server or
      * the original locking node) of the migration.
      * Add the controlling node to the this node's list of monitor nodes.
      */
     if (rlm_getsrvtype() == FUSION_SRV)
     {
          if ( (results = rlm_call(server, MIGRATED_LOCK)) == FAIL)
          {
              nfs_relock_cleanup(reclock)
              klm_relock_reply(results)
          }
          rlm_add_monitor(server, reclock)
    }
    else
```

```
    {
          if ( (results  = rlm_call(original_node, MIGRATED_LOCK)) == FAIL)
          {
              nfs_relock_cleanup(reclock)
              klm_relock_reply(results)
          }
          rlm_add_monitor(original_node, reclock)
    }
    klm_relock_reply(results)
}
```

3.1.2.4.2.4 rlm_relock_add(lock_list)

3.1.2.4.2.4.1 Overview

Add the list of locks received from the migrated-from node to in-memory lock tables. Called by proc_rlm_putlocks() as part of the relock vfs operation. Returns OK or the error value returned by either blocked() or add_reclock(). This code resides in rpc.lockd.

3.1.2.4.2.4.2 Detailed Design

```
rlm_relock_add(lock_list)
{
        struct fs_rlck *fp;
        reclock *insrtp;
        reclock a;

        /*
         * loop thru the lock_list
         * placing info from current list entry in the reclock
         * then call blocked() then add_reclock()
         */
        while(lock_list && !error)
        {
            extract_lock_info_from_list
            error = blocked(&fp, &insrtp, &a)
            if( !error)
                error = add_reclock(fp, insrtp, &a)
            lock_list = next_entry
        }
        return(error)
}
```

3.1.2.4.2.5 rlm_add_monitor(node, reclock)

3.1.2.4.2.5.1 Overview

Add a node to the monitor table. This function is called in the migrated-to node by proc_rlm_putlocks() to set a status monitor entry for the controlling node of a reclock. The pointer to the private data record is set to NULL so this entry will not be process by the prot_priv_crash(). This code resides in rpc.lockd.

3.1.2.4.2.5.2 Detailed Design

```
rlm_add_monitor(node)
 {
 }
```

3.1.2.4.2.6 nfs_relock_cleanup(reclock)

3.1.2.4.2.6.1 Overview

Free all resources allocated during proc_rlm_putlocks(). Called when the relock operation fails. This code resides in the rpc.lockd.

3.1.2.4.2.6.2 Detailed Design

```
nfs_relock_cleanup(reclock)
{
    /*
     * XXX
     */
}
```

3.1.2.4.2.7 rlm_prog(Rqstp, Transp)

3.1.2.4.2.7.1 Overview

Service program for the rlm xdr protocol. This will service all the requests needed to run the rlm_prog().

3.1.2.4.2.7.2 Detailed Design

```
rlm_prog(Rqstp, Transp)
    struct svc_req *Rqstp;
    SVCXPRT        *Transp;
{
    reclock        *req;
    remote_result  *reply;
    char           *(*Local)();
    int            oldmask;

    oldmask = sigblock (1 << (SIGALRM - 1));
    switch (Rqstp->rq_proc)
    {
        /*
         * set xdr_Argument, xdr_Result, and routine to process
         * the request
```

```
         */
    }

    if (surrogate_request)
    {
        /*
         * pass surrogate lock requests to network lock manager
         */
    }
    if (surrogate_response)
    {
        /*
         * pass surrogate lock responses to local processing
         */
    }

    if (relock_request)
    {
        /*
         * pass relock requests to local handler
         */
    }
    if (relock_response)
    {
        /*
         * do local processing of rlm replies
         * then pass response to nlm handler
         */
    }
}
```

3.1.2.4.2.8 rlm_res_routine(reply, local_cont)

3.1.2.4.2.8.1 Overview

Surrogate lock response handler. Called in the original locking node when the nlm response arrives from the NFS server. Builds the response message for the migrate-to node and continues the lock processing in the original locking node. This code resides in rpc.lockd.

3.1.2.4.2.8.2 Detailed Design

```
rlm_res_routine(reply, local_cont)
    remote_result   *reply;
    remote_result   *(*local_cont)();
{
    reclock   *lckp;
```

```
    lckp = find_msg(reply)

    /*
     * build rlm response
     */
    *local_cont(lckp)          /* continue local operation */
    rlm_reply(,,,)             /* forward reply to migrated-to node */
}
```

3.1.2.4.2.9 rlm_xtimer()

3.1.2.4.2.9.1 Overview

Retransmit timed-out rlm messages in queue. Called by xtimer() before it does any processing. Functions like xtimer except rlm locks do not expire. This code resides in rpc.lockd.

3.1.2.4.2.9.2 Detailed Design

```
rlm_xtimer()
{
    /*
     * process all messages in the msg_q that belong to rlm
     * see xtimer() for details.
     */
}
```

3.1.2.4.2.10 rlm_relock_check(reclock)

3.1.2.4.2.10.1 Overview

Check resources needed to complete a lock migration. Called in the migrated-to node as a result of the nfs_relock() vnode operation. This code resides in rpc.lockd.

3.1.2.4.2.10.2 Detailed Design

```
rlm_relock_check(reclock)
{
    reclock *lckp

    /*
     * fail if the migrated-to node involved in lock reclaim
     */
     if (doing_reclaims)
         return(FAIL)

    /*
     * fail if any node's rpc.lockd is inaccessible
     */
    if (rlm_check_network(servername) == FAIL)
```

```
}
            return(FAIL)

        if (!FUSION_NFS_server)
            if (rlm_check_network(original_locking_node) == FAIL)
                return(FAIL)

        return(PASS)
    }
```

3.1.2.4.2.11 rlm_check_network(node)

3.1.2.4.2.11.1 Overview

Check for access to the node's rpc.lockd. Called in the migrated-to node as a part of the nfs_relock() operation. This code resides in rpc.lockd.

3.1.2.4.2.11.2 Detailed Design

```
    rlm_check_network(node)
        char *node
    {
        /*
         * scan the in-memory lock tables
         * for a reference to this node
         * if found, return OK
         */
        if(node_in_locktables)
            return(OK)
        /*
         * no locks held for this node,
         * see if the node's rpc.lockd is running
         */
        else
            return(rlm_talk2node(node, NLM_PROG, NLM_VERS))
    }
```

3.1.2.4.2.12 find_re(a)

3.1.2.4.2.12.1 Overview

Find an entry in the rlm lock table. This code resides in rpc.lockd.

3.1.2.4.2.12.2 Detailed Design

```
    struct fs_rlck *
    find_re(a)
        reclock *a
    {
            /*
             * see find_fe()
```

```
        */
}
```

3.1.2.4.2.13 find_re(a)

3.1.2.4.2.13.1 Overview

Add entry to remote lock table. This code resides in rpc.lockd.

3.1.2.4.2.13.2 Detailed Design

```
void
insert_re(rp)
    struct fs_rlck *rp
{
        /*
         * see insert_fe()
         */
}
```

3.1.2.4.2.14 release_re()

3.1.2.4.2.14.1 Overview

Release remote lock table entry. This code resides in rpc.lockd.

3.1.2.4.2.14.2 Detailed Design

```
release_re()
{
        /*
         * see release_fe()
         */
}
```

3.1.2.4.2.15 rlm_reply(proc, reply, reclock)

3.1.2.4.2.15.1 Overview

Send remote lock manager reply over-the-wire. Called in the original locking node when a nlm reply arrives for an rlm request. Called in the original locking node or FUSION server to reply to to a migrate request. This code resides in rpc.lockd.

3.1.2.4.2.15.2 Detailed Design

```
rlm_reply(proc, reply, reclock)
    int           proc;
    remote_result *reply;
    reclock       *a;

{
    /*
     * build network reply
     * see nlm_reply()
```

```
        */
       udp_call(....)
}
```

3.1.2.4.2.16 rlm_call(proc, reclock, retransmitted)

3.1.2.4.2.16.1 Overview

Send remote lock manager request over-the-wire. Called in the migrated-to node when surrogate locks are inuse or as part the nfs_relock(). Adds the request to message queue if first call. This code resides in rpc.lockd.

3.1.2.4.2.16.2 Detailed Design

```
rlm_call(proc, reclock, retransmitted)
{
    /*
     * build_network_request
     * see nlm_call()
     */

    udp_call()

    if (!retransmitted)
       queue(reclock, proc)
}
```

3.1.2.4.2.17 rlm_locate_surrogate(reclock)

3.1.2.4.2.17.1 Overview

Search the table_fp for a matching lock entry. Called by rlm_prog() when a surrogate request arrives. Sets rlm_lock in the lock entry. Returns a pointer to the matching lock or NULL if matching entry not found. This code resides in rpc.lockd.

3.1.2.4.2.17.2 Detailed Design

```
relock *
rlm_locate_surrogate(reclock)
{
}
```

3.1.2.4.2.18 rlm_locate_request(reclock)

3.1.2.4.2.18.1 Overview

Search the msg_q for a matching lock entry. Called by rlm_prog() in the migrated-to node when a surrogate response arrives. Returns a pointer to the matching lock or NULL if matching entry not found. This code resides in rpc.lockd.

3.1.2.4.2.18.2 Detailed Design

```
relock *
rlm_locate_request(reclock)
```

```
{
    /*
     * XXX
     */
}
```

3.1.2.4.2.19 proc_rlm_migratelock(reclock)

3.1.2.4.2.19.1 Overview

Mark in-memory lock as migrated. Increments the reference count for this lock and setups a monitor entry for migrated-to node. Called in either the **FUSION** server or original locking node as part of nfs_relock() vfs operation. Sends results to migrate-to node by calling rlm_reply(). This operation will fail if the node is involved in a reclaim operation. This code resides in rpc.lockd.

3.1.2.4.2.19.2 Detailed Design

```
proc_rlm_migratelock(reclock)
{
    if (doing_reclaim)
    {
        results = FAIL
        rlm_reply(,,,)
    }
    reclock->ref_cnt++
    rlm_add_monitor(calling_node);
    results = PASS;
    rlm_reply(,,,);
}
```

3.1.2.4.2.20 rlm_priv_crash(statp)

3.1.2.4.2.20.1 Overview

Process any remote lock entries in file table for the rebooted client. Called from prot_priv_crash() after rpc.statd notices that a client has rebooted. All locks for all locks for which this node is the original locking node and are inuse by the rebooted client are unlocked. A signal is send to all processes which using the rebooted client as a surrogate locking node.This code resides in rpc.lockd.

3.1.2.4.2.20.2 Detailed Design

```
rlm_priv_crash(statp)
{
    /*
     * XXX
     */
}
```

3.1.2.4.2.21 rlm_getsrvtype(server)

3.1.2.4.2.21.1 Overview

Returns the server type of "server" held in the vfs mount info for the server. Locates the vfs for the server and checks the vfs mount info server type field. If the field is UNK_SRV, then a call to rlm_talk2node() is made to setup the server type in the mount info of the vfs. Returns the server type held in the vfs mount info.

3.1.2.4.2.21.2 Detailed Design

```
rlm_getsrvtype(server)
    char *server;
{
        locate vfs for server

        if server type in vfs mount info == UNK_SRV
        {
           if (rlm_talk2node(server, NLM_PROG, NLM_VER) == OK
               set vfs server type to FUSION_SRV
           else
               set vfs server type to STD_SRV
        }
        return vfs server type
}
```

3.1.2.4.2.22 rlm_talk2node(node, prognum, vers)

3.1.2.4.2.22.1 Overview

Attempt to talk to "node" with rpc "prognum" and "vers". Called to determine if the server's rpc.lockd support FUSION extensions (rlm_prog) or as part of nfs_relock() to check access to a NFS server's rpc.lockd. Returns FAIL node didn't respond to the XDR protocol else returns OK. This code resides in rpc.lockd.

3.1.2.4.2.22.2 Detailed Design

```
rlm_talk2node(node, prognum, vers)
        char *node;
        int  prognum;
        int  vers;
{
        /*
         * XXX
         */
}
```

3.1.2.4.3 Changes to existing functions

```
File:          prot_proc.c
Routine:       nlm_call()
```

```
Change:     Allow rlm_call() to process call if not owner of lock.

File:       prot_pnlm.c
Routine:    nlm_res_routine()
Change:     Allow rlm_res_routine() to process response if not user
            of lock.

File:       prot_msg.c
Routine:    xtimer()
Change:     Allow rlm_xtimer() to preprocess message queue.

File:       prot_libr.c
Routine:    map_kernel_klm()
Change:     Set user and owner fields in reclock.

File:       prot_msg.c
Routine:    queue()
Change:     Only set klm_msg if !rlm_active.

File:       prot_priv.c
Routine:    prot_priv_crash()
Change:     Allow rlm_priv_crash() to preprocess entries.

File:       prot_lock.c
Routine:    delete_reclock()
Change:     Setup the user and owner any new
            lock entries created by delete_reclock().

File:       prot_main.c
Routine:    nlm_prog()
Change:     Get the node name of the connection address.  Place
            this name in the "reply_to" field of the lock structure.
Change:     Set the req->sysid to a constant value.
Routine:    main()
Change:     Add startup code for rlm protocol service.

File:       prot_pnlm.c
Routine:    nlm_reply()
Change:     Use the "reply_to" field instead of the "clnt" field of lock
            structure when generating the name to pass to udp_call().
```

3.1.2.4.4 Data Structures

3.1.2.4.4.1 Changes

3.1.2.4.4.1.1 reclock structure additions in nfs/com/cmd/etc/rpc.lockd/prot_lock.h

```
bool_t   owner                  /* client is the original lock node */
bool_t   user                   /* client is user of lock */
bool_t   rlm_lock               /* this lock inuse by rlm */
u_char   refcnt                 /* number of lockds holding locks on
                                 * this lock */
char     *reply_to              /* name of client node to send
                                 * reply */
```

3.1.2.4.4.1.2 mntinfo structure additions in /usr/include/nfs/nfs_clnt.h

```
/*
 * server type used by rpc.lockd
 * to determine if the NFS server
 * supports is running a FUSION rpc.lockd
 */
u_int   mi_srvtype      :2;
#define UNK_SRV         0x0
#define FUSION_SRV      0x1
#define STD_SRV         0x2
```

3.1.2.4.4.2 Additions

3.1.2.4.4.2.1 Globals

```
fs_rlck table_re[HASH_SIZE]     /* table of files being relocked */
int     rlm_active              /* current operation is for rlm */
int     rlm_msg                 /* last message processed by rlm */
```

3.1.2.4.4.2.2 rlm protocol

```
/*
 * Over-the-wire protocol used between the remote lock managers
 * and kernel to local remote lock manager.
 * All the nlm message passing style request plus:
 *        RLM_MIGRATE_MSG
 *        RLM_MIGRATE_RES
 *        RLM_GETLOCKS
 *        RLM_PUTLOCKS
 */

program RLM_PROG {
        version RLM_VERS {

                /*
                 * message passing style of requesting lock
                 * (same as the nlm protocol)
```

```
                */
                void            RLM_TEST_MSG(struct rlm_testargs) = 1;
                void            RLM_LOCK_MSG(struct rlm_lockargs) = 2;
                void            RLM_CANCEL_MSG(struct rlm_cancargs) =3;
                void            RLM_UNLOCK_MSG(struct rlm_unlockargs) = 4;
                void            RLM_GRANTED_MSG(struct rlm_testargs) = 5;
                void            RLM_MIGRATED_MSG(struct rlm_migrateargs) = 6;
                void            RLM_TEST_RES(rlm_testres) = 7;
                void            RLM_LOCK_RES(rlm_res) = 8;
                void            RLM_CANCEL_RES(rlm_res) = 9;
                void            RLM_UNLOCK_RES(rlm_res) = 10;
                void            RLM_GRANTED_RES(rlm_res) = 11;
                void            RLM_MIGRATED_RES(rlm_res) = 12;
                /*
                 * Migration rpc style requests
                 * Kernel <-> local remote lock manager
                 */
                rlm_getlockres RLM_GETLOCKS(struct rlm_getlockargs) = 13;

        } = 1;
} = ??????;

#define LM_MAXSTRLEN     1024

/*
 * status of a call to the lock manager
 */
enum rlm_stats {
     rlm_granted = 0,
     rlm_denied = 1,
     rlm_denied_nolocks = 2,
     rlm_blocked = 3,
     rlm_denied_grace_period = 4
};

struct rlm_holder {
     bool exclusive;
     int svid;
     netobj oh;
     unsigned l_offset;
     unsigned l_len;
};
```

```
union rlm_testrply switch (rlm_stats stat) {
     case rlm_granted:
          void v;              /* the lock is 'grantable' */
     case rlm_denied:
          struct rlm_holder;
     case rlm_denied_nolocks:
          void v;
     case rlm_blocked:
          void v;
     case rlm_denied_grace_period:
          void v;
};

struct rlm_stat {
     rlm_stats stat;
};

struct rlm_res {
     netobj cookie;
     rlm_stat stat;
};

struct rlm_testres {
     netobj cookie;
     rlm_testrply stat;
};

struct rlm_lock {
     string caller_name<LM_MAXSTRLEN>;
     netobj fh;                    /* identify a file */
     netobj oh;                    /* identify owner of a lock */
     int svid;                     /* generated from pid for svid */
                                   /* XXX missing in AIX 3.1 */
     unsigned l_offset;
     unsigned l_len;
};

struct rlm_cancargs {
     netobj cookie;
     bool block;
     bool exclusive;
     struct rlm_lock lock;
};
```

```
struct rlm_testargs {
     netobj cookie;
     bool exclusive;
     struct rlm_lock lock;
};

struct rlm_unlockargs {
     netobj cookie;
     struct rlm_lock lock;
};

struct rlm_migratedargs {
     netobj cookie;
     struct rlm_lock lock;
     bool reclaim;              /* used for recovering locks */
     int state;                 /* specify local status monitor state */
};

/*
 * Arguments to getlocks
 */
struct rlm_getlockargs {
        netobj fh;
        int pid;
        netobj cookie;
        unsigned count;
};

struct rlm_entry {
        unsigned l_offset;
        unsigned l_len;
        rlm_entry *nextentry;
};

struct rlm_locklist {
        struct rlm_lock lock;
        rlm_entry *entries;
        bool eol;
};

union getlockres switch (rlm_stat status) {
     case rlm_granted:
        rlm_locklist reply;
```

```
    case rlm_denied:
        void v;
};

/*
 * Arguments to putlocks
 * relock extension
 */
struct rlm_putlocksargs {
        rlm_locklist entries;
};



/*
 * protocol used between the UNIX kernel (the "client") and the
 * local lock manager FUSION remote locking extension (rlm) .
 */

/*
 * lock manager status returns
 */
enum krlm_stats {
        krlm_granted = 0,        /* lock is granted */
        krlm_denied = 1,         /* lock is denied */
        krlm_denied_nolocks = 2, /* no lock entry available */
};

/*
 * lock manager lock identifier
 */
struct krlm_lock {
        string server_name<LM_MAXSTRLEN>;
        netobj fh;               /* a counted file handle */
        int pid;                 /* holder of the lock */
                                  * zero means through end of file */
        unsigned l_len;          /* byte length of the lock;
        unsigned l_offset;
};

/*
 * reply to KLM_RELOCK
 */
```

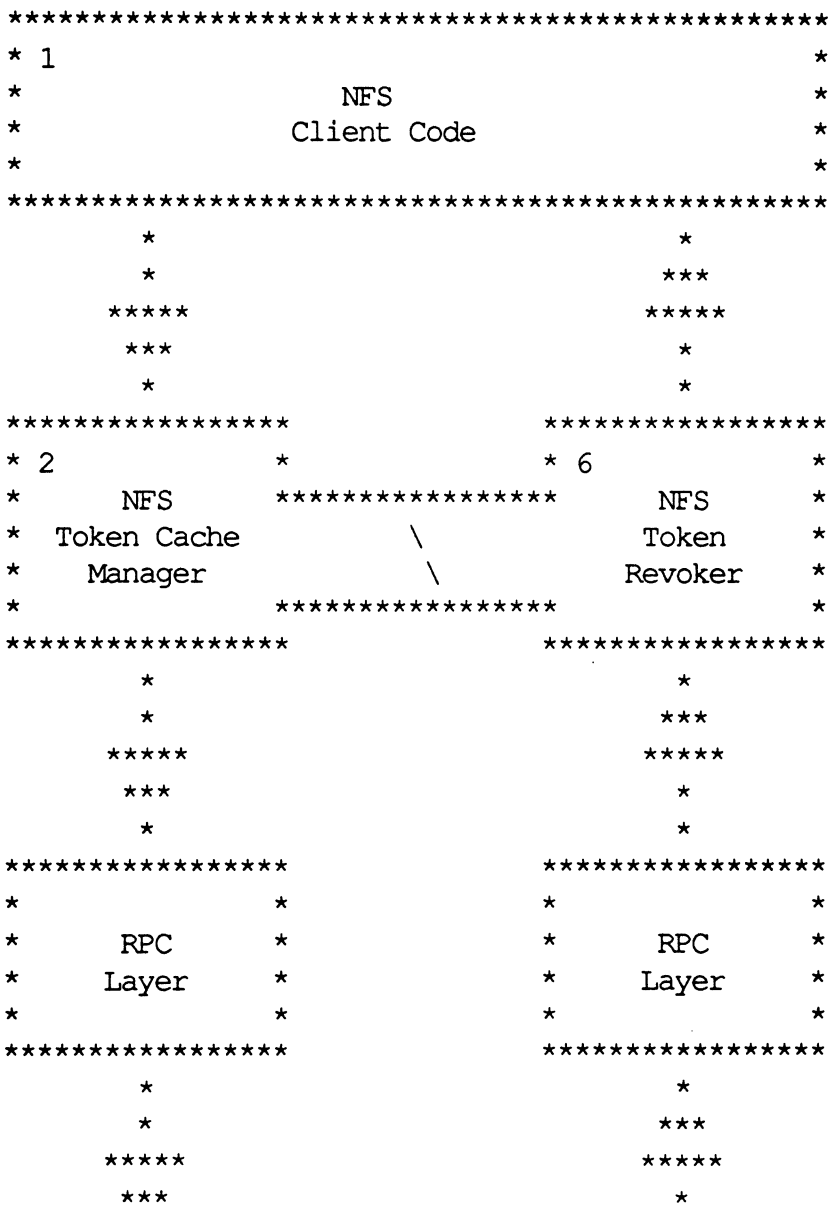
```
struct krlm_stat {
        krlm_stats stat;
};
```

3.1.2.5 Migration and secure NFS

XXX

3.1.2.6 NFS Cache Coherency

3.1.2.6.1 Overview



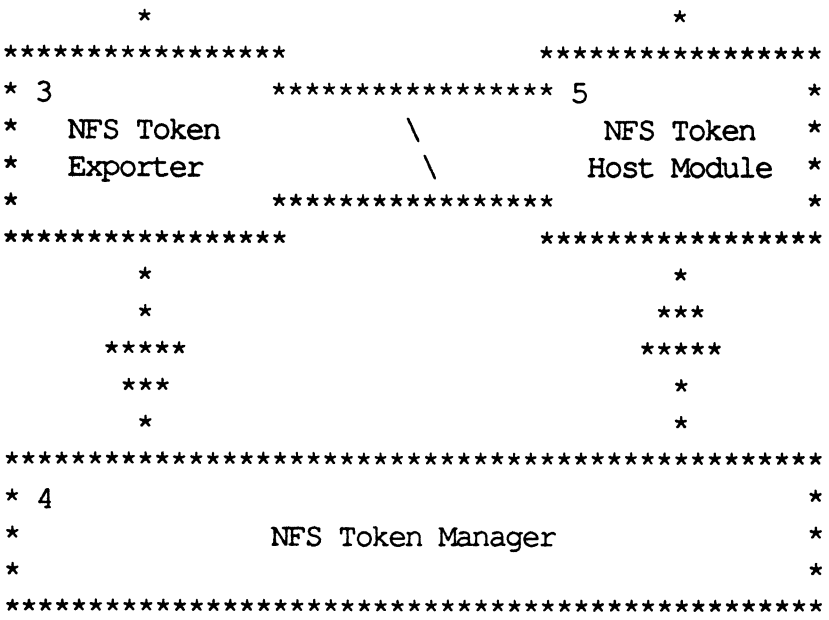


Figure 27. NFS Tokens Block Diagram

NFS being stateless in design precludes having both client caching of data and having a consistent view of the data on all clients. Also as many operations require more than a single RPC call disabling of client caching results in the data only being mostly consistent. In order to provide the required data consistency a method of providing guarantees to the various clients about data they hold in the local caches must be provided. This is accomplished by having a central registry of these guarantees. These guarantees are referred to as tokens and the central registry is referred to as the NFS token exporter. The implementation and use of these tokens has six major functional pieces. These are shown in Figure 27.

3.1.2.6.1.1 NFS Client code

The existing NFS client code has to be modified to request the appropriate tokens before any operation. In some cases when the token is granted additional information will be returned in order that cached data which is still valid does not have to be flushed. In cases where multiple tokens are required it may be necessary upon deadlock detection to release tokens which have been acquired. It will then be necessary to re-execute some or all of the vnode operation when deadlocks are detected.

3.1.2.6.1.2 NFS Token Cache Manager

The NFS client code calls the NFS token cache manager when a token is required. A check is done to see if the token is locally held and if not the token is requested from the NFS token exporter. When the client code returns a token it is marked as being unused. Unused tokens are returned to the NFS token exporter when a revoke request is received or the NFS token cache manager needs to reclaim storage space. The NFS

token cache manager calls the NFS client code to have any data flushed before returning tokens to the NFS token exporter.

3.1.2.6.1.3 NFS Token Exporter

Each NFS mounted file system within a cluster has a single token exporter. The NFS token exporter in called by the NFS cache managers to get tokens for a particular file handle. If the file handle is not known about a token structure will be created. The NFS token exporter then asks the NFS token manager for the appropriate tokens. If the token is granted it along with any required time stamps are returned to the NFS cache manager. If the NFS token manager returns an error this is passed along to the cache manager.

3.1.2.6.1.4 NFS Token Manager

This will be supplied by DCE. It is called by the NFS token exporter to get tokens for a particular file handle. The NFS token manager will call the NFS host module to revoke (call back) tokens from the various nodes within the cluster when the requested token conflicts with tokens which have already been granted.

3.1.2.6.1.5 NFS host module

This module shares both data and code with the NFS token exporter. The NFS host module is responsible for getting tokens back from the various clients. This module is planned to be heavily built on DCE code. Until this code is functional, exact detail of operation can not be specified at this time. However the function it will provide can be specified.

3.1.2.6.1.6 NFS token revoker

This module shares both data and code with the NFS token cache manager. When the NFS host module requests a token back the NFS token revoker will in general return the token if it is not busy or wait for some short period (seconds) for it to become unbusy. If it is still busy it will return a error indicating that the NFS server is down. If there are dirty buffers associated with the token the NFS token revoker will schedule the I/O on the buffers in a manner which guarantees that the NFS token revoker will not get put to sleep waiting for the I/O to complete.

3.1.2.6.1.7 Token time stamps

In a typical use of tokens no time stamps would be required. When a guarantee about some item was needed a token would be gotten and when that guarantee was no longer needed the token would be released. There is however a generic problem with this approach. Any data which is gotten by the same action which gets the handle to talk about the data is useless. An example of this is the NFS lookup RPC call. The two basic arguments to the RPC call are a file handle and a string. The file handle references the directory in which to lookup the string. What is returned from the lookup RPC is a file handle for the file which was looked up and stat information about the file. However as no token is held for the looked up item other nodes may have already already altered the stat data. To circumvent this problem time stamps are associated with the various tokens. The client saves the current time before doing the

lookup RPC. After the RPC is complete the client asks the NFS token exporter for a stat token. The token exporter returns the token along with how far in the past the token can be guaranteed. If this is prior to the lookup RPC call then it is known that the information returned with the lookup is still valid. If the time is after the RPC call started then the file stat information can not be trusted.

3.1.2.6.1.8 Time synchronization

It is unlikely that the clocks on the various nodes will be synchronized well enough to pass around absolute time data. Also an unknown amount of time will pass transporting the data from one node to another. This unknown amount of time could be in minutes if there are communication problems. For this reason the time stamps are not absolute times but deltas into the past that a token was good. If the token is delayed by communication problems the time delta into the past may no longer be large enough to guarantee the token. This results is the client having to talk to the NFS server again. The time used to reference the tokens should be a time since boot rather than one of the operator settable clocks.

3.1.2.6.1.9 NFS Client code

The following rules must be followed to obtain correct operation with variable RPC delays. These delays occur both between the NFS client and the NFS server and the NFS client and the NFS token exporter.

3.1.2.6.1.9.1 Stat time reference to beginning of NFS RPC.

When getting stat information from an NFS server the stat information that is returned could be from any time after the RPC call is made. In order to insure that the data hasn't changed after the RPC reply it must be assumed that the stat information was put in the packet at the same instant that the request was made. When the stat token is gotten in order for the stat data to be valid the guarantees must extend back until at least when the RPC call was made.

3.1.2.6.1.9.2 Mod time referenced to end of NFS RPC.

When a RPC call is made to an NFS server which changes the stat information the time that the call completed must be saved. This guarantees that when another client gets stat information it will be considered invalid unless it occurred after the RPC call which updated the vnode completed, and therefore after the server saved the updated stat information.

3.1.2.6.1.9.3 Lookup time reference to end of NFS RPC.

The first time a lookup RPC completes on a file handle the time is save in with the token information for the file handle the lookup was done in. When the lookup token is surrendered the directory name lookup cache (dnlc) is not flushed. If the lookup token is reacquired for the directory the dnlc will be flush if the token time is after the time saved. The time of completion can be save as lookups and deletes are conflicting tokens and can not happen at the same time.

3.1.2.6.1.9.4 Lookup time referenced to start of NFS RPC

The first time a delete RPC is done on a file handle the time is save with the delete token information. When the delete token is returned this time will also be returned. This time will be used when a lookup token is gotten to determine if the dnlc needs to be flushed.

3.1.2.6.1.9.5 Lookup time not really needed

There is nothing about the lookup function which prohibits using a standard token mechanism for flushing the dnlc. The lookup token could be gotten whenever a lookup in a directory was going to occur. It could then be held until some other node did a delete which required the token to be returned. At that time the dnlc would be flushed. This would result in virtually the same performance as is achieved using lookup time. The use of time stamps to flush the dnlc leaves a easy path for future performance improvement. It would work basically as follows:

1. Return file handles of entries deleted from a directory when returning the delete token for the directory.
2. The NFS token exporter also passes the file handles and time stamps of recently deleted directory entries when giving out the lookup token.
3. Token cache manager causes dnlc entries which match up with file handles gotten with lookup token to be flushed.

3.1.2.6.2 NFS client code / NFS token cache manager interface

3.1.2.6.2.1 nfs_gettoken(vp, token, flag, result)

3.1.2.6.2.1.1 Overview

NFS client code request for token(s) from the NFS cache token manager.

3.1.2.6.2.1.2 Detailed Design

```
nfs_gettoken(vp, token, flag, result)
{
        /*
         * XXX
         */
}
```

3.1.2.6.2.2 nfs_puttoken(vp, token, flag, result)

3.1.2.6.2.2.1 Overview

NFS client returns token(s) to NFS token cache manager.

3.1.2.6.2.2.2 Detailed Design

```
nfs_puttoken(vp, token, flag, result)
{
        /*
         * XXX
```

```
        */
}
```

3.1.2.6.2.3 nfs_flushdata(vp)

3.1.2.6.2.3.1 Overview

NFS token cache manager requests NFS client to flush data cache.

3.1.2.6.2.3.2 Detailed Design

```
nfs_flushdata(vp)
{
        /*
         * XXX
         */
}
```

3.1.2.6.2.4 data structures

`struct vnode`	`*vp`	`vnode to be operated on`
`token_t`	`token`	`token(s) being gotten or returned`
`struct tokresult`	`*result`	`return information from operation`
`int`	`flags`	`bit encoded operation dependent flags`

3.1.2.6.2.4.1 Tokens

LOOKUP This token must be gotten for a directory vnode before using the dnlc or using the lookup RPC on the vnode. This token conflicts with DELETE. When this token is gotten if the time stamp associated with the token is prior to the time stamp for this token keep in the remote node (rnode) any dnlc entries for this vnode must be flushed.

DELETE This token must be gotten for a directory vnode before using a RPC call which will result in an entry being removed or renamed in the directory. The basic purpose of this token is to cause the LOOKUP token to be recalled. This token conflicts with STAT, LOOKUP, RDATA, OPEN, and ADD.

ADD This token must be gotten for directory vnode before using a RPC call which will result in a entry being added to the directory. This token conflicts with STAT, RDATA, DELETE. Because the non existence of files are not cached in the dnlc this token does not conflict with LOOKUP.

STAT This token must be gotten before checking the attribute cache or doing an RPC call to fetch the attributes for a vnode. This token conflicts with ADD, DELETE, RDATA, WDATA.

RDATA This token must be gotten before checking for valid buffer data or doing an RPC call to fetch data for a vnode. This token conflicts with ADD, DELETE, WDATA, STAT.

WDATA This token must be gotten before copying data from use space into a system buffer. This token conflicts with STAT, WDATA, RDATA.

OPEN This token must be gotten when a vnode is to be opened. The OPEN token can only be returned when reference count for the vnode goes to zero. The purpose of this vnode is to:

a. prevent open files from being deleted.

b. prevent in use file systems from being unmounted.

This token conflicts with EXCLUSIVE.

EXCLUSIVE This token must be gotten before a vnode can be unlinked. The purpose of this token is to detect open files. If a delete is attempted and this token can not be gotten then the directory entry must be renamed.

3.1.2.6.2.4.2 flags

This field is a bit encoded catch all which means different things at different times.

MULTI This is used with nfs_gettoken() to indicate that this token request is part of a multi token operation which may deadlock. This information is needed to make deadlock detection easier.

NOCHANGE This flag is used with the nfs_puttoken to indicate that data associate with this token was not changed. When this flag is asserted time stamps associated with this token will not be updated.

vp This is the vnode for which a token is being requested or released.

3.1.2.6.2.4.3 result

There are four pieces of info in result struct. They are:

Deadlock If other locks are held deadlocks may occur. If a possible deadlock is detected the nfs_gettoken() function will return with deadlock asserted. the calling function must then release any locks it has, sleeps and then restart the vnode operation.

Sleeptime This is the amount of time to sleep before trying to reacquire locks after a dead lock was detected or the NFS server is not responding.

Stalecache This indicated that the directory for which a lookup token is being requested has changed since that last time the lookup token was released. It is therefore necessary to flush the directory name lookup cache (dnlc).

Stalestat This indicates that since the last time the stat token was granted for this file or directory the stat info has become invalid. This means that the attribute cache must be flushed for this entry.

3.1.2.6.3 NFS token cache manager / NFS token exporter interface

3.1.2.6.3.1 gettokens(filehandle, tokens, result)

3.1.2.6.3.1.1 Overview

The NFS token cache manager talks to the NFS token exporter though the NFS token RPC layer. This layer can not be described at this time as it is desired to use much of the DCE code for token timeout and renewal. In general the NFS token cache manger request token it does not already hold from the NFS token exporter. The cache manager also receives information about how long in the past the token could have been granted.

3.1.2.6.3.1.2 Detailed Design

```
gettokens(filehandle, tokens, result)
{
        /*
         * XXX
         */
}
```

3.1.2.6.3.1.3 data structures

filehandle — This is an identifier which uniquely identifies this file to the NFS token exporter. Within this field are the NFS file handle and a unique identifier for the mounted file system.

tokens — This field contains the tokens being requested. Multiple tokens may be requested in a single transaction. This field is subdivided into two fields one containing the token which is required and the other contains other tokens the cache manager would like if they are available. These are the same tokens as described in 3.1.2.6.2.

result — This field contains the reply to the get token request. Within it are several sub-fields.

status — This field contains one of four values:

GRANTED	Operation succeeded primary token granted.
DEADLOCK	Operation may be blocked. Release any tokens associated with this process and try again.
NOSERVER	The NFS server appears not to be responding. Wait a while and retry the operation. No requirement to give up any tokens.

GOTOCMS — Node is no longer the NFS token exporter, consult the cluster mount server (CMS) for location of the NFS token exporter.

tokens

These are the tokens which are being granted to the requesting site. These contain the primary token, some or all optional tokens and any other tokens which the cache manager decided to grant. If the primary token could not be granted no tokens will be granted.

deltas

For lookup and stat the length of time in the past the corresponding token could have been granted. This value is NOT an absolute time but the length of time in the past the appropriate token could have been granted. It is mandatory that all times which are on the wire are expressed in deltas rather than absolute time.

3.1.2.6.4 NFS token exporter / NFS token manager interface

3.1.2.6.4.1 Overview

This interface can not be described until the DCE token manager is fully functional. In general, the NFS token exporter will ask the NFS token manager for a particular token or set of tokens for a file handle. The NFS token manager will respond with granted, deadlock, or server down.

3.1.2.6.5 NFS token manager / NFS host module interface

3.1.2.6.5.1 Overview

This interface can not be described in much detail until the DCE token manager is fully functional. In general, the NFS token manager will ask the NFS host module to revoke a set of tokens from a particular node. Possible returns from the NFS token revoker are success or NFS server not responding.

3.1.2.6.6 NFS host module / NFS token revoker interface

3.1.2.6.6.1 nfs_revoketoken(filehandle, token, result)

3.1.2.6.6.1.1 Overview

The NFS host module through the NFS token RPC layer requests a token be revoked. When the token is returned data is also present about how long in the past the token could have been revoked.

3.1.2.6.6.1.2 Detailed design

```
nfs_revoketoken(filehandle, token, result)
{
```

```
        /*
         * XXX
         */
}
```

3.1.2.6.6.1.3 data structures

filehandle — This is an identifier which uniquely identifies this file to the NFS token revoker. This is the same file handle which is described 3.1.2.6.3.

tokens — These are the tokens which need to be revoked. These are the same tokens as described in 3.1.2.6.2.

result — This field contains the reply to the get token request. Within it are several subfields.

- status — This filed contains one of three values:
  - GRANTED — Operation succeeded primary token granted.
  - DEADLOCK — Operation may be blocked by other process release any tokens associated with this process and try again.
  - NOSERVER — The NFS server appears not to be responding. Wait a while and retry the operation. No requirement to give up any tokens.
- tokens — These are the tokens which are being returned to the NFS host module. These contain the requested tokens along with any other the NFS token revoker decided to return.
- deltas — For DELETE, ADD, RDATA, WDATA the length of time in the past that this token could have been revoked. These fields are only valid if the corresponding token is being returned. This value is NOT an absolute time but the length of time in the past the appropriate token could have been returned. Along with the number portion of this field is a Nochange flag. If this flag is asserted the NFS token exporter can ignore the value being returned and retain the time stamp it currently holds.

3.1.2.6.7 NFS token exporter / NFS host module

3.1.2.6.7.1 Overview

This interface can not be described in much detail until the DCE token manager is fully functional. The additional functionality needed for NFS are described below.

In order to for the NFS client to be able to use the stat information supplied from the lookup RPC the NFS token exporter must supply a time stamp which precedes the lookup RPC. This requires that the NFS token exporter be able to provide a time stamp somewhat in the past for files handle that it has no history. To accomplish this the NFS token exporter / NFS host module must remember the most recent time stamp for file handles which have been flushed from its cache.

The following is a short example using stat to show the guarantees which can be made. When the NFS token exporter receives its first request it can guarantee that the stat token that it is giving out was good back to the time when the NFS token exporter started execution. Subsequent request for tokens can have the same guarantee made. Once some tokens have been returned then the time stamp which is given out is either that contained in the token if it exists or the time when the NFS token exporter started execution. At some point it will be necessary to discard tokens which are no longer referenced to reclaim storage. If the time stamp in the tokens is ignored then the NFS token exporter can make guarantees back to the point in time when tokens where discarded. If a token is being discarded has a time stamp five seconds previous then the NFS token exporter can guarantee not only from the current time but also five seconds back from the current time. If at the same time another token is discarded with a 10 second old time stamp it is ignored as the NFS token exporter is already only guaranteeing 5 seconds into the past.

3.1.2.6.8 Detailed Designs

3.1.2.6.8.1 NFS client code

3.1.2.6.8.2 NFS token cache manager

3.1.2.6.8.3 NFS token exporter

3.1.2.6.8.4 NFS token manager

3.1.2.6.8.5 NFS token revoker

3.1.2.6.9 NIDL prototypes

```
const           GFH_MAXDATA      ????

struct gfh_t {
        short   gfh_type;
        int     gfh_mntid;
        char    gfh_data[GFH_MAXDATA];
};

/*
 * Structure used by kernel to store most
 * addresses.
 * from socket.h
```

```
 */
struct nfs_sockaddr {
        short   sa_family;
        char    sa_data[14];
};


/*
 * mount arguments
 * from mount.h
 */
struct nfs_mntargs {
        [out]           struct nfs_sockaddr     *addr,
        [out]           int                     *flags,
        [out]           int                     *wsize,
        [out]           int                     *rsize,
        [out]           int                     *timeo,
        [out]           int                     *retrans,
        [out]           int                     *host_len,
                        char                    [last_is(host_len),
                                                in, out]
                                                hostname[],
        [out]           int                     *acregmin;
        [out]           int                     *acregmax;
        [out]           int                     *acdirmin;
        [out]           int                     *acdirmax;
        [out]           int                     *net_len,
                        char                    [last_is(net_len),
                                                in, out]
                                                netname[],
        [out]           int                     *biods,
        [out]           int                     *st
};

/*
 * get mount info from migrated-from node
 */
int nfs_getmntargs (
        [in]    handle_t                h,
        [in]    int                     mntid,
        [out]   struct nfs_mntargs      *mntinfo,
        [out]   unsigned32              *st
);
```

```
        /*
         * nfs / cluster mount interface
         *  push mount info to CMS
         *   1) mounted-on gfh
         *   2) root gfh
         *   3) mount arguments
         */
        int nfs_cms_pushmntinfo (
                [in]    handle_t                h,
                [in]    struct gfh_t            mnton_gfh,
                [in]    struct gfh_t            root_gfh,
                [in]    struct nfs_mntargs      mntinfo,
                [out]   unsigned32              *st
        );

        /*
         * nfs / cluster mount interface
         *  give CMS global file system id
         *  get mount info
         */
        int nfs_cms_getmntargs (
                [in]    handle_t                h,
                [in]    int                     mntid,
                [out]   struct nfs_mntargs      *mntinfo,
                [out]   unsigned32              *st
        );

        /*
         * nfs / cluster mount interface
         *  give CMS global file system id
         *  get mounted-on global file handle
         */
        int nfs_cms_getmntdongfh (
                [in]    handle_t                h,
                [in]    int                     mntid,
                [out]   struct gfh_t            *gfh,
                [out]   unsigned32              *st
        );

XXX - CMS / token exporter interface
XXX - token cache manager / CMS interface
XXX - token cache manager / token exporter interface
XXX - token host module / token revoker interface
```

XXX - token exporter / token exporter interface

3.2 Invocation Load Balancing

In the initial release of FUSION, exec and rexec will automatically select a node if no node was explicitly specified and either

1. the load module is not executable on the current node
2. the load module is marked in its header indicating that it is "load levelable"

If no node was explicitly specified and neither of those conditions is true, then the program will be executed locally. (Note that you never explicitly specify a node when you use exec, and you may or may not specify a specific node when you use rexec.)

On systems with COFF or XCOFF object file formats, the mark can be put into the header in a new type of section. This section will also contain lists of attributes specifying the needs and expected behavior of the program. An invocation load balancing kernel extension could make use of this information to provide arbitrarily sophisticated load balancing algorithms. On systems with other object file formats, as much of this functionality as possible will be provided.

When automatic node selection is performed, the system will choose from among the intersection of the following sets of nodes:

a. nodes in the Sphere of Interest,

b. nodes in the cluster,

c. nodes on which the program can execute,

d. nodes on which the user is authorized to execute.

Given the set of nodes in this intersection, the NSS is queried to determine the relative loads and speeds of those nodes. Among the 5 (a configurable parameter) fastest nodes, the probability of selecting the node is proportional to "speed / (load + 1)". *

If the selected node is found to be unacceptable (for example, Kerberos denies access), then the search is resumed with that node excluded. The set of nodes on which a user is forbidden to execute is cached so those nodes may be avoided in future searches.

3.3 Dynamic Load Balancing

For long running processes process migration provides the most desirable load balancing mechanism.

This design provides sample dynamic load balancing code and the tools necessary with which a user supplied load balancer can accomplish its task. The most important tool provided is SIGMIGRATE itself. This allows a load balancer to migrate other processes as needed. In addition, the migrate system call allows processes to move themselves.

A load balancer needs information about the expected behavior of processes to be able to achieve a reasonable balancing strategy. The same information from the program header which is used at exec time can also be used by a load balancer. In particular, the mark indicating that processes running this program are load levelable is saved in the vproc to allow load balancers to easily check this. This allows a migrator to choose the best candidates.

Finally, the NSS provides central locations for determining necessary load information. A general interface to NSS allows application programs to consult this server.

In the sample load balancer, processes will only be migrated within their own Spheres of Interest. A run time option should allow the load balancer to decide whether only processes load levelable at execution time will be migrated. Only processes which have already been running for more than a set time threshold should be considered candidates for migration. The sample load balancer will also take care not to migrate the same process more often than a particular frequency.

4. File System Replication Services

4.1 Introduction

Currently DCE offers a read-only form of replication termed "lazy" replication designed for "read-mostly" types of file systems where up-to-date versions are non-essential and updates are done relatively seldom. There are three primary limitations of "lazy" replication:

1. a different path name is necessary to access the read/write replica than that used to access the read-only replicas.
2. the read-only replicas are not guaranteed to be current but simply no more than "n" minutes old. Sometime within "n" minutes, the read/write file system will be cloned and the clone copied to the local storage node, thus updating all files.
3. lazy replicas use episode specific features such as cloning to propagate files so there is some limitation with regard to physical file system implementation.

FUSION provides two additional types of replication: full UNIX semantic (full) which strictly conforms to single site UNIX semantics and "loose" replication which does not. Loose replication is similar to the current DCE lazy replication in that it does not guarantee latest version access for a file. However, DCE lazy replication propagates an entire file system within a specified time period where **FUSION** loose replication asynchronously propagates each file immediately after modifications are complete. In addition, since **FUSION** replication is designed to run on any underlying physical file system including Episode, it does not have the same limitations as lazy with regard to the physical file system on which it resides.

In addition, **FUSION** replication has no fixed limit on the number of file system replicas, the upper limit being determined by system resources. For file access, the server selection will be automatic based on type of access requested and the attributes of the storage node. Changing file servers within a single file access will be transparent to the application.

4.2 Overview

4.2.1 DCE File System

FUSION replication is based on the DCE distributed file system. A cursory knowledge of the DCE File System is helpful in reading this document. The following section briefly describes DCE.

The DCE File System is designed to maximize performance for access to remote files by caching files and performing all operations on the locally cached version, writing any changes to the file system when all modifications are complete. The DCE File System uses a Cache Manager to manage all caching of file data on the client side and a Protocol Exporter which manages file system access on the server side. For local access, the DCE File System uses a Glue Layer which wraps around the vnode ops.

In order to manage remote client access, the Protocol Exporter requires that each client have the appropriate tokens for the file in question. The DCE Token Manager issues tokens to clients and guarantees that no two clients have incompatible tokens by revoking incompatible tokens when a new token request is made. The Glue Layer gets its tokens directly from the Token Manager rather than going through the Protocol Exporter.

The DCE File System also uses a Volume Location Data Base (VLDB) to create and manage file systems. The Cache Manager finds out which server to talk to by obtaining that information from the VLDB. See Figure 28.

4.2.2 FUSION Replication and the DCE File System

Figure 29 shows how **FUSION** Replication works with the DCE File System. The main component of **FUSION** Replication is the **FUSION** Replication Service (FRS). This service wraps around the Token Manager and intercepts all calls which either obtain, renew, or release tokens. The FRS keeps a record of tokens granted to each file within a replicated file system and determines which **FUSION** File Servers a client should use based on the activity within the **FUSION** Replicated File System (FRFS).

In addition to the FRS, extensions will be added to the local volume registry within the Protocol Exporter, the VLDB, the Volume Server within the Cache Manager, and the VOS applications. These extensions identify a file system as **FUSION** Replicated and identify the type of replica the server is managing.

The Cache Manager will contain hooks which will manage switching servers for FRFS clients. The Glue Layer will have similar hooks for accessing the FRS and for switching from local access to remote by going to the Cache Manager when appropriate. Additional modifications to the Glue Layer will manage file versioning, critical to the propagation of files among the replicas.

4.2.3 FUSION Replication Functional Overview

Each **FUSION** Replicated File System (FRFS) is allowed to have an indefinite number

Figure 28. VLDB

of copies. Only one copy will be a read/write (RW) replica. All other replicas will be read-only (RO). All RO replicas are additionally classified as either Principal RO or Secondary RO replicas. See Section 4.4.4, "Scaling Replication". When it is created, the file system will be defined as either a fully or loosely replicated file system.

All calls to modify a file stored on a FRFS will be forwarded to the RW replica. Calls which access a file will be forwarded to a server selected based on the type of replication, full or loose. In full replication, the call will go to a RO server unless the file is currently being modified or is out of date. If the file is being modified the call will be forwarded to the RW replica. If the file is out of date, the call will be forwarded to a replica which has an up-to-date version. This allows remote access to emulate single site full UNIX semantics. In loose replication, all access calls will be handled by the local server if one exists or by any other RO server regardless of current modifications being made. Local access will only be redirected if the RW replica has informed the RO replicas that a new version of the file is available for propagation. At that time the access will be redirected to an up-to-date server until the local copy can propagate the latest version.

When a file is modified at the RW node, each replica will asynchronously propagate the file to its storage node. Once the latest version of the file has propagated to a RO storage node, file access requests will once again use the node for file service.

In order to eliminate the limitations of having a single node serve as a reference point for all replicas, **FUSION** provides a hierarchical registration system for its RO replicas. This allows RPC's to be handled in parallel with one small group of nodes being responsible for informing a subset of replicas for each instruction sent from the RW replica. In addition to allowing RPCs to run in parallel, the hierarchical configuration avoids the bottleneck of having a single node for propagation.

In order to manage replication information and replica versioning, two files will be hidden within the root of the **FUSION** replicated file system: .frfsinfo, which contains file system information and is the same file propagated to all replicas and .replinfo, which contains replica specific information and does not propagate but, instead, is different on each replica. These files cannot be modified except by the **FUSION** replication services. By having a secret user listed in the ACL list for the file and having the **FUSION** replication services change to this secret user, these files cannot be modified even by the super user.

4.2.4 Document Organization Overview

This document is organized into seven sections describing the functional sections of the **FUSION** Replication design followed by sections on subroutine functionality, pseudo code, and data structure definitions.

The seven main functional components of the **FUSION** Replication design are:

Figure 29. Replication and the DCE File System

1. **FUSION** Replication Service
2. File Access (includes cache manager and glue layer enhancements)
3. File Propagation
4. File System Registration (include VLDB and CMS)
5. Network Instability Management
6. User Programs
7. Packaging and Installation

The **FUSION** Replication Service manages redirecting applications to various servers for file service and propagating the files modified on the RW replica. It interfaces with the DCE cache manager and glue layer which are servicing the application's[4] file access. This is covered in sections 4.4 and 4.5.

In order to create a **FUSION** Replicated File System, the replicas must be registered with either the VLDB or the CMS. This is covered in section 4.7. Because the network may have nodes leaving and joining, the **FUSION** Replication Service must be able to coordinate services among those replicas rejoining a network. The manner in which **FUSION** Replication manages network instability is discussed in section 4.6. User programs are also provided to assist the system administrator in managing **FUSION** Replicated File Systems. These are described in section 4.9. The section on Packaging and Installation includes dependencies on DCE installation and dependencies on other parts of the **FUSION** product. These dependencies are discussed in in setion 4.10.

Each section within this document discusses the operation of **FUSION** Replication with regard to fully replicated file systems. Loose replication will be discussed at the end of each section if its behavior is different than that described for full.

4. For purposes of this document the term "application" is being used to indicate a process, either user or system originated, which is accessing or modifying a file.

4.3 FUSION REPLICATION SERVICE

In DCE, each file system has a token manager which coordinates access to files within the file system. In a **FUSION** Replicated File System (FRFS), each replica has a **FUSION** Replication Service (FRS) which wraps around the DCE token manager at the node which stores the replica. The FRS determines which file system replica to access on a file by file basis. Once the replica is selected, access will be gained using the DCE cache manager or glue layer.

The FRSs for RW and RO replicas function slightly differently. The FRS which manages the RW replica (RW FRS) is responsible for keeping status on all modifications to the file system and notifying the FRSs for the RO replicas (RO FRSs). The RO FRSs keep information sent from the RW FRS and refer clients to the appropriate server based on the information they have received.

4.3.1 Exception Tokens

The FRS keeps track of which files are being modified with the use of exception tokens. The tokens used in DCE are open_exclusive, open_shared, open_read, open_write, lock_read, lock_write, status_read, status_write, data_read, and data_write. The FRS classifies the following tokens as "exception tokens" since they are incompatible with other tokens in their class (classes being open, lock, status, and data): open_exclusive, open_shared, lock_write, status_write, and data_write. All other tokens are compatible with one another.

When an application wishes to access a file with a call requiring an exception token, the call is referred to the RW replica. If the RW replica grants the token then the file id is placed in the exception token table. Once a file has an exception token listed in the exception table, all accesses to that file will be referred to the RW replica.

4.3.1.1 Granting Exception Tokens

The RW FRS keeps a table of all exception tokens granted for each file. Only the RW FRS can grant an exception token and it is responsible for updating all exception tokens as necessary.

FUSION replication services use timestamps to coordinate activities among the servers. Each time a RW FRS sends an RPC regarding and exception token or file propagation, the RPC will contain a timestamp.[5] The timestamps are always generated at the RW node and represents the time at which the RPC was generated. These timestamps are recorded in the exception token table and propagation queues and are used for tracking propagation activities.

---

5. The use of timestamps makes the assumption that time will not go backward at the RW replica's node.

When a request for an exception token is received, the RW FRS checks its exception table for the file and determines whether or not this exception token has already been granted. If it has, the timestamp for the token is updated. If another exception has been granted for this file, then the new token is added to the token list in the exception table and the timestamp is updated. In both these cases, the token request call is then passed on to the DCE token manager.

If, however, the file has had no exception tokens granted the RW FRS must contact all RO nodes to revoke all outstanding tokens on this file. If the RO replica is successful in revoking all its tokens, it will notify the RW replica, add the file and token to its exception table and refer all subsequent calls to the RW replica.

The RW replica may only grant the exception token if all RO nodes have successfully revoked their tokens. In the event that a RO node is unable to to revoke its tokens, the call fails and the RW notifies all nodes to relinquish the exception token.

Whenever the RW replica is unable to contact a node, it places that node in a list of nodes which it knows are potentially out of sync. If that node later contacts the RW replica, it will be told to reconcile itself before continuing. The RW replica will then remove this node from its list of out-of-sync replicas.

At the RO FRS, the exception table contains the file and the timestamp for the last exception token granted. Each time the RW grants a new exception token, it sends this information to all RO nodes. If the file exists in the RO's exception table then the timestamp for that token is updated. Otherwise, the file is added with the timestamp of this call.

4.3.1.2 Relinquishing Exception Tokens

In the RW node's exception table is a list of all exception tokens granted for a file. This list contains the file id, the token, and the timestamp for which the token is granted. Byte range tokens are considered as whole file tokens for the purpose of replication. A background activity, frfs_CheckEx() will run periodically at the RW node to see if a token has timed out. This process will check the elapsed time based on the timestamp and, if this time is greater than the timeout listed in the token id, it will call the DCE Token Manager with tkm_LookupToken() to see if the token is still valid.

Tokens are removed from the RW's exception token table whenever the token is given back to the token manager, either by timing out or by a call to tkm_ReturnToken(). When this occurs, the RW replica informs the RO replicas telling them to relinquish this token. If modifications were made to the file, the relinquish token call will also indicate that the file should be propagated.

4.3.2 Propagation Table

FUSION replication uses a form of versioning known as the commit count. The commit count is an increasing version number and represents the state of the file after the last write. The current design uses the value of ctime for its commit count. This

assumes time on the RW replica's storage node does not go backward. In order to guarantee incremental versions with updates more frequent than one per second, the current design plans on borrowing time values from the future.

When a file is given a new commit count (see Section 4.5.1.1 Status Writes) because of file modifications, the RW replica will issue a propagation instruction. The RO replica FRS contains a propagation queue which contains a list of files which are out-of-date and the commit count of the propagation instruction.

Upon receiving a propagation instruction the RO FRS will check its queue to determine if a previous instruction to propagate that file is waiting to be processed. If an entry for a previous instruction on the file does exist, the entry will be updated with the commit count for the most recent instruction. If no entry exists, one will be added for this file.

The propagation queue is accessed by the File Propagator which is responsible for actually propagating the file and notifying the FRS when the file has successfully propagated. Once the file has propagated, it may be removed from the exception table. Whether or not the file is removed from the exception table is determined by comparing the timestamp of the propagation queue entry with the timestamp of the exception table. If the timestamp in the propagation queue is greater then no exception tokens have been granted on the file since the propagation instruction was sent so the file is now up-do-date and the token is removed from the exception table.

4.3.3 Multiple Replicas on a Single Node

4.3.3.1 One node servicing replicas for several file systems

Only one FRS will run on any given node. If a node functions as the storage node for replicas for several different replicated file systems, the FRS will manage a separate exception table and prop queue for each file system for which it stores a replica.

4.3.3.2 One node servicing several replicas for the same file system

If a single node contains more than one replica of the same file system, the FRS will treat all copies of the same file system as a single entity unless one replica is the RW replica. The RW replica will always keep its own set of data.

Access to a file from any of the replicas will be checked against the same exception table. For each entry in the propagation queue, the propagator will propagate the file to all copies before removing it from the propagation queue or the exception table.

The FRS will determine which physical file system it is accessing by distributing the access requests evenly among the replicas such that if the node stores two replicas, every other simultaneous access will use the same replica. If the node stores three replicas, every third simultaneous access will use the same replica.

4.4 File Access

There are two types of file access in DCE, remote and local. The DCE file system is designed to maximize ease of access by a remote client by caching the file at the client side. This allows the client processes to perform multiple operations on a remote file without having to talk to the remote node. There will be hooks in the cache manager to allow the FUSION Replication Service to switch servers without interrupting service to the client process.

Local access is achieved through the glue layer which provides an extended set of vnode operations for coordinating local file access with remote. This layer will contain hooks to allow the FRS to have the client move from local access to remote by switching from the glue layer to the cache manager.

Both the cache manager and the glue layer communicate with the protocol exporter. One of the functions of the protocol exporter is to manage the tokens for each client using the node. It does so by calling the token manager. The FUSION Replication Service wraps around the token manager and manages replica services for both the cache manager and the glue layer.

4.4.1 Local Access

If the application for a FRFS is running on a storage node for the FRFS, the application will access the local FRS. For full replication, the local copy will be used as long as there are no exception tokens for this file and as long as the local version is up-to-date. As with remote clients, the local client will be sent to the RW node for files which are either out of date or are currently being modified, i.e. an exception token exists.

In loose replication, the local copy will be used for all files which are not known to be out of date. Once the FRS has been told to propagate the file, the local application will be told to access that file remotely until the local copy is up-to-date.

4.4.2 Cache Manager

4.4.2.1 File System Registry

The cache manager communicates with the VLDB or CMS and uses the information stored there to select a FRS for the client. Once a server has been selected, hooks in the cache manager will enable it to access a server on a file by file, rather than file system by file system basis. The new server information will be stored in the private data area of the vnode. The cache manager will use this server for the file rather than using the server associated with the file system in the volume registry.[6]

---

6. Because of the unique nature of Episode to store more than one file system on a physical partition, Transarc refers to the physical partition as the file system and the various complete subtrees stored on the physical partition as volumes. However, since FRFSs are not limited to the Episode physical file system, this document will refer to all complete subtrees as file systems, whether they be UFS file systems or AFS volumes. However, references to those registry services provided by

Hooks into the VLDB and volume registry (struct cm_volume) will allow a new field for **FUSION** Replicated File Systems. This field will be used to determine if a file system is a FRFS and, if so, if it is fully or loosely replicated.

4.4.2.2 Communicating with the Token Manager

The cache manager uses the protocol exporter for all file system operations requiring access to the physical file system. All calls to the protocol exporter are set up through cm_Conn() which looks up the file system for a specific file and sets up the RPC connection to one of the servers listed in the volume structure. Hooks will be provided in cm_Conn() so that if the file system is of type FRFS, cm_Conn() will check for a server stored in the vnode. If one exists, then the RPC will be set up for that node. If not, cm_Conn() will call frfs_SelectServer() to select a server and place this server in the vnode.

After setting up the RPC the cache manager performs one of many DCE RPC calls to the protocol exporter at the storage node. Among the many tasks of the protocol exporter is that of ensuring the cache manager has the appropriate tokens for the call. Therefore, all RPC calls received at the protocol exporter side will ensure the appropriate tokens have been granted for that function. This involves a call to tkm_GetToken().[7]

The **FUSION** Replication Service intercepts all calls to tkm_GetToken() with a call to frfs_PickSS(). A hook will be placed in ObtainSet() to call frfs_PickSS() if the file system type is FRFS. frfs_PickSS() will either pass the call on to tkm_GetToken() or return one of four new return codes, collectively referred to as GOTO<SERVER>. The four codes are GOTORW, GOTOPRINCIPAL, GOTOANYSERVER, or GOTOSERVER. If the error code is GOTOSERVER, a subsequent call must be made to frfs_whichServer() to obtain which specific server should be used.

If either the call to cm_Conn() or the DCE call fail, the cache manager analyzes the failure with a subroutine called cm_Analyze() which determines the reason for the failure based on the errorCode set by cm_Conn() and the subsequent DCE call. Hooks will be added to cm_Analyze() to accept the return codes GOTO<SERVER>.

Two additional token manager calls will be intercepted by the FRS: tkm_ReturnToken() called by SAFS_ReleaseTokens() and tkm_RenewTokens() called by SAFS_RenewTokens(). Hooks will be provided to call frfs_FreeToken() and

---

the DCE file system will be referred to as volume registries for consistency with the current DCE code.

7. All calls call tkset_AddTokenSet() which in turn calls ObtainSet() which calls tkm_GetToken(). These must pass the new GOTO<SERVER> return codes back to the client.

frfs_RenewTokens(), respectively. These interceptions are merely for FRS bookkeeping purposes and will not affect the cache manager functionality.

Additionally, the cache manager subroutine cm_FidToServer() looks up the file system server for a given file. Hooks will be added to cm_FidToServer() so that if the file system is of type FRFS, cm_FidToServer() will return the server located in the vnode rather than one listed in the volume entry. If no server has yet been selected, cm_FidToServer() will call frfs_SelectServer().

4.4.2.3 Automatic Switching to the Read/Write Replica

The cache manager has its own set of vnode_ops. Each operation calls cm_GetTokens() to ensure that the client has the appropriate token for the required operation. Hooks will be added to cm_GetTokens() to ensure that if the token requested is an exception and the server listed in the vnode is not the RW server, the client will be switched to the new server. The same routines will be used as those called by cm_Analyze() when a GOTO<SERVER> return code is recieved.

4.4.2.4 Changing File Servers

When the cache manager receives the return code of GOTO<SERVER> it is necessary to ensure that whatever tickets and tokens have already been granted by the old server are granted by the new one. The authentication tickets are automatically handled and it appears that the current DCE implementation will suffice. However, a new call, frfs_ChangeTokens() will be added to the cache manager. frfs_ChangeTokens() checks to see what tokens, if any, have been granted. If tokens were granted from the old server, the new server is contacted and those tokens are requested. Once the tokens for the new server have been obtained, the old server's tokens will be returned to the token manager.[8]

4.4.3 Glue Layer

4.4.3.1 Volume Registry

The local volume registry registers all locally attached file systems. Hooks will be provided in the local volume structure similar to those in the cache manager file system structure. These hooks will provide for a new type of file system, the FRFS. Only locally attached file systems are registered with the volume registry. The local application only accesses the VLDB or CMS after it has been switched to the cache manager for remote access.

---

8. It may be possible to reduce RPC overhead by allowing the old tokens to timeout rather than returning them to the old server's token manager.

4.4.3.2 Communicating with the Token Manager

In order to coordinate local access with remote, the glue layer obtains the appropriate tokens by calling the token manager. Each glue layer operation calls ts_GetOneToken(). Hooks will be placed in the glue layer operations to accept the new return codes of GOTO<SERVER> after each call to ts_GetOneToken().[9] If ts_GetOneToken() returns GOTO<SERVER> the local application will jump into the cache manager using the appropriate server and operations will continue from there.[10]

In addition to ts_GetOneToken() hooks will be put in ts_PutOneToken() which calls tkm_ReleaseTokens() so that frfs_FreeToken() will be called for file systems of type FRFS. Currently ts_RenewToken() is a stub. However, it is reasonable to assume it will eventually call tkm_RenewTokens() in which case it will need a hook to call frfs_RenewTokens() for FRFS file systems.

4.4.3.3 Changing File Servers

Once the code for changing from the glue layer to the cache manager becomes available, it will be necessary to ensure that all tokens held locally are granted remotely. This will be done with frfs_ChangeTokens() just as in the cache manager.

4.4.4 Scaling Replication

The **FUSION** replication service can support an unlimited number of replicas the only limitations being based on network resources. However, there is a point of diminishing return when a single node is providing replication services for more replicas than it can easily manage, with some types of machines being capable of serving more replicas than others.

Two limitations are quickly met when too many replicas are using the same node for all communications: the load on the serving node becomes so high its performance is severely degraded and the number of network messages becomes so many that waiting for round trip messages impacts performance negatively.

FUSION provides a hierarchical registration system for its replicas. This system eliminates the limitations of having a single node serving as a reference point for all replicas. It allows for RPC's to be handled in parallel with one small group of nodes being responsible for informing a subset of the replicas for each instruction sent from

---

9. ts_GetOneToken() calls GetTokens() which calls tcache_GetElement() which calls ProcessNewEntry() which calls GetToken() which calls tkm_GetToken(). All of these will ensure that the new return code from frfs_PickSS() is returned in the call to ts_GetOneToken(). GetToken() will need the hook to call frfs_PickSS() if the file system is of type FRFS.
10. Current research has not revealed whether the code which allows a local applications to become a remote client of another server exists in the DCE Snap 2 source. This code must exist for local applications to traverse a mount point for a remote AFS file system. This will provide the entry point for switching from local to remote access.

the RW replica. In addition, a hierarchical configuration avoids the bottleneck of having a single node for file access or propagation.

The hierarchical configuration involves designating RO replicas as either principal or secondary replicas. The principal replicas are responsible for talking to the RW and disseminating information to and from the secondaries for which it is responsible. The principal is also responsible for propagating modified files from the RW replica and providing the propagation service to the secondaries which will propagate their files from a principal.

4.4.4.1 Registering Replicas

FUSION replicated file systems are created either by registering the file system with the VLDB using vos_create or by using frfs_mkfs or frfs_modfs which register the file systems with the CMS for mounting using /etc/mount. (See section 4.9.1, VOS Commands and section X.X in Cluster Mount Service).

When a FRFS is created it can be specified as either a principal or secondary replica. By default, all replicas will be defined as principal. This way if there are a small number of replicas the scaling hierarchy is not utilized. For a small number of replicas, this will be more efficient.

4.4.4.2 Secondary Registration With Principals

When a secondary replica joins a network (See section 4.6.2.2, Principal Initializing from a VALID RW Replica). it will obtain the list of principal replicas from either the VLDB or the CMS. The secondary FRS will rank the list of servers based on the load leveling algorithm described in section 4.4.5, Automatic Replica Selection. The first server in the ranked list will be its principal. The secondary server will contact this server and register with it. The secondary may select the RW replica as its principal. When a secondary registers with a principal the principal adds that node to the list of nodes for which it is responsible.

The secondary FRS is responsible for staying in contact with the principal. Should the secondary be unable to talk to its principal, it will try to register with another principal. If no other principals are available, it registers with the RW replica which will add the secondary to the list of nodes for which it is responsible which automatically includes all principals.

4.4.4.3 Loose Replication

Since loose replication uses whatever node is selected as a server regardless of the state of the file, the hierarchical configuration does not impact loose replication with regard to file access. However, propagation for loose replication will be faster with the hierarchical configuration, both because it allows parallel propagation using principals, and because the loose replica can use the principal for file access while propagating the file to its own node.

4.4.5 Automatic Replica Selection

The volume structure in the cache manager contains an array of servers for the replicated file systems. This array is currently able to store 16 servers for each file system. Though there may be more than 16 replicas for a given FRFS, only 16 need to be stored in the volume structure. FRFS provides a load leveling algorithm for creating a ranked list of servers. The ranked list will contain the RW replica, a principal replica and 14 other RO replicas as long as there are 14 additional replicas.

If during operation any one server becomes unavailable, one of the remaining 14 will be selected using the load leveling algorithm to select among the remaining servers in the file system structure. If, at any time, the principal server becomes unavailable, a new principal server will be selected in the same manner. If, at any time during operation, all 14 servers and the principal become unavailable, the list will be reranked to include 14 currently operating servers and an available principal.

4.4.5.1 Load Leveling

The load leveling algorithm is not completely designed at this point. It will provide a better than random choice when ordering the list of preferred servers.

The following will be considered when ordering the list of servers for load leveling:

- average IO response time
- average ping time
- average load on the node
- network load

Other conditions such as the number of clients the node is serving and the number of replicas on a node, while relevant, do not give decisive information since different nodes may have different capacities for serving clients and managing replicas. This information will not be used in determining the ranked list of servers. However, it is available to any node which may need to offload clients in order to keep its load reasonable.

4.4.5.1.1 IO Response

I am not at this time certain how we will track IO response.

4.4.5.1.2 Ping Time

Currently AFS provides a service which pings all nodes with which it is communicating. This service will be expanded to provide keep-alives for all nodes a **FUSION** node needs to remain in contact with. The average ping times for these nodes will be maintained and used as part of the load leveling algorithm. If nodes are being considered for possible replication servers and they are not currently being polled, these nodes will be added to the keep-alive function. The frequency of pinging will be long for those nodes which have consistent ping times and shorter for those nodes whose ping times are wildly divergent. Once the ordered list of server

has been selected, only those servers in the list will continue to remain in the keep-alive service, but with ping times set for infrequent pinging. (See section 4.6.4.1, Keep-Alives.)

4.4.5.1.3 Load Average

The Node Status Server contains load level averages for each **FUSION** node in the cell. This information can be obtained by a simple query to the NSS regarding the node in question.

4.4.5.1.4 Network load

I am also not certain how to best determine network load.

4.4.5.1.5 Weighing the various features when determining ranking

Rapid ping time and rapid IO response are most important for selecting a replication server. Load average and network load will have less influence in determining the rank.

4.4.5.2 Changing servers for load leveling

Each node storing a FRFS will register with the NSS a set of ideal conditions for servicing FRSs. These conditions include a maximum number of replicas to be serving, a maximum number of clients, a maximum load, etc. These will be listed as part of the node attributes list.

If a principal finds that it is exceeding its maximum values for ideal FRS performance, it will need to level its load. This is done by moving secondaries to another principal. When a principal server discovers that it needs to move some secondaries to another node, it will notify the least ideal secondaries (ranked in inverse order using the same load leveling algorithm if possible) that they need to move to another principal. This allows the FRS activities to remain as evenly distributed among available nodes as possible.

4.4.6 Using a List of Preferred Replicas

Only the cache manager needs to concern itself with the list of preferred replicas since the glue layer only uses the local server. Any time an application is referred to another server it enters the cache manager for access.

The cache manager first accesses a FRFS via a lookup. If the FRFS is accessed via AFS on-disk mounts, the cache manager will contact the VLDB for information regarding the file system. If, however, the FRFS is accessed via /etc/mount, the cache manager will not actually traverse the mount point. The mount point will be traversed at the logical file system level and the file system structure will be built by the CMS and put into the cache manager volume registry. (See section x.x in Cluster Mount Service).

The VLDB stores all replica information in a vldbentry structure. The vldbentry structure has an array of 16 replicas for a given file system. Currently the DCE documentation indicates that the VLDB will be modified to allow multiple vldbentry

structures for a single file system. **FUSION** will use this increased functionality to allow an undetermined number of replicas.

The cache manager uses the the call VL_GetEntryByName() to get the file system information from the VLDB. The following two subroutines are used to access the VLDB: cm_CheckVLDB() and cm_GetVolumeByName(). These subroutines contain the following code:

```
VL_GetEntryByName();            /* gets entry from VLDB */
cm_InstallVolumeEntry();        /* caches entry         */
```

The following hooks will be added after cm_InstallVolumeEntry():

```
/* modifications for FRFS */
if (voltype == FRFS)
{       frfs_storeVol();
        while (nextentry != NULL)
        {       nextentry = VL_GetNextServersByName ();
                frfs_storeVol();
        }
        frfs_setServers();
}
```

This reads all servers in the VLDB and selects MAXSERVERS from the list. These servers are stored in cm_volume.servers[]. They are stored as follows:

```
cm_volume.servers[0] = RW
cm_volume.servers[1] = principal  (RW if no principals are up)
cm_volume.servers[2-MAXSERVERS] = secondaries or principals
```

4.5 File Propagator

The file propagator is responsible for file synchronization among the replicas. It reads the propagation queue and propagates files from the appropriate replica.

4.5.1 Modifications to the File System

In order to manage propagation, a replacement set of vnode-ops will be added to write the commit count to the disk both before each write begins and when each write is complete.

The following vnode operations in the DCE Physical File System (Episode) modify the status of the file:

```
vnd_Create()
vnd_Delete()
vnd_Replace()
efs_rdwr()
efs_setattr()
efs_symlink()
efsx_makemountpoint()
```

Each of these vnode operations calls epif_Mark() which updates the ctime, among other status information. **FUSION** will provide replacement vnode_ops which will call frfs_CommitFile() to manage the status writes necessary for file versioning.

4.5.1.1 Status Writes

FUSION defines two types of status writes, the increment commit count write and the full status write. The increment commit count write is necessary before beginning modifications on a file so that if the write fails after partially modifying the file, the commit count accurately reflects a different version of the file. The full status write is necessary when a write is completed in order to write out the new length of the file, mod times, etc.

Commit count writes can be triggered not only by the beginning of a write, but also whenever a writer loses the modification tokens, when a request for a propagation token is made and the file is actively being written, or when the file is being written and a maximum time period has elapsed since the last incremental commit count write (been too long).[11] Status writes, however, are only written when the file modifications are complete. If the last write was a full status write, an incremental commit count

11. Certain files, like log files, never complete their updates. If **FUSION** replication did not interrupt such modifications, these files would never propagate to the RO replicas. This would mean that all access to these files would always end up at the RW replica thus causing a performance degradation for file access. This is a particularly serious degradation with regard to loose replication.

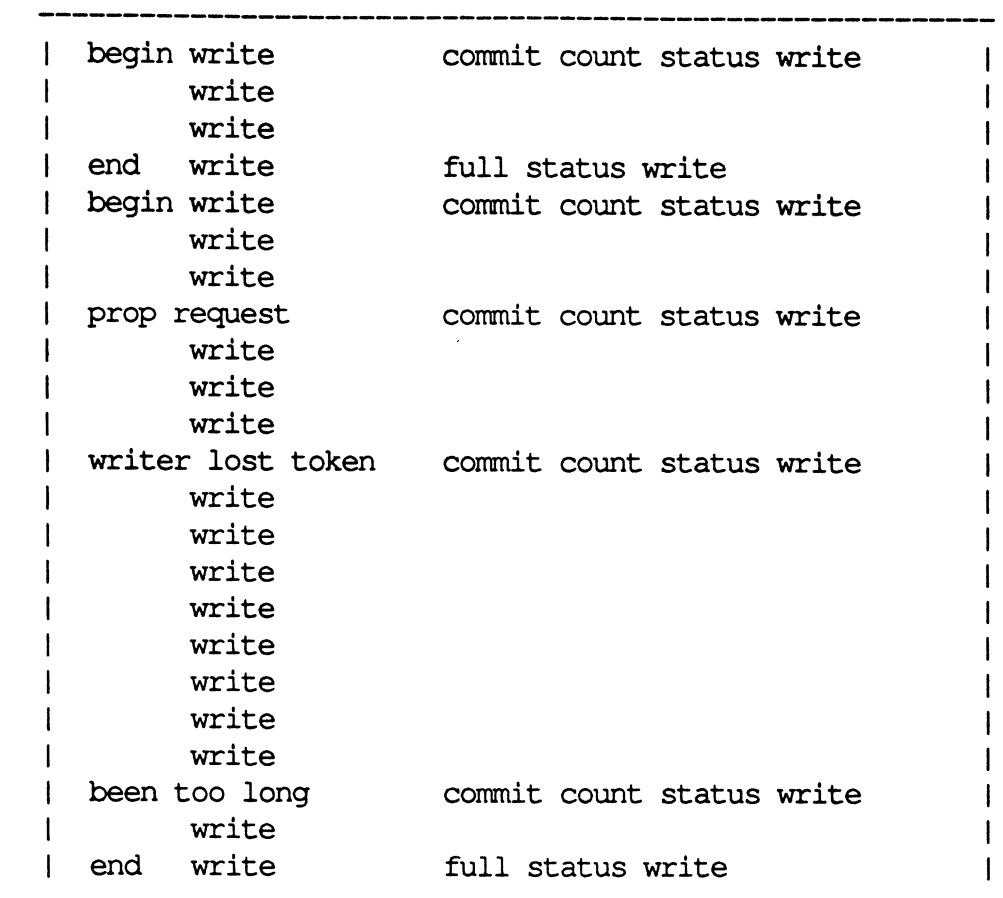


Figure 30. Flow of Status Writes

write will be necessary at the beginning of the next write Figure 30 shows the flow of status writes.

4.5.1.2 FRS Management of Status Writes

The above mentioned Episode vnode operations will be modified to call frfs_CommitFile() before actually performing the write. If frfs_CommitFile() returns TRUE, the vnode operation will call epif_Mark() to write the status. At the end of the operation, where the call is currently calling epif_Mark(), frfs_CommitFile() will be called as well. Similar functionality will be provided for UFS and JFS file systems as well.

frfs_CommitFile() will keep track of when an incremental commit count write needs to be made, when the writer loses the modification tokens, and when it has been too long since the last incremental commit count write. In addition, it will store the last commit count on a file and ensure that the commit counts are incremental. frfs_CommitFile() will convert the vnode used in the vnode operation into an afsFid using VOPX_AFSFID. The last type of status write will be stored according to the afsFid. If the last type of status write was a full status write and the request is for the

begin of a modification, then frfs_CommitFile() will change the last type of status write to incremental commit count and return TRUE. If the request is for the end of the operation, frfs_CommitFile() will change the last type of write for the file to full status write and return TRUE. Other information pertinent to propagation and time since last status write will be stored with the file handle as well.

The RW FRS will issue a propagation instruction whenever frfs_CommitFile() returns TRUE.

4.5.2 Byte-range Propagation

In addition to managing commit count status writes, frfs_CommitFile() will keep track of byte ranges modified. DFS uses 64K byte ranges when doing partial caching of files. These same ranges will be used for file propagation. If only part of a file is modified and need to be propagated, the propagation instruction will include the byte range to be propagated.

4.5.3 Propagation Instructions

When the FRS receives a propagation instruction from the RW replica, it places that file in the propagation queue. Each queue entry contains two commit counts, original commit count and current commit count. When the first instruction to propagate a file is queued, these two commit counts are equal. Subsequent instructions to propagate the same file will result in the the current commit count being updated to reflect the highest commit count for which a propagation instruction has been received. In addition, if the byte ranges for propagation has changed, this range will be expanded in the propagation queue entry.

The original commit count will be used for determining the replica's commit count. See section 4.5.8, Low Water Mark.

4.5.4 Propagation Queue Entry States

The propagation queue entries have the following states:

WAITING this is a new entry waiting to be serviced

HOLDING this entry failed to propagate and is holding to be retried

BEGUN the appropriate tokens have been granted to begin propagating

WRITING the file is now being written to the disk

ERROR an error occurred when writing to the disk and this file is potentially corrupt

These states are referenced when aborting a propagation, determining if a local copy is up-to-date, or being asked to relinquish propagation tokens.

4.5.5 Reading the Propagation Queue

When the **FUSION** Replication Service is initialized, a background activity will be created which waits for an entry to appear in the propagation queue. If an entry is

added to the queue this background activity will wakeup and start the file propagation. If the propagation is successful, the entry will be removed from the queue. If not, the status of the entry will be changed to HOLDING. Holding indicates that it has been tried and is "on hold" to try again at a later time. If an error occurred once data was written to the disk but before the propagation was completed, the queue entry's status is set to ERROR indicating the file is corrupt and should not be used.

4.5.6 Propagation Tokens

Because propagation affects file access differently than other operations, **FUSION** will define two new token types, FRS_READPROP and FRS_WRITEPROP.

DCE provides a call tkm_AddCompatRelation() which allows the addition of new compatibility relationships to the token manager. This allows the addition of new tokens. However, the new token will have to be added to the structure tkm_TokenType_t in tkm_tokens.h in order to be recognized by tkm_AddCompatRelation().

A compatibility relationship is currently restricted to two states, conflict or no conflict. The structure containing the relationship contains the two token types and a value of 0 (no conflict) or 1 (conflict). Currently, the token relationships are kept in a token_conflict table. For two tokens, X and Y, token_conflict[X][Y] = token_conflict[Y][X] = relationship.value. The following conflicting relationships will be established for FRS_READPROP and FRS_WRITEPROP using tkm_AddCompatRelation():

```
FRS_READPROP,        TKM_DATA_WRITE
FRS_READPROP,        TKM_STATUS_WRITE
FRS_READPROP,        TKM_NUKE
FRS_READPROP,        FRS_WRITEPROP
FRS_WRITEPROP,       TKM_LOCK_READ
FRS_WRITEPROP,       TKM_LOCK_WRITE
FRS_WRITEPROP,       TKM_DATA_READ
FRS_WRITEPROP,       TKM_DATA_WRITE
FRS_WRITEPROP,       TKM_OPEN_READ
FRS_WRITEPROP,       TKM_OPEN_WRITE
FRS_WRITEPROP,       TKM_OPEN_SHARED
FRS_WRITEPROP,       TKM_OPEN_EXCLUSIVE
FRS_WRITEPROP,       TKM_OPEN_DELETE
FRS_WRITEPROP,       TKM_OPEN_PRESERVE
FRS_WRITEPROP,       TKM_STATUS_READ
FRS_WRITEPROP,       TKM_STATUS_WRITE
FRS_WRITEPROP,       TKM_NUKE
```

Whenever a token conflict is present and an application attempts to gain one of the conflicting tokens, the token manager tries to revoke the existing token. The subroutine hs_RevokeToken() has a stub for future use. Currently it always agrees to

revoke all tokens without actually doing so.

Hooks will be added to hs_RevokeToken() to call the FRS to see if the propagation token should take precedence over the current token. In general, the propagation token will cause a modification token to be revoked. However, since the revocation of a modification token causes a status write to disk and since the status write generates a new propagation instruction, file writes could end up being continuously interrupted to propagate a limited number of changes. But if the propagation token did not generally revoke the modification token, it would be possible for a node to never find a window from which to propagate a file which is frequently modified. If such a window were never found, certain files would never propagate and the replica's low water mark would be allowed to be incremented. Since this would cause a severe performance degradation, it is necessary to allow propagation interrupts while limiting their frequency.

To limit the number of interrupts a node can cause on a file in order to propagate a newer version, the FRS may refuse to revoke the token. The RW FRS keeps a list of every node for which it has revoked a modification token since the last full status write. This information is stored with the frfs_CommitFile() data within the FRS. If the request to revoke the token for propagation comes from a node which has already interrupted this write, the FRS refuses to revoke the token. The node already has a version of the file which caused the initial propagation instruction. The propagation entry will be placed on HOLD and wait for a new propagation instruction. A new propagation instruction will cause the file to go from HOLDING to WAITING.

Whenever a full status write or been_to_long commit count status write occurs, the list of nodes which have propagated from this version is cleared.

4.5.7 Propagating the File

The subroutine frfs_PropIn() is the heart of the propagation mechanism. It sets up the RPC connections to the server, gets the appropriate tokens, fetches the file from the server and writes the file to disk.

4.5.7.1 Selecting the Server

In order to propagate a file, the secondary storage node functions as a cache manager for retrieving the data and as the protocol exporter for storing the data on disk. The node from which to obtain the file for propagation is selected based upon the storage type of the propagating node. If the node is a principal replica, the RW will be the storage node from which to propagate. If the node is a secondary replica, the principal with which the secondary is registered will serve as the propagation node.

Before propagation begins, the FRS will obtain the FRS_WRITEPROP token from its local token manager. Then, after selecting a server, the RPC will be set up with cm_Conn() and the FRS_READPROP token requested from the serving node's token manager. If the token cannot be gotten, the connection is released and a new server, if one is available, will be tried.

If either the FRS_WRITEPROP token cannot be obtained or no server can be found which will grant the FRS_READPROP token the queue entry is put on HOLDING to try again later.[12] Once both propagation tokens have been received, the status of the queue entry is changed to BEGUN. If the status equal BEGUN when the token is revoked, the FRS_READPROP and FRS_WRITEPROP tokens will be returned. Until the propagation tokens are relinquished frfs_KeepCurrent() will keep them current.

4.5.7.2 Fetching the File

The routine AFS_FetchStatus() is used to get the status of the file from which propagation is taking place and store it in the scache. The commit count of the file is checked against that in the prop queue and if the stored file is an older version than the current commit count, the propagation fails. The FRS will return the FRS_READPROP token to the server and go to another server, if any. If no other server is available, the FRS will release the FRS_WRITEPROP token, set the status to HOLDING and continue with the next file in the queue.

Once it has been determined that the file is to be propagated to this node, the FRS calls AFS_FetchData() which gets the data from the propagation node's protocol exporter. The routine SAFS_PutData() is used to put data onto the local disk. The data from AFS_FetchData() will be piped into SAFS_PutData(). Once the data has begun to be written, the status of the entry is changed to WRITING. Any request to revoke either propagation token will fail until the propagation is complete. If an error should occur during writing, the prop queue entry will have the status changed to WAITING and the propagator will exit. This will cause the failed entry to be first in the queue and be tried again immediately. If the propagation fails three times the entry is set to ERROR and retried periodically. No file access will be granted to a file with a propagation queue entry whose status is ERROR.

Once the data has successfully been written to disk, the cached status will be written, via SAFS_PutStatus() to the disk. The status is written last to ensure that the commit count reflects the true version of the file in case of node failure during propagation.

If a failure occurs during propagation or if the propagation tokens are revoked, the call frfs_Cleanprop() will manage all token releasing and cleanup based on the status of the propagation queue entry. Once the file is written to disk, frfs_CleanProp() will release the FRS_WRITEPROP and FRS_READPROP tokens, remove the entry from the propagation queue and remove the file from the exception token table if no more

---

12. There will be two types of HOLDING states. HOLDING1 indicates a problem accessing the file and the propagation queue will retry whenever there are no other entries marked WAITING. HOLDING2 indicates that the server is not yet ready to process this entry. A HOLDING2 entry will not be retried by the propagator since its status will be changed by the server for which it is holding.

recent exception tokens have been granted.

4.5.8 Low Water Mark

Each file system has a commit count referred to as the Low Water Mark (LWM). This value is stored in a non-propagating file called .replinfo. The LWM represents a guarantee that all files with a lower commit count than the LWM have successfully propagated to this replica.

Since not all writes to the RW server's disk will result in the propagation of a file and because propagation can take place out of order, the commit count of the RO storage node's file system will need to be updated after each propagation. The routine frfs_UpdateLWM() will check the propagation queue. If there are no entries for which the original commit count is lower than the commit count of the propagated file, the low water mark will be set to the commit count of the recently propagated file. The replica's LWM is used by the recovery program when initializing a replica which has been off the net for some time.

4.6 Network Instability Management

Because a distributed computing network is a dynamic rather than static environment, **FUSION** replication must contend with nodes leaving and joining the network throughout its normal activity. The goal of **FUSION** replication is for the application to see no difference in file access should a node leave the network.

In the current design, this is true for RO nodes. However, since there is only one RW replica and because modifications can only take place on the RW replica, if the RW replica becomes unavailable modification to the file system will fail. Read-only file access, though, will still be available through the RO nodes. A slight delay in access may occur while the principal replicas find the latest copies of all files within the file system. Only versions of files which have been propagated to at least one principal will be available so without the RW the application is not guaranteed an up-to-date version. With the scaling of propagation to relieve the load on the RW replica, it is hoped that in most cases at least one replica will have the latest version.

4.6.1 FUSION Replication Server States

To manage conditions of network instability the **FUSION** replication server has four possible states. So far this document has only discussed FRSs whose state is VALID. However, the following additional FRS states are also possible:

INVALID — this server has not begun initialization; use another server if possible.

RECONCILING — this server is currently reconciling its tables with another replica; please wait.

NORWREPLICA — the network is running without the RW replica; RO replicas agree on the current state of the file system; access is guaranteed to the most recent version able to propagate to any node within the current partition.

NOMEMORY — the kernel is out of memory and the RO FRS cannot keep all bookkeeping data; access is same as for VALID or NORWREPLICA, but full reinitialization is required as soon as memory is available.

4.6.2 Replication Server Initialization

The replication service will be initialized any time a FRFS replica becomes available to clients. There are two ways a replica becomes available: by starting the AFS bosserver or by having /etc/mount mount the replica on the current node.

In order for the bosserver to initialize and keep the FRS running, the **FUSION** Replication Service must be listed in /usr/afs/local/BosConfig for the node on which the service is to be maintained. This entry will be created whenever a FRFS is created using vos_create or when a replicated file is created using frfs_mkfs or frfs_fsmod. If the file system is created for mounting via /etc/mount, the initialization

code will only run if the replica is mounted. If the replica is not mounted until after Bosconfig is run, /etc/mount will force the initialization code to be run.

If the bosserver is not currently running when the /etc/mount of a replicated file system begins, the FRS initialization code will be called. A subsequent call to bosserver will not cause the FRS to reinitialize as the initialization code will prevent restarts of the service. The entry in the BosConfig file should indicate as well that no restarts should take place (most services restart every Sunday morning at 4:00 AM).[13]

4.6.2.1 RW Initialization

When the RW first comes up, it simply zeroes its exception token table and sets its state to VALID.

The RW has no way of knowing that it is entering a currently running NORWREPLICA system, as opposed to being the first node up, since it does not take responsibility for knowing which nodes it is talking to (it assumes all nodes are up). It is up to the various running nodes to reconcile themselves with the RW replica and detect any conflicts.

All nodes running in NORWREPLICA state will be polling the RW replica to see if it is back. If a principal FRS finds the RW, it sets its state to RECONCILING, notifies its secondaries, and reconciles itself with the RW replica. If the node is a secondary, it informs its principal and waits to get the new information. If the principal notifies the secondary that it cannot talk to the RW, the secondary reconciles from the RW and registers with it for service.

4.6.2.2 Principal Initializing From a VALID RW Replica

When a principal FRS initializes from a VALID RW replica, it sets its state from INVALID (the state the server begins in) to RECONCILING. It contacts the RW informing it of its LWM. The RW replica returns to the RO FRS the current list of exception tokens. The RW checks for all files it stores and passes a list of files and their commit counts for all those files whose commit counts are higher than the replica's low water mark. For each file in the list, if the commit count on the RW replica is greater than that currently stored on the replica, the file is placed in the propagation queue and added to the exception token table.

Once the FRS knows all files for which it is not up-to-date and those which are currently being modified, it sets its state to VALID and begins servicing clients of this replica.

---

13. Is there a reason we would ever want to restart the FRS once it is running?

4.6.2.3 Principal initializing from NORWREPLICA Principal

If the principal FRS entering the network is unable to contact the RW replica, it looks for a principal in the NORWREPLICA state. If the principal it tries to contact is RECONCILING, it waits for the state to change. If no principals are available, it looks for a secondary. If no secondaries exist it sets its state to NORWREPLICA.

The NORWREPLICA principal's exception table only has those files the principal can propagate from within the current partition. Since these are different for each node, the exception table is not obtained from the NORWREPLICA principal as it would be from the RW, but built from the list of files which need to be propagated to this node.

During the NORWREPLICA state, the FRS maintains a reconciliation propagation queue which remains separate from the propagation queue used in VALID states. This allows the node's propagation queue to remain intact while out of touch with the RW replica.

Because the NORWREPLICA principal may not have the latest copy of all files in addition to returning all files it stores whose commit counts are higher than the LWM, the NORWREPLICA also checks all files in the reconciliation propagation queue. If there is a file in the propagation queue whose commit count is higher than the low water mark, that file will be added to the list of files which the initializing replica will need to add to its reconciliation propagation queue. The propagation instruction will include the node from which to propagate the file.

4.6.2.4 Principal waiting to initialize

The principal waits to initialize from the RW replica if the RW's state is INVALID or RECONCILING. It waits to initialize from a NORWREPLICA principal whose state is RECONCILING. However if the NORWREPLICA principal's state is INVALID, the initializing principal will look for another principal. If no other principals or secondaries can be found, the initializing principal will set its state to RECONCILING and contact the principals whose state was INVALID so they can reconcile themselves into a functioning partition.

When a FRS contacts another FRS either the request can be granted, because the state is VALID or NORWREPLICA, or the calling server will be told to wait if the servers state is RECONCILING or this server is the RW FRS and its state is INVALID. The calling server will be told to try another server if this is not the RW FRS and its state is INVALID. Whenever a calling FRS is told to wait, the FRS telling the server to wait will record the node in a list of nodes to notify when the state of the server changes.

4.6.2.5 Secondary initializing from VALID Principal

A secondary initializes from a principal exactly as the principal initializes from the RW. In addition, all files in the principal's propagation queue will be sent as propagation instructions.

4.6.2.6 Secondary initializing from NORWREPLICA Principal

A secondary initializes from the NORWREPLICA Principal just as a principal initializes from a NORWREPLICA Principal.

4.6.2.7 Secondary initializing from NORWREPLICA Secondary

If the secondary cannot find a principal, it will look for a NORWREPLICA secondary from which to initialize. If no other secondaries are available it sets itself to NORWREPLICA. As well as looking for the RW replica, the secondary will look for a principal to come up and reconcile with the NORWREPLICA principal as soon as possible.

4.6.2.8 Secondary waiting to initialize

If a secondary finds a principal or other secondary in the state of RECONCILING it will wait and initialize from that node. If the secondary finds only other secondaries whose states are also INVALID, it sets its state to RECONCILING and contacts the other INVALID nodes so they can reconcile among themselves and form a functioning partition.

4.6.3 Replication Server Reconciliation

As well as initialization triggering a reconciliation of servers, simple leaving and rejoining an active network may trigger reconciliation. For example, if the network is partitioned and comes together, some nodes may be running as NORWREPLICA while others are running as VALID. In this case the nodes running in NORWREPLICA state will need to reconcile themselves with the RW replica.

4.6.3.1 Running in NORWREPLICA state

When the RW replica is unavailable, the RO replicas will service the file system with the most recent files available. This is referred to as running a NORWREPLICA partition.

4.6.3.1.1 Principal Reconciliation

When the RW becomes inaccessible to a principal, it sets its state to RECONCILING and sets a special bit to indicate that it is reconciling with a principal rather than the RW. After notifying its secondaries, it then contacts all principals telling them the RW is down and including this nodes's lwm. Those principals which can communicate form the partition. The node with the highest low water mark is elected "reconciler". If more than one node has the same lwm the first in the file system list stored in .frfsinfo will be the reconciler. The principals in the partition keep track of who their reconciler was at the time of reconciliation. This will be used to simplify reconciliation among two or more NORWREPLICA partitions.

The reconciler goes through its exception table and sends a list of all files in its table and the commit count of the file on the reconciler's node to all reconciling principals. If any node has a more recent version than the reconciler, it will notify the reconciler who will in turn notify the other principals. Once the reconciler has a list of all files it needs to propagate it will add those files and the node from which to propagate to

the reconciliation propagation queue and to its exception table. It will then set its state to NORWREPLICA and notify its secondaries that they can now reconcile with it.

The principals receive a list of files from the reconciler. This list contains the commit count of the version stored on the reconciler's node. For each file in the list, the FRS checks the commit count on its replica and, if the commit count stored locally is less than that of the reconciler it adds the file to its reconciliation propagation queue. If the file is not in the exception table, it is added. If the commit counts are equal, it removes the file from its exception table. If it has a more recent commit count, it removes the file from its exception table and notifies the reconciler that it has a more recent version.

Once the reconciler has received notifications from all the principals for the files in its lits, it sends an end-of-reconciliation message to the principals. When a principal receives the end-of-reconciliation message, it creates an entry for all files left in its exception table to its reconciliation propagation queue and sets its state to NORWREPLICA. The principal then notifies all secondaries for which it is responsible that they can now reconcile with it.

4.6.3.1.2 File Access in a NORWREPLICA network

When running in the NORWREPLICA state the exception tables at all nodes contain those files which need to be propagated to that replica. If a request comes in to any FRS, the FRS checks its exception token table. If it does not have the most recent version, it looks in its reconciliation propagation queue for the node which has it and refers the client to that replica.

4.6.3.2 RW Replica reentering the network

When the RW reenters the NORWREPLICA network it can either be coming from a partition in which the RW was continuing to function and for which modifications may have been made or it can have actually been down, in which case it is initializing on reentering the network.

4.6.3.2.1 Low Water Mark

When a server goes to reconcile with a returning RW replica, it checks the last low water mark it knows of (the most recent commit count in the prop queue) against that of the RW. If the low water marks are the same, then the RO server rebuilds its exception token table from those files waiting to propagate from the RW replica and the exception table sent from the RW FRS. The FRS then sets its state to VALID.

However, if the RW has a more recent LWM, the RO FRS will initialize from the RW using the RO replica's LWM. In either case, the RO replica will queue any incoming instructions for processing after reconciliation is complete.

4.6.3.2.2 Initialization number

Each time the RW FRS is initialized, it registers an atomically increasing number. Whenever a RO FRS reconciles with the RW it stores this initialization number. This

number represents the state of the RW replica the last time a node reconciled with it. Whenever the node communicates with the RW replica it checks the initializing number. So, if the RW went down and rebooted before the RO node knew it was gone, the RO FRS would know that its current tables are incorrect and will reconcile with the RW. The principal nodes pass this number on to their secondaries so that the secondary will be guaranteed to be reinitialized whether it detected the RW down or not. The RW replica's initialization number is stored in its .replinfo file along with the low water mark.

In addition, if the RW FRS does not receive a return message to one of its requests, it lists that node in an array of nodes it knows are out of sync. If the node contacts the RW again, the RW FRS tells the node to reconcile before it continues since this node missed at least one instruction.

4.6.3.3 Principal reentering the network

A partitioned principal may reenter the network at any time. It may reenter by having a poll to the RW replica respond that the RW replica is again available. When a principal rejoins a VALID network, it initializes itself from the RW replica and notifies all its secondaries that it is now VALID.

As well as being responsible for reconciling the principals, the reconciler also has the responsibility of polling those principals which are not currently in the partition. When the reconciler discovers it can talk to a new principal, it finds the reconciler for that partition. If the reconciler cannot talk to the new principal's reconciler, it reconciles with the new principal and the new principal contacts its own reconciler with any later versions of files which can be propagated. If the two reconcilers can communicate, they reconcile and notify the other principals in their partitions of any changes that are made to the reconciliation propagation queues. Once all principals have been notified, the first of the two reconcilers in the volume entry will become the new reconciler.

4.6.3.4 Secondary reentering network

When a secondary node is out of contact with the network, it not only polls the RW replica, but also the principals. In the meantime it is running as NORWREPLICA and servicing all files stored on the replica. When the secondary node contacts either the RW or a principal replica, it reconciles with that FRS.

4.6.4 Notice of node unavailability

The DCE File System has a service which runs from the client to all nodes with which it is communicating and pings them every 4 to 10 minutes to ensure that the file service node is still available. **FUSION** will also be providing a similar service to other parts of the product. The DCE service will need to be coordinated with the **FUSION** service to reduce network traffic.

The FRS will use these services by adding all replicas a FRS needs to keep track of to the service. Currently only the DCE client maintains a list of nodes to keep in contact

with. In addition to coordinating the client server changes will need to be made to the file server to access the polling service.

4.6.4.1 Keep-alives

The following assumptions are being made with regard to the polling service:

- A **FUSION** service can register a node with the polling service and the **FUSION** service will be notified whenever the node's state changes (either UP or DOWN).
- A **FUSION** service with a registered node will inform the polling service whenever that node was contacted so the polling service does not do unnecessary pings.
- Minimum and maximum ping times can be requested for the service based on the need for immediacy when the node's state actually changes. For example: a node may suspect that a routing problem is about to be resolved and want to ping a specific node more often. Or it may suspect that a node is gone for some significant period of time and want a long wait between pings. These times can be modified by the registering service during normal operations.

4.6.4.2 Receiving notice

When the RW node notices the demise of a node, the FRS stores that node in a list and if the node comes back, tells it that it is out of date. The RW does not notice nodes which come back unless they are in the list. For nodes in the list, the node is told to reconcile before continuing. The RW does not poll nodes that are down. It is each node's responsibility to keep in contact with the RW.

A RW receives notice that a node is up when that node's FRS registers with the RW and begins initialization. The RW FRS keeps registration information only for those secondaries which have come up and are unable to contact a principal. It keeps this registration in order to make sure that these secondaries receive all the information normally sent to principals. If the secondary discovers a principal has become available, it will register with the principal and remove itself from the list on the RW FRS.

4.6.4.2.1 Principal and RW

When a principal node notices the demise of contact with the RW it notifies all principals and all its secondaries. If the principal's state is VALID, it notifies all its secondaries and they set their states to NORWREPLICA referring all calls in their exception table to the principal until the principal has completed reconciling. Once the principal has reconciled itself, it tells all its secondaries who, in turn, reconcile with the principal.

If a principal is notified from another principal that the RW is down, the second principal will test to see if the RW is down to it. If not, it tells the notifying principal that it can talk to the RW and the notifying principal can increase its polling waiting for the routing problems to resolve themselves. While waiting, all modification requests will fail as though the network were running in a NORWREPLICA partition.

If neither principal can contact the RW then they begin reconciling themselves. Each sets its state to RECONCILING and indicates that it is reconciling from a principal.

If the principal's state is RECONCILING when it is notified that the RW has gone down, the principal checks to see if it is RECONCILING with the RW or from a principal. If reconciling from the RW FRS, it begins reconciling with the other principals in the partition. If reconciling from a principal, it simply adds that node's lwm to its list for selecting a reconciler. If a reconciler has already been selected then it ignores the information as the node will contact a NORWREPLICA principal soon.

When the RW comes up, if the principal node's state is RECONCILING and the principal was reconciling with another principal, it immediately stops its current reconciliation and contacts the RW FRS and begins reconciliation. If the principal was reconciling with the RW FRS and it has gone down during reconciliation, the principal contacts the other principals and they begin reconciliations. However, this principal will not be a first choice for the reconciler since its tables are not guaranteed to be accurate.

4.6.4.2.2 Secondary and RW

When a secondary node notices the demise of contact with the RW it notifies its principal. The principal tests to see if it can contact the RW. If so, it notifies the secondary which will set its state to RECONCILING and wait until the router problems resolve themselves. If the principal cannot contact the RW it behaves as described in 4.6.4.2.1.

When a secondary node notices the return of the RW replica, if its state is NORWREPLICA it notifies its principal. If the secondary was unable to contact a principal, it reconciles with the RW FRS and registers itself so that it will receive all communications which are normally sent to principal servers.

Otherwise, once the secondary notifies the principal it sets its state to RECONCILING and waits for further information from the principal.

4.6.5 Special cases

Because user level programs will be provided which allow a user to convert a RW replica to a RO replica, it is possible for strange cases of files system synchronization to occur. For example, a network partitions, the RW continues functioning and then crashes. We now have two partitions functioning without RW replicas, but one partition is more current than the other. If all the nodes from the more current partition are brought down, it would be possible to convert a RO node from the less current partition into a RW. When one of the nodes on the more recent partition then comes back, the two nodes may have conflicts. Determining which data to keep may not be possible without human interference.

At this time, the RW replica will be considered the accurate file system and replicas with conflicts will not be allowed to join. The .replinfo file which contains information about the replicated file system will contain a history of the past updates

made on the primary. Out of sync replicas will be discovered by comparing the RO's history with that of the RW. If a replica which had more recent contact with the old RW replica were to join a network with a converted RW, the file system histories will differ. If these differ there is a potential conflict between the two replicas and the new node will not be allowed to join.

Because special conditions make it possible for replicas to be out_of_sync with the rest of the partition, it is necessary to allow a special type of mount to allow maintenance of the out_of_sync replica. When a replica is mounted this way, its FRS state will be INVALID, but certain maintenance operations will be allowed.

4.6.6 Managing Out of Memory Situations

FUSION will manage its in-kernel tables by allocating what should be a sufficient amount of memory for the exception token table and propagation queues. These tables and queues will then be able to grow should the need arise. Even though many kernels, such as AIX 3.1 are able to grow, there is still the potential problem of a heavily modified file system running on a fully loaded machined causing all kernel memory to be utilized. **FUSION** Replication Servers manage out-of-memory differently, depending on the type of server.

4.6.6.1 Read Only FRS

RO **FUSION** Replication Servers keep a large amount of data which is used primarily for ease of maintenance and performance enhancements for network instability. If this data were not kept, the tables would be 1/3 smaller. However, the FRS would be less efficient and would require full initialization when reconciling with other nodes.

To manage such a condition, **FUSION** will define an additional state called NOMEMORY. When the FRS is set to this state it can still redirect users to the RW node if modifications are being made to a file and it can still propagate files to its replica. It does not know the commit count of the files it needs to propagate, cannot update its low water mark, and must be initialized as though rebooting if it is asked to reconcile with another node. An FRS which is running NOMEMORY cannot be selected as the reconciler unless all nodes are running NOMEMORY, in which case reconciliation will involve all other nodes initializing from the NOMEMORY reconciler.

When a RO FRS discovers it cannot expand its tables because the node has run out of memory, it sets its state to NOMEMORY and begins rewriting its tables. The FRS has a reserved structure which is the beginning of the NOMEMORY exception token table. The first three file IDs listed in the exception token table will be listed in this reserved structures. The data area containing these three entries is then freed to contain the next entries in the exception token table. This process will continue until all entries in the exception token table have been copied to a linked list which contains only the file ID's of the table.

Once the exception token table has been transferred, the remaining entries are freed and can be used to begin transferring the propagation queue entries into a similar linked list containing only the fileIDs of those files requiring propagation. Propagation entries which are HOLDING or ERROR will be placed in separate linked list containing only file IDs whose state should be set to the appropriate state.

Because the FRS is not doing the requisite record keeping to make its own decisions regarding file propagation and token releasing, propagation must change for a FRS in NOMEMORY state. Once the file has propagated, the RW FRS will be asked whether or not this file is still in the exception token table. If not, the RO FRS will remove the file. If so, the file will stay in the exception table. In addition, because there is no record of those commit counts waiting to be propagated, the RO FRS cannot update its low water mark.

While changing tables from VALID state to NOMEMORY state, all incoming RPC's will wait. This will not affect user access, since all user requests get the state of the FRS before continuing. However, secondaries trying to register or RW instructions will wait until the tables have been modified.

If, after the data has been stripped from the RO FRS it still runs out of memory, the RO FRS will have to resort to a memory swapping scheme such as that which will be used by the RW FRS. This is defined in the next section.

4.6.6.2 Read Write FRS

Unlike the RO FRSs, all data kept at the RW node is essential to the operation of the file system. Therefore, it will be unable to declare the RW FRS as running in a state of NOMEMORY. Instead, the memory can be expanded by utilizing some of the on-disk memory accessible to the cache manager. The RW FRS will reserve a data cache for its tables and swap them to and from the disk as necessary to preserve its data.

4.6.6.3 Reclaiming Memory

The RO FRS will know both the number of files in its exception token table and in its propagation queue when it ran out of memory. When the number of files in both these lists gets lower than the numbers when the node ran out of memory, enough memory to manage table of this size will be requested. If the memory is available, the RO FRS will reinitialize from the RW node and begin functioning as a VALID FRS once again.

4.7 VLDB

The DCE documentation promises two routines to manage the addition of more than one vldb entry per volume. Though the code does not currently exist in snap 2, it can be assumed that it will exist in a future release. We will use the routines VL_GetNextServersByID() and VL_GetNextServersByName() to get the additional server entries used by the cache manager. It is possible that Transarc may be expanding the cache manager's file system cache to accept more than 16 servers as well, though I have found no indication of this. Therefore, **FUSION** will continue to store an ordered list of servers in the 16 slots provided for the cache manager's file system cache.

4.8 File System Access for Non-FUSION Nodes

Because **FUSION** Replicated File Systems may coexist in cells with non-**FUSION** nodes, it is important that the file system be accessible to the non-**FUSION** client. Such clients will access the FRFS as though it were only one replica — the RW. The following code will need to be added cm_ConnByMHosts():

```
if ( volumetype == FRFS )
        cm_ConnByHost (vldbentry.server[0]);
```

4.9 User Level Commands

4.9.1 VOS Commands

The vos commands are used by DCE to create file systems in the VLDB and to modify their entries once they have been created. A new set of commands, vosx, will be used for creating and managing replicated volumes. Hooks will be added to the commands to tell the user to use the equivalent vosx command if the file system is of type FRFS. The following commands will be used to create and manage replicated volumes:

vosx_create — creates a **FUSION** Replicated File System and establishes the RW replica. Accepts a new argument, –type, which sets the frfsrepl field in the vldbentry to either FULL or LOOSE depending on the type or replication. In addition to creating the VLDB entry and creating the volume header, vosx_create will also make an entry into /usr/afs/local/BosConfig to initialize FRS for the RW replica. It will also create a file called .frfsinfo and place the vldbentry information in this file.

vosx_addsite — adds RO replicas to the specified FRFS. vosx_addsite accepts a new argument, –secondary, to determine the type of RO replica. If no flag is given, the replica is a principal replica. In addition to adding the new entry into the VLDB as in vos_addsite, vosx_addsite will create a volume header similar to that created with vos_create. Because DCE's lazy replication uses vos_release to release a new clone of the RW replica, vos_addsite does not create a volume header. **FUSION** Replication does not use the vos_release command to manage propagating to the new node, so the vosx_addsite command will need to create the volume header for the new node. vosx_addsite will also place an entry in /usr/afs/local/BosConfig to initialize the FRS for the RO replica.

vosx_move — moves the RW replica from one partition to another. When this takes place, the entry for BosConfig is deleted on the old node and added to the new node. The file .frfsinfo is updated to reflect the move.

vosx_rename — renames the FRFS. This will not only cause the vldb entry to be modified, but also all RO volume headers. In addition, the BosConfig entries will be modified to reflect the new name and the .frfsinfo file will be modified to reflect the change as well.

vosx_remove — removes the RW replica from the VLDB and removes the volume header as well. If no RO replicas exist, the entry will be removed from the VLDB. If a RO replica exists, the .frfsinfo file will be modified first to reflect the removal of the RW replica. The

command will not return until at least one node has propagated this file. Then the volume header will be removed. In addition, the FRFS initialization entry will be removed from the BosConfig file.

vosx_remsite — removes a RO replica from the VLDB. Unlike vos_remsite, vosx_remsite will remove the volume header as well. The .frfsinfo file will be modified (unless there is no RW replica), and the FRFS initialization entry will be removed from the BosConfig file.

vosx_zap — removes a volume without modifying the VLDB. This will also remove the BosConfig entry for initializing the FRS.

vosx_examine — will be modified to indicate fields for **FUSION** Replicated File Systems.

vosx_help — will be modified to include help for **FUSION** Replicated File Systems.

vosx_listpart — will be modified to include FRFS information.

vosx_listvldb — will be modified to include FRFS information.

vosx_listvol — will be modified to include FRS information.

vosx_syncserv — will be modified to synchronize **FUSION** Replicated File Systems including **FUSION** relevant data entries. It will update .frfsinfo files as applicable.

vosx_syncvldb — will be modified to synchronize **FUSION** Replicated File Systems including **FUSION** relevant data entries. It will update .frfsinfo files as applicable.

4.9.2 FRFS_mkfs

This application will make a **FUSION** Replicated File System. It accepts the same arguments as mkfs and FRFS_modfs and basically calls mkfs to make the file system and FRFS_modfs to convert it to a FRFS.

4.9.3 FRFS_modfs

This application will modify a file system created with mkfs or its equivalent into a **FUSION** Replicated File System. It notifies the CMS of the new replica and, in turn, updates the .frfsinfo files for the file system. This application can also be run to modify already existing replicas by changing principal replicas to secondaries or vice versa. No such equivalent will be able for the VOSX commands. To change from a secondary to principal the vosx_rmentry command will be used to remove the original entry and vosx_addentry will be used to add the new entry.

4.9.4 Recovery

The application level recovery program is available so that a system administrator can force a replica to reconcile with another replica. This program is run when a replica

appears to be seriously out of sync. The recovery program, frfs_fsrecover accepts as its arguments the node from which to recover, the node which is recovering, and a special flag to indicate that all files in the file system should be examined rather than only those above the low water mark.

When application level recovery begins, the FRS on the node which is recovering is set to INVALID, the node from which recovery is taking place is sent a low water mark and responds with a list of files which are more recent than the low water mark. This is the same way a node reconciles with a principal or the RW replica when it initializes. Once the recovery program is complete, the recovering node will set its state to RECONCILING and reconcile with the appropriate principal or RW replica.

4.9.5 Changing RW replicas

It is possible that it will be necessary to change the replica which is serving as the RW replica for a FRFS. Two things must happen for this to take place. Because a FRFS can only have one RW replica at a time, the current RW replica must first be converted to a RO replica, and the list in the VLDB and CMS file system entries for the FRFS must be modified.

4.9.5.1 Converting the RW replica to RO

A RW replica does not have to be running to be changed to a RO replica. If the replica is not running when the conversion takes place the FRS will discover it is no longer the RW when it tries to initialize itself. If this happens the FRS will change its internal flags and get the name of the node which is now the RW replica. The old RW replica will propagate the .frfsinfo file and then initialize itself as a RO replica.

If the replica was created using the vos command, the vldb entry for the file system will be modified so that the replica will be moved from the first entry in the volume structure, reserved for the RW, to a later entry in the same list. The server flags will be changed from RW to either PRINCIPAL or SECONDARY, depending on which flags were passed to the conversion program.

If the replica was created using /etc/mount, the replica must first be unmounted. It is also necessary that at least one replica for the FRFS must be available. The CMS is contacted and changes the status of the RW replica to PRINCIPAL or SECONDARY, depending on which flags were passed to the conversion program. This changed information is then written into all .frfsinfo files on all currently mounted replicas of the FRFS. The modified replica must be remounted via /etc/mount after the conversion program has completed.

4.9.5.2 Converting a RO replica to a RW replica

There can only be one RW replica per FRFS. If a RW replica already exists, the conversion will fail.

The RO replica does not have to be running to be changed to a RW replica. However, a lot of checking to ensure that the replica is as up-to-date as possible within its current partition will not run if the RO replica is not currently part of a

partition. If the node is not running, the replica will discover it is the RW when it begins its initialization.

If the replica was created using the vos commands, the entry for this FRFS will be modified so that the RO replica will have its server flags changed to RW and have its current entry in the list of servers deleted while a new entry in the list will be created at the head of the list, where the RW replica is listed. During this operation, if the replica is up, the FRS will change its state to INVALID. When the conversion is complete, the FRS will initialize itself as the RW replica.

If the replica was created using /etc/mount the file system does not have to be unmounted before being converted. In fact, it is best if it is mounted so that all the up-to-date checking can take place. Once the node has been checked to ensure it is up-to-date with regard to its current partition, the FRS changes its state to INVALID. The CMS will be contacted to change the frfs_repltype to RW. All currently mounted FRFS replicas will have their .frfsinfo files modified to indicate the change. Once the CMS has completed the changes, the FRS begins initializing itself. This enables the application to convert a RO replica to a RW replica without having to unmount and remount the replica.

4.10 Packaging and Installation

FUSION Replication can be packaged independently from much of the rest of the FUSION kernel extensions. However, it has several dependencies upon the rest of the FUSION product and replication performance will differ without all these components. In addition, FUSION replication depends on special DFS installation configurations.

4.10.1 FUSION Dependencies

4.10.1.1 Cluster Mount Server

Without the Cluster Mount Server, all FUSION Replicated File Systems must be registered with the VLDB and mounted using DCE on-disk mount junctions.

4.10.1.2 Node Status Server

FUSION uses the node status server to provide polling services for nodes which the FRSs need to keep in contact with. This particular service is the only service essential to FUSION replication.

In addition the NSS provides the FRS and client with information used to determine which servers are optimum file access and to help with load balancing. Without the NSS these features of FUSION replication will be hindered and access will be done by selecting servers randomly.

4.10.2 DFS Dependencies

There is no intrinsic requirement in DFS for a file server node to also provide a cache manager running on the node. However, FUSION replication requires that all nodes serving FRFS replicas have cache managers running on them.

4.11 Subroutines

The subroutines which make up the FUSION Replication Service can be divided into five categories:

1. Changes to Existing DCE Code
2. Exception Token Management
3. Propagation Management
4. Node Selection
5. Network Instability Management

4.11.1 Changes to Existing DCE Code

The following DCE modules need modifications:

cm_Analyze	used to analyze errors in either the connection made by cm_Conn() or by the DCE call itself. This routine will be modified to accept the new error codes for GOTO<SERVER>.
cm_Conn	used to make all remote connections within the cache manager. This subroutine will be modified to look in the vnode for the FRS server to use and call frfs_SelectServer() should no server be listed in the vnode.
cm_FidToServer	finds the server handling a particular file ID. It will be modified to look in the vnode for an FRS file and return that server or call frfs_SelectServers() if a server has not been selected for this file.
hs_RevokeToken	revokes outstanding tokens for the token manager. It will be modified to manage propagation tokens FRFS_READPROP and FRFS_WRITEPROP revoking tokens as appropriate.
cm_ServerDown	will contain a hook to call frfs_ServerDown if FRFS.

See also sections 4.4.2, 4.4.6, and 4.8 for additional DCE modules to be modified and those modifications required.

4.11.2 Exception Token Management

The following subroutines will manage the exception token tables:

frfs_EXaddfile	adds a file to the exception token table.
frfs_EXaddnode	adds a node to a file entry in the exception token table.
frfs_EXfindfile	finds a file entry in the exception token table.
frfs_EXfindnode	finds a node in a file entry in the exception token table.

frfs_EXrmfile	removes a file entry from the exception token table.
frfs_EXrmnode	removes a node from a file entry in the exception token table.
frfs_ExTblHash	manages the exception token hash table and returns pointers to to exception token structures as requested.
frfs_EXCheck	runs in background on the RW and periodically polls the token manager for all files in the exception token table to ensure the tokens have not timed out.
frfs_EXFree	frees a token from the exception token table.
frfs_EXupdate	updates a token in the exception token table
frfs_FreeToken	wraps around calls to tkm_FreeToken. Calls frfs_ReleaseToken() if this is the RW and the token is an exception token.
frfs_GetNode	gets the node from the protocol exporter. Called by frfs_FreeToken.
frfs_GrantEx	sent from RW to ROs which will put the token in their exception token tables and revoke tokens as necessary.
frfs_KeepCurrent	runs in background and keeps tokens from timing out while being manipulated by the FRS.
frfs_ReleaseToken	removes the token form the RW exception token table and notifies all principal nodes if this is the last entry in the table.
frfs_RenewTokens	wrap around for tkm_RenewTokens(). Updates the timestamp for this token if it is in the exception token table.
frfs_StripEx	strips exceptions tokens from those granted by RO node's token managers.

4.11.3 Propagation Management

The following subroutines will manage propagation:

frfs_CommitFile	will manage status writes and send propagation instructions when appropriate.
frfs_CFfindfile	looks up file in commit file list
frfs_CFfindnode	looks up node in commit file list file entry
frfs_CFaddfile	adds file to commit file list
frfs_CFaddnode	adds node to commit file list file entry
frfs_CFrmentry	removes entry from commit file list

frfs_CleanProp	cleans up propagation which has been aborted either because of an error or a revoked token.
frfs_PropFile	notifies replicas that this file, commit count #, needs to be propagated.
frfs_PropAnalyze	analyzes errors returned during propagation and either aborts the propagation or tries propagating from another node depending on the error.
frfs_ReadPropQ	runs in background waiting for an entry on the propagation queue and begins propagation as soon as an entry appears
frfs_PropIn	manages propagation of a file.
frfs_PQfindfile	find file in propagation queue.
frfs_PQaddentry	adds file to propagation queue.
frfs_PQrmentry	remove entry from propagation queue.
frfs_UpdateLWM	looks in the propagation queue for a file with a lower original commit count than that of the most recently propagated file and if none is found resets the low water mark to the commit count for the most recently propagated file.

4.11.4 Node Selection

The following subroutines are used to handle automatic node selection:

frfs_ChangeTokens	gets all outstanding tokens from the new server and frees them from the old server. Called whenever a client is changing servers on a file.
frfs_ConvertServer	converts the VLDB or CMS server list into a cm_server structure adding FRFS features where applicable.
frfs_GetIDServer	returns a pointer to an frfs_serverid structure given a host ID.
frfs_GetServer	returns a server of type from all available servers stored in the VLDB.
frfs_GetVolServer	returns a pointer to an frfs_server structure which contains the file system requested.
frfs_OrderServers	orders a list of server nodes based on load and client count information received from the NSS.
frfs_PickRepl	manages the selection of a replica when a single node stores more than one replica of a given file system.
frfs_PicksSS	wraps around all calls to the token manager. This routine selects which server to use based on the exception token table

and status of the FRS. It will either select itself as the file server or return the error code GOTO<SERVER> indicating which server should be used.

frfs_RORegister — registers a server with the RW as a principal server. If this is a secondary server, marks itself as temporarily functioning as a principal server.

frfs_RefreshServers — reoptimizes the list of servers to use when a number of servers have gone down.

frfs_SelectServers — selects a server of type from the optimized list and places that server in the vnode for a file.

frfs_RenewServers — renews list of available servers for selection.

frfs_SetServers — puts selected servers in cm_server.servers[MAXSERVERS].

frfs_StoreVol — stores replication servers retrieved from the VLDB or CMS.

frfs_AddVolServer — adds a volume server to the list of frfs_volume servers.

frfs_WhichServer — called after an error return of GOTOSERVER to specify which server the FRS intends the client to goto.

4.11.5 Network Instability Management

The following subroutines manage network instability:

frfs_BeginRecon — zeroes all current commit count values in the propagation queue, obtains a list of all principal nodes and notifies them that the RW is down, and chooses a reconciler based on their returned low water marks.

frfs_EndRecon — receives notice from the reconciler that reconciliation is complete. Adds any remaining files in its exception token table to the propagation queue and sets the FRS status to NORWREPLICA.

frfs_GetRecon — receives notice from the reconciler for a file and returns the commit count of the file as it is stored on this node. Clears the entry from the exception token table.

frfs_NoRW — changes the state of the server and begins reconciliation and notification as necessary.

frfs_Reconciler — walks the exception token table and notifies all other principals of any files contained within. Waits for responses and adds the file to the propagation queue if necessary. Notifies other principals of any files to be propagated. Also notifies other principals when reconciliation is complete and

	sets its status to NORWREPLICA.
frfs_RPQaddentry	adds an entry into the reconciliation propagation queue
frfs_RPQrmentry	removes an entry from the reconciliation propagation queue
frfs_ReplInit	initializes all tables and queues for the FRS when a new node joins the network or when the RW returns and the node must reconcile with it.
frfs_ReadInfo	reads information from .frfsinfo
frfs_VerifyInfo	verifies information from .frfsinfo with that in CMS or VLDB.
frfs_ROinit	initializes into a NORWREPLICA file system
frfs_PollNode	adds node to list to be polled
frfs_CompHist	compares file system history with node from which FRS is initializing.
frfs_EXtblinit	initializes exception token table.
frfs_PQinit	initializes propagation queue.
ROreconcile	reconciles read only node with FRS.
frfs_ServerDown	sets the server status as DOWN if the node has gone down according to the NSS.
frfs_FRSServerDown	sets the server status as DOWN if the node has gone down according to the NSS. Begins reconciliation if RW or reconciler down.

4.12 NIDL Specifications

The following RPC's will be defined using NIDL:

```
interface frfs {

typedef struct fs_entry {
        char    name[MAXNAMELEN];
        u_long  replType;
        u_long  nServers;
        struct  afsNetAddr  siteAddr[MAXNSERVERS];
        u_long  sitePartition[MAXNSERVERS];
        u_long  siteFlags[MAXNSERVERS];
        u_long  flags;
} fs_entry_t;

typedef struct extbl {
        struct  afsFid fid;
        u_long  timestamp;
} extbl_t;

typedef struct filelist {
        struct  afsFid fid;
        u_long  commitcount;
} filelist_t;

typedef struct hyper tkm_tokenID_t;

/* Initialization routines */
boolean
frfs_WriteInfo (
        [in]  handle_t   handle,
        [in]  fs_entry_t entry )

boolean
frfs_UpdateBosConfig (
        [in]  handle_t  handle,
        [in]  char      frs_entry[FRSENTRYSIZE] )

boolean
frfs_ROReconcile (
        [in]  handle_t   handle,
        [in]  long       lwm,
        [in]  long       repltype,
```

```
        [out] short      init_number,
        [out] char       history[HISTORYLEN],
        [out] extbl_t    extable,
        [out] filelist_t filelist[MAXFILES] )

/* Server selection routines */
boolean
frfs_GrantEx (
        [in]  handle_t   handle,
        [in]  afsFid_t   file,
        [in]  tkm_tokenID_t      extoken,
        [in]  u_long     timestamp )

boolean
frfs_WhichServer (
        [in]  handle_t  handle,
        [in]  afsFid_t  file,
        [out] server_t  server )

/* Propagation routines */
VOID
frfs_EXfree
        [in]  handle_t  handle,
        [in]  afsFid_t  file,
        [in]  tkm_tokenID_t      extoken,
        [in]  u_long    timestamp )

VOID
frfs_GetFile (
        [in]  handle_t  handle,
        [in]  afsFid_t  file,
        [in]  long      commmitcount,
        [in]  u_long    timestamp )

boolean
frfs_StartProp (
        [in]  handle_t  handle,
        [in]  afsFid_t  file,
        [out] token_t   extoken )


/* Network Instability Management */
boolean
```

```
frfs_NoRW (
        [in]  handle_t  handle,
        [in]  long      mylwm,
        [out] long      hislwm )

boolean
frfs_RORegister (
        [in]  handle_t   handle,
        [in]  char       history[HISTORYLEN],
        [out] short      errno )

VOID
frfs_GetRecon (
        [in]  handle_t   handle,
        [in]  filelist_t filelist[MAXFILES] )

VOID
frfs_EndRecon (
        [in]  handle_t   handle )
} /* end of NIDL interface */
```

4.13 Detailed Design

The following section contains detailed design specifications for **FUSION** Replication. Some sections contains pseudo code. However, the pseudo code has not been thoroughly checked for inconsistencies and is primarily a guide to the logic.

4.13.1 Modifications to DCE Code

```
------------------------  cm_Analyze()  ------------------------------------

/* This routine is used by DCE to analyze errors from either cm_Conn() or
 * the AFS_Call() sent to the storage node.  It is being modified to accept
 * new return codes GOTO<SERVER> for FRFS file systems.
 *
 */
int cm_Analyze(connp, errorCode, fidp, rreqp)
  register struct cm_conn *connp;
  long errorCode;
  register struct cm_rrequest *rreqp;
  struct afsFid *fidp;
{
        long *pp;
        register long i, code, returnCode;
        struct cm_server *serverp;
        register struct cm_volume *volp;


        if (errorCode == GOTO<SERVER>)
                frfs_Analyze();
}

------------------------  end cm_Analyze()  --------------------------------

------------------------  cm_Conn()  ---------------------------------------

/* The following subroutine is used by DCE to make all remote connections
 * within the cache manager.  It includes the modifications needed to support
 * FUSION Replicated File Systems.
 */
struct cm_conn *cm_Conn(fidp, service, rreqp)
  register struct afsFid *fidp;
  long service;
  register struct cm_rrequest *rreqp;
```

```
{
        register struct cm_volume *volp;
        register struct cm_conn *connp;

        if (!(volp = cm_GetVolume(fidp,rreqp)))
        {       if (rreqp)
                {       cm_FinalizeReq(rreqp);
                        rreqp->volumeError = 1;
                }
                return (struct cm_conn *)0;
        }


        if ( ISFRFS(volp) )
                connp = frfs_Conn(...);
        else
                connp = cm_ConnByMHosts(...);

        cm_PutVolume(volp);
        return connp;
}

------------------------  end cm_Conn()  -------------------------------------

------------------------  frfs_Conn() ----------------------------------------

/* frfs_Conn()
 *
 *      looks in private area of vnode and, if no server listed,
 *      sets it.
 */
frfs_Conn()
{
        struct cm_server *serverptr;

        if ( fidp->vnode->v_data )      /* See Note 1 */
                serverptr = fidp->vnode->v_data;
        else
        {       serverptr = frfs_SelectServer( fid, ANY );
                fidp->vnode->v_data = serverptr;
        }
        return (cm_ConnByHost (serverptr, ... ));
}
```

```
----------------------  frfs_Conn()  ------------------------------------

----------------------  cm_FidToServer()  --------------------------------

/*
 * Find the server handling a fid
 */
struct cm_server *cm_FidToServer(afidp)
    register struct afsFid *afidp;
{
        register struct cm_volume *tvp;
        register struct cm_server *tsp = (struct cm_server *)0;

        if (tvp = cm_GetVolume(afidp, (struct cm_rrequest *) 0))
        {
                /* BEGIN FRFS MODIFICATIONS HERE */
                if (tvp.type == FRFS)
                {
                        tsp = afidp.vnode.v_data;
                        if (tsp == NULL)
                        {       tsp = frfs_SelectServer(srvrptr,ANY);
                                afidp.vnode.v_data = tsp;
                        }
                } else
                /* END FRFS MODIFICATIONS HERE */

                        /*
                         * RW server is always first guy in the list
                         */
                        tsp = tvp->serverHost[0];
                }
                cm_PutVolume(tvp);  /* put back file system reference */
        }
        return tsp;                                         /* all done */
}
----------------------  end cm_FidToServer()  ----------------------------

------------------------ hs_RevokeToken ------------------------------------

/* hs_RevokeToken ( hostP , revoke4tkn)
 *
 *      Currently used to revoke tokens.  This is just a stub which pretends
 *      it has revoked all tokens as necessary.
```

```
 *
 *      If revoke4tkn includes FRFS_WRITEPROP
 *              if prop queue entry status equal BEGUN,
 *                      call frfs_CleanProp to clean up before revoking
 *              if prop queue entry status equal WRITING,
 *                      fail
 *              else
 *                      revoke the token
 *
 */

------------------------------ end hs_RevokeToken ------------------------------

------------------------------ frfs_Analyze ------------------------------
/* frfs_Analyze
 *
 *      if GOTO<SERVER> return code,
 *              if GOTOSERVER,
 *                      send RPC frfs_WhichServer to get server for connection
 *              else
 *                      get server of type from frfs_SelectServer
 *              set the private data region to the correct server
 *              get new tokens from the new server, return old tokens to old
 *              return 1 (causes the loop calling cm_Conn() to reiterate
 *
 *      CALLS:
 *              frfs_SelectServer - select server for retry
 *              frfs_ChangeTokens - get tokens from new server
 *              cm_PutConn        - return connp
 *
 *      RPCS:
 *              frfs_WhichServer        - gets server to connect to
 *      RETURNS:
 *              0 - don't retry
 *              1 - retry
 */
frfs_Analyze(errorCode, fidp, connp)
{
        if (errorCode == GOTORW)
        {       oldserver = serverp;
                serverp = frfs_SelectServer(fidp,RW);
                fidp->vnode->v_data = serverp;
                frfs_ChangeTokens(fidp,oldserver,serverp,rreqp);
```

```
                cm_PutConn(connp);
                return 1;
        } else if (errorCode == GOTOPRINCIPAL)
        {       oldserver = serverp;
                serverp = frfs_SelectServer(fidp,PRINCIPAL);
                fidp->vnode->v_data = serverp;
                frfs_ChangeTokens(fidp,oldserver,serverp,rreqp);
                cm_PutConn(connp);
                return 1;
        } else if (errorCode == GOTOANYSERVER)
        {       oldserver = serverp;
                serverp = frfs_SelectServer(fidp,ANY);
                fidp->vnode->v_data = serverp;
                frfs_ChangeTokens(fidp,oldserver,serverp,rreqp);
                cm_PutConn(connp);
                return 1;
        } else if (errorCode == GOTOSERVER)
        {       oldserver = serverp;
                serverp = RPC (frfs_WhichServer(afsFid));
                fidp->vnode->v_data = serverp;
                frfs_ChangeTokens(fidp,oldserver,serverp,rreqp);
                cm_PutConn(connp);
                return 1;
        }
}

------------------------------ end frfs_Analyze ------------------------------

------------------------------ cm_ServerDown ---------------------------------
/* cm_ServerDown
 *
 *              - call frfs_serverDown to set server down
 *
 */
------------------------------ cm_ServerDown ---------------------------------
```

4.13.2 Exception Token Management

```
------------------------- frfs_EXaddfile  ------------------------------------

/* frfs_EXaddfile (AFSfid, token, node, timestamp)
 *
 *      Adds an entry to the exception token table
```

```
 *
 *      If I am RW
 *              add afsFid entry to table
 *              add token and node to token/node pair list
 *              set timestamp
 *      else
 *              add afsFid entry to table
 *              set timestamp
 *
 *      CALLS:
 *              frfs_EXtblhash  - get entry from hash table
 *
 *      RETURNS:
 *              TRUE
 *              FALSE - unable to expand table to add entry
 */
-------------------------- end frfs_EXaddfile  --------------------------------

-------------------------- frfs_EXaddnode -------------------------------------

/* frfs_EXaddnode (EXtblentry, token, node, timestamp)
 *
 *      if node exists,
 *              update token
 *              update timestamp
 *      else
 *              add token/node pair to entry
 *              update timestamp
 *
 *      RETURNS:
 *              EXISTED - node existed and was updated
 *              ADDED   - node did not exist and was added
 *              FAILURE - unable to expand table and add entry
 */

-------------------------- end frfs_EXaddnode ---------------------------------

-------------------------- frfs_EXcheck ---------------------------------------

/* frfs_EXcheck
 *
 *      Started in background when the RW FRS is initialized.  Checks the
 *      exception table at the RW and calls tkm_LookupToken for all tokens
```

```
 *      in the table.   If the token has timed out or is no longer valid,
 *      releases the token at the RW node.
 *
 *      CALLS:
 *              tkm_LookupToken - looks up all token information for token
 *              frfs_ReleaseToken - releases token from exception table
 *
 *      RETURNS:
 *              VOID
 */
frfs_EXcheck()
{
        while (forever) /* this process runs in background continually */
        {   for (i = 0; i < HASH1; i++)
            {   for (j = 0; j < HASH2; j++)
                {       entry = extbl[i][j];
                        if (entry == NULL)
                                continue;
                        do
                        {  do
                           {   if (tkm_GetToken (entry.AFSFid, entry.node.token)
                                                == TKM_ERROR_TOKENCONFLICT)
                                        frfs_ReleaseToken (AFSfid, token);
                                        entry.node = entry.node.next;
                               while (entry.node != NULL )
                           entry = entry.next;
                        } while (entry != NULL)

                }
            }
            sleep (30);
        }
}
---------------------------- end frfs_EXcheck ---------------------------------

---------------------------- frfs_EXtblhash -----------------------------------

/* frfs_EXtblhash ( inode, request, newentry )
 *
 *      Manipulates the exception table entries.  Each entry in the exception
 *      table is a pointer to a linked list of exception tokens whose inodes
 *      hash to this position within the table.
 *
```

```
 *      If request is -1, remove entry.
 *      If request is  0, return current entry.
 *      If request is  1, replace current entry with newentry.
 *
 *      RETURNS: Pointer to structure extoken.
 *               NULL
 */
frfs_EXtblhash (inode, request, newentry)
{
        i = inode % HASH1;
        j = i % HASH2;

        if (request == -1)
        {       extbl[i][j] = NULL;
                return (NULL);
        }

        if (request)
                extbl[i][j] = newentry;
        else
                if (extbl[i][j] == NULL)
                        extbl[i][j] = malloc (sizeof struct(extokens));

        return extbl[i][j];
}
--------------------------- end frfs_EXtblhash ------------------------------

--------------------------- frfs_EXrmfile  ----------------------------------

/* frfs_EXrmfile (AFSfid)
 *
 *      Removes file entry from token table
 *
 *      For each entry in propagation queue
 *              If queue.entry.afsFid is AFSfid,
 *                      return
 *
 *      find entry in hash table
 *      unlink entry
 *      return entry to free pool
 *
 *      RETURNS:
 *              VOID
```

```
 */
---------------------------- end frfs_EXrmfile ------------------------------

---------------------------- frfs_EXrmnode  ---------------------------------

/* frfs_EXrmnode (AFSfid, node)
 *
 *      Removes node from exception table entry
 *
 *      find entry in hash table
 *      unlink node
 *      return node to free pool
 *
 *      RETURNS:
 *              VOID
 */

---------------------------- end frfs_EXrmnode ------------------------------


---------------------------- frfs_FreeToken ---------------------------------

/* frfs_FreeToken (AFSfid, token)
 *
 *      Intercepts tkm_FreeToken calls.
 *
 *      If token is an exception token and this is the RW, call
 *      frfs_ReleaseToken.  Otherwise, pass this on to the token manager.
 *
 *      CALLS:
 *              frfs_GetNode - gets the node which is freeing this token
 *                             from the protocol exporter
 *              frfs_ReleaseToken - removes token from exception table
 *                              and tells replicas if this is the last entry
 *
 *      RETURNS: return code from token manager
 */
frfs_FreeToken (AFSfid, token)
{
        if (iam == RW && token == exception)
        {       node = frfs_GetNode(AFSfid, token);
                frfs_ReleaseToken (AFSfid, token, node);
        }
```

```
        return (tkm_FreeToken (AFSfid, token));

}
---------------------------- end frfs_FreeToken ------------------------------

---------------------------- frfs_GrantEx ------------------------------------

/* frfs_GrantEx (AFSfid, token, timestamp)
 *
 *      RPC sent by the RW to grant and exception token to the client.
 *
 *      add entry to exception token table  (avoids race conditions)
 *
 *      get the token from local token manager (which revokes all outstanding
 *              tokens in order to grant it)
 *      if unable to get token,
 *              return FAILURE
 *              remove entry from exception token table
 *
 *      release token
 *      return TRUE
 *
 *      CALLS:
 *              tkm_GetToken    - get exception token from local TKM
 *              frfs_EXaddfile  - adds file to exception token table
 *              frfs_EXaddnode  - addnode to exception token table entry
 */
---------------------------- end frfs_GrantEx --------------------------------

---------------------------- frfs_KeepCurrent --------------------------------

/* frfs_KeepCurrent (AFSfid, token)
 *
 *      This routine runs in background whenever it is necessary to keep
 *      a token from timing out while manipulating it in the replication
 *      server.  It sleeps for 10 seconds less than the timeout period
 *      and then asks for the token again.
 *
 *      When the token no longer needs to be kept valid, frfs_KillCurrent
 *      will kill it.
 *
 *      RETURN:  void
 */
```

```
frfs_KeepCurrent (AFSfid, token, node)
{
        while (true)
        {       sleep(timout-10);

                /* renew the token */
                token = tkm_GetToken(AFSfid, token, node);
                if (token != VALID)
                {       print error;
                        break;
                }
                timeout = token.timeout;

        }
}
---------------------------  end frfs_KeepCurrent ---------------------------

---------------------------- frfs_EXfindfile ---------------------------------

/* frfs_EXfindfile (afsFid)
 *
 *      Looks up the entry in the exception table.
 *
 *      CALLS:
 *              frfs_ExTblHash - get entry from hash table
 *
 *      RETURNS:
 *                pointer to entry
 *                NULL
 */
frfs_LookupEx (AFSfid)
{
        entry = frfs_ExTblHash (AFSfid.vnode.inode, 0, NULL);

        while (entry.next != NULL)
        {       if (entry.AFSfid == AFSfid)
                        break;
                entry = entry.next;
        }

        if (entry.AFSfid != AFSfid && entry.next == NULL)
                return (FALSE);
```

```
        return (TRUE);
}
---------------------------  end frfs_EXfindfile ---------------------------

---------------------------  frfs_EXfindnode -----------------------------

/* frfs_findnode (EXtableentry, node)
 *
 *      Finds node in exception table for this entry
 *
 *      RETURNS:
 *              pointer to node entry
 *              NULL
 */

---------------------------  end frfs_EXfindnode ---------------------------

--------------------------- frfs_ReleaseToken -------------------------------

/* frfs_ReleaseToken (AFSfid, token, node)
 *
 *      Called from either frfs_FreeToken or frfs_EXcheck.
 *
 *      lookup node in exception table
 *              if this is last node,
 *                      remove entry
 *                      send RPC to all replicas in responsible list
 *              else
 *                      remove node from entry
 *      CALLS:
 *              frfs_EXfindfile - finds file entry in exception table
 *              frfs_EXfindnode - finds node in entry
 *              frfs_EXrmfile   - remove file entry from exception table
 *              frfs_EXrmnode   - remove node from entry
 *
 *      RPCS:
 *              frfs_EXfree     - this exception table entry has been freed
 *
 *      RETURNS:  void
 */
---------------------------  end frfs_ReleaseToken -------------------------

--------------------------- frfs_RenewTokens -------------------------------
```

```
/* frfs_RenewTokens (hostp, tokenCount, tokensToRenew, newExpTime, refTime)
 *
 *      Wraps around calls to tkm_RenewTokens
 *
 *      If this is the RW,
 *              For each exception token in the list,
 *                      If token is in the exception list,
 *                              update timestamp
 *                      else
 *                              report error
 *                              add to exception list
 *                              notify other nodes as necessary
 *
 *              pass on to tkm_RenewTokens
 *
 *      CALLS:
 *              frfs_EXfindfile - finds file in exception token table
 *              frfs_EXfindnode - finds node in exception token entry
 *              tkm_RenewTokens - renew the token at TKM
 *
 *      RPCS:
 *              frfs_GrantEx    - add this token to exception table
 *
 *      RETURNS:
 *              return code from tkm_RenewTokens
 */

------------------------- end frfs_RenewTokens -------------------------------

--------------------------- frfs_StripEx -------------------------------------

/* frfs_StripEx (token)
 *
 *      Called after tokens granted form RO token manager.  Removes
 *      all exception tokens from the token before returning the value to the
 *      cache manager.
 */

---------------------------- end frfs_StripEx ---------------------------------

---------------------------- frfs_EXupdate  -----------------------------------

/* frfs_EXupdate ( AFSfid, token, timestamp, node )
```

```
 *
 *      Add node and token to the entry in the exception table.
 *      Update the timestamp.
 *
 *      CALLS:
 *              frfs_EXfindfile - finds file entry in exception table
 *              frfs_EXaddfile  - adds file entry to exception table
 *              frfs_EXfindnode - finds node in file entry
 *              frfs_EXaddnode  - adds node to file entry
 *
 *      RETURN: void
 */
---------------------------  end frfs_EXupdate  ----------------------------
```

4.13.3 Propagation

```
------------------------ frfs_CommitFile() ---------------------------------
/* frfs_CommitFile (calledby, afsFid, node, token)
 *
 *      Keeps table of files requiring full status writes
 *
 *      If calledby a write system call,
 *              if a data write,
 *                      if afsFid is in list,
 *                              if (current_time - last_time) < been_too_long,
 *                                      return SUCCESS
 *                      commit count status write
 *                      send RPC to all nodes in responsible list to propagate
 *                      if afsFid is not in list,
 *                              create entry
 *                      zero list of nodes to fail
 *                      update time
 *              else if status write,
 *                      if commit count status write
 *                              return SUCCESS
 *                      else
 *                              remove afsFid entry from list
 *                              return SUCCESS
 *
 *      Else if calledby a token revoke call,
 *              if afsFid is in list,
 *                      if token revoking for is PROP token,
 *                              if node is in entry
 *                                      return FAILURE
```

```
 *                              else
 *                                      add node to list of nodes to fail
 *                      commit count status write
 *                      send RPC to all nodes in responsible list to propagate
 *                      update time
 *              return SUCCESS
 *
 *      CALLS:
 *              ### How do we do commit count status writes? ###
 *              frfs_CFfindfile - finds file in list
 *              frfs_CFfindnode - finds node in entry
 *              frfs_CFaddfile  - adds file to list
 *              frfs_CFaddnode  - adds node to entry
 *              frfs_CFrmentry  - removes entry from list
 *
 *      RPCS:
 *              frfs_PropFile   - tells replica to propagate file
 *
 *      RETURNS:
 *              SUCCESS
 *              FAILURE
 */

-------------------------- end frfs_CommitFile() ------------------------------

-------------------------- frfs_CleanProp()  ----------------------------------

/* frfs_CleanProp(propq,serverptr)
 *
 *      looks at the status of the propq and cleans up as necessary
 *
 *      if BEGUN
 *              free the FRS_READPROP token for serverptr
 *              free the FRS_WRITEPROP token
 *              set status to HOLDING
 *
 *      if WRITING (we have disk corruption so reget the file immediately)
 *              free the FRS_READPROP token for serverptr
 *              free the FRS_WRITEPROP token
 *              increment entry error number
 *              if entry error number > 3
 *                      set status to ERROR
 *              else,
```

```
 *                      set status to WAITING
 *
 */     RETURN void

------------------------- end frfs_CleanProp()  -----------------------------

------------------------- frfs_PropAnalyze() --------------------------------

/* frfs_PropAnalyze(errorCode, serverptr)
 *
 *      if this was the RW
 *              return not to try again
 *      if the connection failed,
 *              set the server to DOWN
 *              select a new server
 *              return to try again
 *
 *      if the token manager was unable to grant the FRS_READPROP token,
 *      either someone else is propagating from this file or it is
 *      being modified.  At any rate, we will wait and try again later.
 *
 *      return to not try again (success or unable to get a new server)
 *
 *      RETURNS:  0 - do not try again
 *                1 - try again
 */
frfs_PropAnalyze(errorCode, serverptr, fid, connP)
  int errorCode;
  struct cm_server *serverptr;
  struct afsFid *fid;
  struct cm_conn *connP;
{
        if (serverptr.type == RW)
                return (0);

        if ( !connP || (errorCode == -1) )  /* no connection was made */
        {       frfs_FRSServerDown();
                if (frfs_SelectServer( fid, PRINCIPAL ))
                        return (1);
        }

        return (0);
}
```

```
------------------------- end frfs_propAnalyze() ----------------------------

------------------------- frfs_PropIn()    ----------------------------------

/* frfs_PropIn (propq.entry)
 *
 *      Propagates the file from the appropriate server.
 *
 *      If FRS status is NORWREPLICA,
 *              set server to queue entry server
 *      Else if FRS status is VALID
 *              if I am PRINCIPAL,
 *                      set server to RW replica
 *              else
 *                      set server to PRINCIPAL replica
 *
 *      Send RPC to get FRS_READPROP token and status
 *      Keep FRFS_READPROP token from timing out
 *      Get FRFS_WRITEPROP from local TKM
 *      Keep FRFS_WRITEPROP token from timing out
 *      set propq.entry.status to BEGUN
 *      if (commit count < propq.commitcnt)
 *              Send RPC to fetch the data
 *      Set up pipe to read data directly to disk
 *      set propq.entry.status to WRITING
 *      Write status to disk
 *      Release tokens
 *
 *      If error,
 *              if error number < 3,
 *                      increment error number
 *              else,
 *                      zero error number
 *                      set propq.entry.status to ERROR
 *                      set last try time
 *                      log error
 *                      return VOID
 *              set propq.entry.status to WAITING
 *              call frfs_PropIn to try again
 *              return VOID
 *
 *      Update LWM
 *      Remove QUEUE entry
```

```
 *      return VOID
 *
 *      CALLS:
 *              SAFS_WriteData  - write data to disk
 *              SAFS_WriteStatu - write status to disk
 *              frfs_PropIn     - try to propagate the file again
 *              tkm_GetToken    - get FRFS_WRITEPROP token
 *              frfs_UpdateLWM  - updates low water mark for replica
 *              frfs_PQrmentry  - remove entry from prop queue
 *
 *      RPCS:
 *              AFS_FetchStatus - gets file status (get additional READPROP tkn)
 *              AFS_FetchData   - gets file data
 *
 *      RETURNS:
 *              VOID
 */

------------------------- end frfs_propin()  -------------------------------

------------------------- frfs_ReadPropQ()  -------------------------------

/* frfs_ReadPropQ()
 *
 *      This is a background activity started at initialization time.
 *
 *      It waits for an entry to appear in the propagation queue and processes
 *      it appropriately.
 *
 *      Loop forever:
 *              zero holding counter
 *              wait for new entry to appear in queue
 *                      if FRS status is RECONCILING or INVALID
 *                              wait for status change
 *                      for each entry in queue
 *                              if status is WAITING
 *                                      propagate file
 *                              else if status is HOLDING1
 *                                      increment hodling counter
 *                      if holding counter
 *                              for each entry in queue
 *                                      if status is WAITING or HOLDING1
 *                                              set status to WAITING
```

```
 *                                                    propagate file
 *      End loop
 *
 *      CALLS:
 *              frfs_PropIn     - propagate this file
 *
 *      RETURNS:
 *              never
 */

-------------------------- end frfs_ReadPropQ() ---------------------------

-------------------------- frfs_UpdateLWM() -------------------------------

/* frfs_UpdateLWM (commit_count)
 *
 *      loops through the prop queue looking for commit counts
 *
 *      if no entry with a commit count lower than commit_count is found,
 *      set aggregate commit count to commit_count
 *
 *      RETURN: void
 */
-------------------------- end frfs_UpdateLWM() ---------------------------
```

4.13.4 Node Selection

```
----------------------  frfs_ChangeTokens()  -------------------------------

/* frfs_ChangeTokens(afsFid,oldserver,newserver,rreqp)
 *
 *      for each entry in the scache tokenList, get a new token from the
 *      new server and call the old server to release the token
 *
 *      NOTE:  we do not modify the tokenList since the tokens will still
 *      remain.  Instead, we handle the AFS_GetToken call ourselves.
 *
 *      RETURN: void
 */
frfs_ChangeTokens( fidp, oldsvr, newsvr, rreqp)
  struct afsFid *fidp;
  struct cm_server *oldsvr, newsvr;
  struct cm_rrequest rreqp;
```

```
{
        struct cm_scache *scp;
        struct afsToken *token, *realToken;
        struct cm_tokenList *tlp;
        register struct cm_conn *connp;
        register long code;
        struct afsFetchStatus OutStatus;
        struct afsVolSync tsync;
        long startTime;

        scp = cm_FindSCache(fidp);

        for ( tlp = (struct cm_tokenList *) scp->tokenList.next;
              tlp != (struct cm_tokenList *) &scp->tokenList;
              tlp = (struct cm_tokenList *) tlp->q.next )
        {
                if (tlp->token.expirationTime < now)
                        continue;       /* expired, skip it */

                /* get token from new server */
                token = (struct cm_tokenList *) tlp->token;
                startTime = osi_Time();
                cm_StartTokenGrantingCall();
                do {
                    if (connp = cm_Conn(fidp,AFS_FSSERVICEID,rreqp))
                        code = AFS_GetToken(connp->connp,fidp,token,
                                &cm_hyperZero,0,&realToken,&OutStatus,&tsync);
                    else
                        code = -1;
                } while (cm_Analyze(connp,code,fidp,rreqp);
                cm_EndTokenGrantingCall();
                realToken.expirationTime += startTime;
                cm_MergeStatus(scp,&OutStatus,&realToken,rreqp);

                /* release this token from the old server */
                cm_QueueAToken(oldsvr,&token);
        }
}
----------------------  end frfs_ChangeTokens()  ----------------------------

----------------------  frfs_ConvertServer()  -------------------------------

/* frfs_ConvertServer(vldb_entry,cm_server)
```

```
 *
 *      Adds FRFS information to the struct cm_server
 *
 * RETURNS: void
 */

------------------------  end frfs_ConvertServer()  --------------------------

------------------------  frfs_GetIDserver()  --------------------------------

/* frfs_GetID erver(srvrp,serverID)
 *
 *      loop through frfs_serverid structures for serverID
 *
 *      RETURN:  struct frfs_serverid
 *               FAILURE if not found
 */

------------------------  end frfs_GetIDserver()  ----------------------------

------------------------  frfs_GetServer()  ----------------------------------

/* frfs_GetServer(frfs_server,type,offset)
 *
 *      type is either RW, PRINCIPAL or ANY
 *      if type == any, return the first UNKNOWN server
 *      else loop through the list looking for a server of type
 *              return the nth server where n == offset
 *
 *      RETURNS:  cm_server.serverID
 *                FAILURE if no more servers of type are found
 */
frfs_GetServer ( srvrp, type, offset )
  struct frfs_server *srvrp;
  int type, offset;
{
        struct frfs_serverid *idptr;
        register int i = 0;

        idptr = srvrp.RW.next;
        while (idptr != NULL)
        {       if (type == ANY)
                {       if (idptr.status == UNKNOWN)
```

```
                        return (idptr.cm_server.serverID);
                } else
                {       if ((type == PRINCIPAL && idptr.cm_server.fstore == ALL)
                                || (type == idptr.cm_server.fstore))
                        {       if ( offset == i++ )
                                        return (idptr.cm_server.serverID);
                        }
                }
                idptr = idptr.next;
        }
        return FAILURE;
}

------------------------  end frfs_GetServer()   ----------------------------

------------------------  frfs_GetVolServer()   -----------------------------

/* frfs_GetVolServer(volID)
 *
 *      searches the frfs_server array for the server frfs_server structure
 *      which matches the file system ID
 *
 *      RETURNS: pointer to struct frfs_server
 */

------------------------  end frfs_GetVolServer()   --------------------------

------------------------ frfs_OrderServers() -------------------------------

/* frfs_OrderServers ( serverList )
 *
 *      For each server in serverList
 *              Call NSS and get load average and client count
 *
 *      Reorder serverList such that the server with the lowest
 *      load average + client count is first in the list and that with
 *      the highest is last.  If two servers have the same value, randomly
 *      select the order.
 *
 *      RETURNS:
 *              void
 */
```

```
------------------------ end frfs_OrderServers() -------------------------

------------------------ frfs_PickRepl() ---------------------------------

/* frfs_PickRepl( numServers )
 *
 *      Called whenever an FRS has responsibility for more than one replica
 *      stored on a single node  with numServers the number of servers
 *      stored on this node.
 *
 *      Keeps a static counter which increments each time it i  called.
 *
 *      server selected is counter mod numServers
 *
 *      RETURNS
 *              pointer to server to be used
 *
 */

------------------------ end frfs_PickRepl() -----------------------------

------------------------ frfs_PickSS() -----------------------------------

/* frfs_PickSS (AFSfid, token, timestamp)
 *
 *      This is the main entry point into the replication server.
 *
 *      This routine is called instead of tkm_GetToken() and with the
 *      same arguments.  If this server can manage the call, it will
 *      call tkm_GetToken() and return the token.
 *
 *      If I am the RW FRS,
 *              if status is INVALID or RECONCILING
 *                      wait for status change
 *              if this is not an exception token,
 *                      call tkm_GetToken()
 *                      return token
 *              else
 *                      if token in exception table for this node,
 *                              update timestamp
 *                      else if token in exception table for another node,
 *                              call tkm_GetToken()
 *                              if success,
```

```
*                                       add node to exception table
*                                       return token
*                               return failure
*                       else (token is not in exception table)
*                               for each node in responsibility list
*                                       send RPC revoking confilcting tokens
*                                       (RPCs are sent in parallel)
*                               wait for all nodes to respond
*                               if a node timed out (no response)
*                                       add node to out_of_sync list
*                               if a node responded failure to revoke tokens
*                                       for each node in responsibility list
*                                               send RPC canceling token
*                               add token to exception token table
*                               call tkm_GetToken()
*                               if failure,
*                                       for each node in responsibility list
*                                               send RPC canceling token
*                                       remove from exception token table
*                                       return failure code
*                               return token
*
*       else if RO FRS status is VALID,
*               if this is an exception token,
*                       return GOTORW
*               else if file in exception table,
*                       return GOTORW
*               else,
*                       call tkm_GetToken()
*                       strip exception tokens from token returned
*                       return token
*
*       else if RO FRS status is NORWREPLICA,
*               if this is an exception token,
*                       return ESITEDOWN for RW replica
*               else if file in exception table,
*                       return GOTOSERVER
*               else,
*                       call tkm_GetToken()
*                       strip exceptiontokens from token returned
*                       return token
*
*       else if RO FRS status is RECONCILING
```

```
 *              wait for status change
 *      else if RO FRS status is INVALID
 *              if iam PRINCIPAL
 *                      return GOTOPRINCIPAL
 *              else
 *                      return GOTOANYSERVER
 *
 *      CALLS:
 *              tkm_GetToken()  - gets requested token for file afsFid
 *              frfs_StripEx() - strips exception token from token
 *                                      granted by RO TKM
 *              frfs_EXfindfile - lookup file afsFid in exception table
 *              frfs_EXfindnode - lookup node in file entry
 *              frfs_EXupdate   - updates entry in exception table
 *              frfs_EXaddfile  - adds an entry in exception table for afsFid
 *              frfs_EXaddnode  - adds node to file entry
 *
 *      RETURNS:  GOTORW
 *                GOTOPRINCIPAL
 *                GOTOANYSERVER
 *                GOTOSERVER
 *                return code from tkm_GetToken
 *                TKM_ERROR_TOKENCONFLICT
 */
------------------------ end frfs_PickSS() ---------------------------------

------------------------  frfs_RefreshServers()  ---------------------------

/* frfs_RefreshServers(volp,type)
 *
 *      if type == PRINCIPAL
 *              get a new principal server whose status is UNKNOWN or UP
 *      else (type == ALL)
 *              for all servers in the list,
 *                      check with the NSS to see if the server is back up
 *                      if UP, mark status UP
 *                      else replace server with one whose status is UNKNOWN
 *                              or UP
 *
 *      RETURN:  SUCCESS
 *               FAILURE if unable to find a server which is up
 */
frfs_RefreshServers( volp, type )
```

```
  struct cm_volume volp;
{
        struct frfs_server *srvrp;
        struct frfs_serverid *idptr;
        register int i,j,stat;
        struct afsHyper *srvrid;

        srvrp = frfs_GetVolServer( volp.volID);
        if (type == PRINCIPAL)
        {       for (idptr=srvrp.RW.next; idptr!=NULL; idptr=idptr.next)
                {       if (idptr.srvrp.fstore == ALL && idptr.status != DOWN )
                        {       volp.servers[1] = idptr.srvrp;
                                return SUCCESS;
                        }
                }
                return FAILURE;
        }

        for (i = 2, stat = 0; i < MAXSERVERS; i++)
        {       if (nss_isup(volp.servers[i].serverID)
                {       idptr=frfs_GetIDserver(srvrp,volp.servers[i].serverID);
                        idptr.status = UP;
                        stat++;
                }
        }

        /* if we found a server which is up, don't bother refilling the
         * the server structures at this time
         */
        if (stat)
                return SUCCESS;

        /* none of the servers in the list are up, so find some that are
         * note that we're not too picky here, we'll take anything that's up
         */
        for ( i = 2; i < MAXSERVERS; i++)
        {       for (idptr=srvrp.RW.next; itdptr!=NULL; idptr=idptr.next)
                {       if (idptr.srvrp.serverID == volp.servers[i].serverID)
                                continue;
                        if (idptr.status == DOWN)
                                continue;
                        idptr.status = UP;
                        volp.servers[i] = idptr.srvrp;
```

```
                        break;
                }
                if (idptr == NULL)
                        break;
        }
        if (i == 2)
                return FAILURE;
        else
                return SUCCESS;
}
------------------------  end frfs_RefreshServers()  -----------------  -------

------------------------  frfs_SelectServer()  ------------------------  -------

/* frfs_SelectServer(afsFid,type)
 *
 *      returns a server structure pointer of type for file afsFid
 *
 *      gets file system information for afsFid
 *
 *      if type equal RW, returns RW server (first in list)
 *      if type equal PRINCIPAL, returns principal
 *              if principal which is second in list is down, find others
 *      if type equal ANY, choose next available RO
 *      ## we might want to enhance selection of ROs to be more random
 *
 *      RETURNS: struct cm_server
 *               NOTFOUND
 */
frfs_SelectServer( fidp, type)
  struct afsFid fidp;
  int type;
{
        struct cm_volume *volp;
        struct cm_server *srvrp;

        volp = cm_GetVolume(fidp);

        if (type == RW)
                return (volp->cm_server[0]);

        if (type == PRINCIPAL)
        {       if ( volp->cm_server[1].status == UP )
```

```
                        return (volp->cm_server[1]);
                srvrp = frfs_RenewServer( volp, PRINCIPAL );
                if (srvrp)
                        return (srvrp);
                else
                        return NOTFOUND;
        }

tryagain:
        /* type == ANY */
        for (i = 2; i < MAXSERVERS; i++)
        {       if ( volp->cm_server[i].status == UP )
                        return (volp->cm_server[i]);
        }

        if (frfs_RefreshServers(volp,ALL) == SUCCESS)
                goto tryagain;
        if ( volp->cm_server[1] == UP )
                return (volp->cm_server[1]);
        if ( volp->cm_server[0] == UP )
                return (volp->cm_server[0]);

        return NOTFOUND;
}

------------------------  end frfs_SelectServer()  ----------------------------

------------------------  frfs_SetServer()  -----------------------------------

/* frfs_SetServer( cm_volume )
 *
 *      Selects MAXSERVERS servers for cm_volume.servers[] and fills in
 *      the cm_volume.servers[] array.
 *
 *      sets cm_volume.servers[0] to RW, cm_volume[1] to a principal,
 *      and the rest of the servers to the next 14 severs in ordered list
 *
 *      if the number of secondaries < MAXSERVERS-2
 *              add principals
 *
 *      RETURNS: void
 */
frfs_SetServers( volptr )
```

```
  struct cm_volume *volptr;
{
        struct frfs_server *srvrptr;
        struct afsHyper    *replicas;
        struct afsHyper    *principals;
        struct afsHyper    *ordered;
        struct frfs_serverid *idptr;
        struct attributes  *attrs;

        srvrptr = frfs_GetVolServer(volptr.volID);

        srcrptr.RW.status = UP;
        cm_volume.server[0] = srvrptr.RW.svrp;

        replicas = malloc((srvrptr.replicas+1) * sizeof(cm_server.serverID));
        principals =
                malloc((srvrptr.principals+1) * sizeof(cm_server.serverID));

        for( i = 0; i < srcrptr.principals; i++ )
                principals[i] = frfs_GetServer(srvrptr,principal,i);
        principals[i] = 0;

        /* Call the Node Status Server to prioritize the servers in the
         * argument list
         *
         * ## Define exact nodess interface call
         */
        ordered = malloc( srvrptr.principals + 1 * sizeof(cm_server.serverID));
        nss_PrioritizeNodes( &principals, &ordered );
        for( i = 0; i < srvrptr.principals; i++ )
                principal[i] = ordered[i];
        free( ordered );

        idptr = frfs_GetIDServer(srvrptr,principal[0]);
        cm_volume.server[1] = idptr.srvrp;
        idptr.status = UP;

        for( i = 0; i < srvrptr.replicas; i++ )
        {       replicas[i] = frfs_GetServer(srvrptr,ANY,i);
                if (replicas[i] == 0)
                        break;
        }
```

```
        ordered = malloc( i+1 * sizeof(cm_server.serverID));
        nss_PrioritizeNodes( &replicas, &ordered );

        for( j = 0; j < i; j++)
        {       if ( j+2 >= MAXSERVERS)
                        break;
                idptr = frfs_GetIDServer(srvrptr,replicas[j]);
                cm_volume.server[j+2] = idptr.srvrp;
                idptr.status = UP;
        }
        free (ordered);

        /* We have filled in the RW, principal and as many partials as
         * possible.  If there are more slots, fill them in with
         * remaining principals.
         */
        j+=2;
        if (j < MAXSERVERS)
        {       for( i = 0; i+j < MAXSERVERS; i++ )
                {       if ( principals[i+1] == 0 )
                                break;
                        idptr = frfs_GetIDServer(srvrptr, principals[i+1]);
                        cm_volume.server[j+i] = idptr.srvrp;
                        idptr.status = UP;
                }
        }
}
----------------------  end frfs_SetServer()  ------------------------------

----------------------  frfs_StoreVol()  -----------------------------------

/* frfs_StoreVol( vldb_entry )
 *
 *      Gets all replicas from the VLDB and stores them for future reference
 *
 *      Searches the frfs_server list for file system ID.
 *
 *      If file systemid is not found, then
 *              add a server to the frfs_server list
 *              initialize the server
 *              set frfs_server.RW to vldb_entry.servers[0]
 *              frfs_ConvertServer()
 *              set frfs_serverid.status to UP
```

```
 *              set index to 1
 *      else (this is not the first vldb_entry to be read for this file system)
 *              loop through the server list to the end
 *              set index to 0
 *
 *      starting with vldb_entry.servers[index], loop through vldb_entry.servers
 *              allocate serverid storage
 *              frfs_ConvertServer()
 *              set frfs_serverid.status to UNKNOWN
 *
 *      call frfs_SetServer() to set the servers in the file system array to
 *              the "best" servers
 *
 *      CALLS:
 *              frfs_GetVolServer - looks for entry in volume list
 *              frfs_AddVolServer - add entry to volume list
 *              frfs_ConvertServer - converts server entry into frfs entry
 *              frfs_SetServer    - set server array to 16 ordered servers
 *
 *      RETURNS: void
 */

------------------------  end frfs_StoreVol()  -------------------------------

------------------------ frfs_WhichServer() ---------------------------------

/* frfs_WhichServer (afsFid)
 *
 *      Called by CM when an errorcode of GOTOSERVER was received
 *
 *      Loops through prop queue for afsFid
 *              return server listed in queue
 *
 *      CALLS:
 *              frfs_PQfindfile
 *
 *      RETURNS:
 *              pointer to server to use for this file
 *              error if file not listed
 */

----------------------- end frfs_WhichServer() ------------------------------
```

4.13.5 Network Instability Management

```
------------------------- frfs_BeginRecon() ---------------------------------

/* frfs_BeginRecon( node type )
 *
 *      Called when PRINCIPAL or SECONDARY is trying to select a reconciler
 *      and reconcile into a NORWREPLICA partition
 *
 *      Spawn this as a background process and return immediately
 *
 *      if I am SECONDARY
 *              reconcile list contains all secondaries
 *      else if node type is SECONDARY
 *              for each node in responsible list
 *                      send RPC notifying of NoRW
 *              reconcile list contains all principals
 *
 *      for each replica in reconcile list
 *              send RPC notifying no RW
 *      wait for nodes to return
 *      for each node returning,
 *              put lwm in reconciliation table
 *
 *      select reconciler based on highest lwm and position in volume list
 *
 *      if I am reconciler
 *              send out reconciliation information
 *
 *      CALLS:
 *              frfs_Reconciler - manages actual reconciliation information
 *
 *      RETURNS:
 *              VOID
 */

---------------------- end frfs_BeginRecon() ---------------------------------

------------------------- frfs_EndRecon() -----------------------------------

/* frfs_EndRecon()
 *
```

```
 *      RPC received from reconciler
 *
 *      for each entry in exception token table,
 *              add entry to propagation queue
 *                      original commit count = current commit count
 *                      current commit count = reconciler's lwm
 *
 *      set status to NORWREPLICA
 *
 *      RETURNS:
 *              void
 */

-------------------------  end frfs_EndRecon() -------------------------------

-------------------------  frfs_FindRW() -------------------------------

/* frfs_FindRW()  (Will be replaced with keep-alive service - keep for logic)
 *
 *      runs in background when status is NORWREPLICA
 *
 *      sleeps for a maximum of 60 seconds on an address which is updated
 *      whenever a request for modification comes and the status is NORWREPLICA
 *
 *      sets static counter to time of last poll
 *      if time elapsed since last poll is greater than 5 seconds
 *              call NSS to see if RW is still down
 *
 *      if NSS indicates RW is up, set state to RECONCILING
 *              if (iam == principal)
 *                      notify all secondaries
 *                      frfs_ReplInit()
 *              else
 *                      notify principal
 *                      frfs_ReplInit()
 *
 *      RETURNS:  void
 */

--------------------------- end frfs_FindRW()  -------------------------------

--------------------------- frfs_GetExTable() -------------------------------
```

```
/* frfs_GetExTable()
 *
 *      Called by RO replica to either a principal or RW
 *
 *      If principal and status == RECONCILING
 *              wait until status changes
 *
 *      RETURNS:
 *              exception table
 *
 */

---------------------------- end frfs_GetExTable() ----------------------------

---------------------------- frfs_GetFiles() ----------------------------------

/* frfs_GetFiles ( lwm )
 *
 *      Called by RO replica to either a principal or RW
 *
 *      If principal,
 *              for each file in the propagation queue,
 *                      send propagation instruction
 *
 *      for each file in the file system
 *              check commit count against lwm
 *              if (commit count > lwm )
 *                      send propagation instruction
 *
 *      RETURNS:
 *              SUCCESS when all propagation instructions have been acknowledged
 *              FAILURE if a  propagation instruction times out
 */

------------------------- end frfs_GetFiles() ---------------------------------

------------------------- frfs_GetRecon() -------------------------------------

/* frfs_GetRecon ( filelist )
 *
 *      called by the reconciler and contains list of files it needs
 *
 *      for each file in filelist,
```

```
 *              look up locally stored file afsFid
 *                      if commitcount < stored commitcount
 *                              clear file from exception token table
 *                              put file in list to return
 *                      else if commitcount == stored commitcount
 *                              clear file from exception token table
 *                              return 0
 *                      else
 *                              add file to propagation queue; node = reconciler
 *                              return 0
 *
 *      CALLS:
 *              frfs_RPQaddfile - add file to reconciliation prop queue
 *
 *      RETURNS:
 *              list of files whose commit count is higher
 *              NULL
 *
 */

------------------------------ end frfs_GotRecon() ----------------------------

------------------------------ frfs_NoRW() ------------------------------------

/* frfs_NoRW(node, LWM)
 *
 *      RPC recieved when a node discovers there is no RW
 *
 *      If FRS status is VALID
 *              set status to RECONCILING
 *              if I am PRINCIPAL or node is SECONDARY (secondaries reconciling)
 *                      begin reconciliation (returns immediately)
 *
 *      Return my lwm
 *
 *      CALLS: frfs_BeginRecon - begin reconciliation into NOREREPLICA state
 *
 *      RETURNS:
 *              Low Water Mark
 */

------------------------------ end frfs_NoRW() --------------------------------
```

```
---------------------------- frfs_Reconciler() ------------------------------

/* frfs_Reconciler()
 *
 *      for each entry in the exception token table
 *              add file and commit count to list of files to get
 *              zero entry
 *
 *      for each node reconciling
 *              send RPC containing list of files to get
 *      wait for all RPCs to return
 *      for each node returning
 *              for each file in file list returned
 *                      add file to exception table
 *                      add file and node to reconciliation prop queue
 *      for each node reconciling
 *              send RPC indicating end of reconciliation (send recon prop q)
 *      set status to NORWREPLICA
 *
 *      CALLS:
 *              frfs_EXaddfile   - add entry to exception table
 *              frfs_RPQaddentry - add entry to reconciliation prop q
 *
 *      RPCS:
 *              frfs_GotRecon    - sends lits of files to get for reconciliation
 *              frfs_EndRecon    - end reconciliation (sends recon prop queue)
 *
 *      RETURNS:
 *              VOID
 */

------------------------- end frfs_Reconciler() ------------------------------

---------------------------- frfs_ReplInit ---------------------------------

/* frfs_ReplInit ( fsname )
 *
 *      Initialize FRS for file system fsname.
 *
 *      Set state to RECONCILING
 *      Get file system information from VLDB and CMS
 *      Verify .frfsinfo file ( ### How to handle errors? ### )
 *      Set iam to RW, PRINCIPAL, or SECONDARY
```

```
*
*       If RW,
*               allocate memory for replica lists
*               put all principals in list of nodes for which RW is responsible
*               zero lock table
*               zero RPCwait table
*               zero exception token table
*
*       else if PRINCIPAL,
*               allocate memory for secondary replica list
*               send RPC frfs_ROreconcil to RW replic
*
*               if unable to contact,
*                       for each principal replica,
*                               send RPC frfs_ROreconcil to replica
*                               if successful, break
*                       if successful,
*                               initialize for NORWREPLICA partition
*                       else
*                               for each untried replica
*                                       send RPC frfs_ROreconcil to replica
*                                       if successful, break
*                               if successful,
*                                       initialize for NORWREPLICA partition
*                               else
*                                       set state to NORWREPLICA
*                                       set poll for RW
*                                       return VOID
*                       return void
*
*               compare RW history with local history
*               if incompatible,
*                       set state to INVALID
*                       log error
*                       return VOID
*
*               set initialization number
*               zero exception token table
*               zero propagation queue
*
*               set exception token table using RW exception table
*               create prop queue entries from file returned from RW
*
```

```
 *              set state to VALID
 *              return VOID
 *
 *      CALLS:
 *              VL_GetEntry     - get volume header entry for file system
 *              CMS_GetEntry    - get file system entry from CMS
 *              frfs_ReadInfo   - read .frfsinfo file
 *              frfs_VerifyInfo - verify .frfsinfo against VLDB and CMS info
 *              frfs_ROinit     - initialize into NORWREPLICA partition
 *              frfs_PollNode   - add node to list to be polled for state change
 *              frfs_CompHist   - compare history of remote node with local
 *              frfs_EXtblinit  - read initialization table from remote node
 *                                and enter into local exception table
 *              frfs_PQinit     - read file list from remote node and enter
 *                                into propagation queue and exception table
 *                                if appropriate
 *      RPCS:
 *              frfs_ROreconcil - RPC telling replica the node is ready to
 *                                reconcile.
 *
 *      RETURNS:
 *              VOID
 *
 */
------------------------- end frfs_ReplInit ---------------------------------

------------------------  frfs_ServerDown()  --------------------------------
/* frfs_ServerDown ( server, afsFid, flag )
 *
 *      call the NSS to verify the server is down
 *      if server is up,
 *              return TRYAGAIN
 *
 *      set server to DOWN in server list
 *
 *      if flag is TRUE,
 *              if server is RW, return FAILURE
 *              select another server
 *              put new server in vnode
 *
 *      CALLS:
 *              frfs_SelectServer - select a new server
 *
```

```
 *      RETURNS:
 *              TRYAGAIN
 *              FAILURE
 *              OK
 */
------------------------  end frfs_ServerDown()  ------------------------------

------------------------  frfs_FRSServerDown()  ------------------------------

/* frfs_FRSServerDown(afsFid,serverID)
 *
 *      call t.  NSS to verify the server is down
 *      if server is up,
 *              return TRYAGAIN
 *
 *      set server to DOWN in server list
 *
 *      if server is RW
 *              set FRS state to RECONCILING
 *              if I am PRINCIPAL,
 *                      for each principal replica
 *                              send RPC notifying RW is down
 *              else
 *                      send RPC to principal notifying RW is down
 *      else if server is PRINCIPAL
 *              if another PRINCIPAL available,
 *                      send RPC to principal registering
 *                      mark principal as registered server
 *              else
 *                      send RPC to RW registering
 *                      mark RW as registered server
 *                      set up poll for principal server
 *
 *      return DOWN
 *
 *      RPCS:
 *              frfs_NoRW       - notify replica that RW is not available
 *              frfs_RORegister - register with replica for responsibility
 *
 *      RETURNS:
 *              TRYAGAIN - node is still up, try again
 *              DOWN     - node is down
 */
```

```
------------------------  end frfs_ServerDown()  ----------------------------
```

4.14 Data Structures

The following data structures are used by the **FUSION** Replication Service.

4.14.1 Modifications to DCE Data Structures

```
/* external view of the vldbentry structure as stored in the VLDB */
struct vldbentry   {
        char    name[MAXNAMELEN];     /* Volume name */
        u_long  volumeType;      /* Volume type (RWVOL, ROVOL, BACKVOL) */
/****  BEGIN FRFS MODIFICATIONS HERE ****/
        u_long  replType;             /* Full or Loose FRFS Replication */
/****  END FRFS MODIFICATIONS HERE ****/
        u_long  nServers;      /* Number of servers that have this file system */

        /* Server # for each server that holds file system */
        struct afsNetAddr       siteAddr[MAXNSERVERS];

        u_long  sitePartition[MAXNSERVERS];     /* Server Partition number */
        u_long  siteFlags[MAXNSERVERS]; /* Server flags */
        u_long  sitemaxReplicaLatency[MAXNSERVERS];   /* Per-site max latency */
        struct afsHyper VolIDs[MAXVOLTYPES];
        u_long  VolTypes[MAXVOLTYPES];
        struct afsHyper cloneId;                /* Used during cloning */
        u_long  flags;                          /* General flags */
        u_long  maxTotalLatency;
        u_long  hardMaxTotalLatency;
        u_long  minimumPounceDally;
        u_long  hardMaxReplicaLatency;
        u_long  reclaimDally;
        u_long  WhenLocked;
        u_long  spare1;
        u_long  spare2;
        u_long  spare3;
        u_long  spare4;
        char    k_principal[MAXKPRINCIPALLEN];
        char    LockerName[MAXLOCKNAMELEN];
        char    charSpares[50];
};

/*
 * One structure per file system, describing where the volume is located
 * and where its mount points are.
 */
```

```
struct cm_volume {
    struct cm_volume *next;        /* Next volume in hash list. */
    struct cm_cell *cellp;       /* this file system's cell */
    struct lock_data lock;       /* the lock for this structure */
    struct afsHyper volume;        /* This volume's ID number. */
    char *volnamep;              /* This file system's name, or 0 if unknown */
/*** BEGIN FRFS MODIFICATIONS HERE ***/
    u_long repltype;             /* full or loose FRFS replication */
/*** END FRFS MODIFICATIONS HERE ***/
    struct cm_server *serverHost
        [AFS_MAXHOSTS];          /* servers serving this file system */
    struct afsFid dotdot;        /* dir to access as .. */
    struct afsFid mtpoint;       /* The mount point for this file system. */
    struct afsHyper roVol;       /* RO filsys id assoc. with volume (if any) */
    struct afsHyper backVol;     /* BACKUP filsys id assoc. with vol (if any) */
    struct afsHyper rwVol;       /* RW file system id for this volume */
    long accessTime;             /* last time we used it */
    long vtix;                   /* file system table index */
    long copyDate;               /* copyDate field, for tracking vol releases */
    short refCount;              /* reference count for allocation */
    char states;                 /* sneak in here for alignment reasons */
};

/************* token ID *********************/
/* token ID returned by token manager */
typedef struct hyper tkm_tokenID_t;

/*********** all the different types of tokens *****************/
/* all the different types of tokens that may be requested */
typedef enum tkm_tokenType      {
  TKM_NO_TOKEN = -1,
  TKM_LOCK_READ, TKM_LOCK_WRITE,
  TKM_DATA_READ, TKM_DATA_WRITE,
  TKM_OPEN_READ, TKM_OPEN_WRITE, TKM_OPEN_SHARED, TKM_OPEN_EXCLUSIVE,
  TKM_OPEN_DELETE, TKM_OPEN_PRESERVE,
  TKM_STATUS_READ, TKM_STATUS_WRITE,
  TKM_NUKE,
/*** BEGIN FRFS MODIFICATIONS HERE ***/
  FRS_READPROP, FRS_WRITEPROP,
/*** END FRFS MODIFICATIONS HERE ***/

  /* If more members are added to the enumeration, TKM_ALL_TOKENS must   */
  /* remain last in the enumeration to avoid breaking things like arrays */
```

```
  /* indexed by instances of this type.                                */
  TKM_ALL_TOKENS
} tkm_tokenType_t;
```

4.14.2 FRS Data Structures

```
struct extokens {
        struct extokens   * previous;
        struct afsFID     afsFID;
        struct node_token * token;
        u_long              timestamp;
        struct extokens   * next;
};

struct node_token {
        struct node_token     * next;
        struct token_id_t       token;
        char                  * node;
};

struct extokens * extbl[HASH1][HASH2];

struct propq {
        struct propq    * previous;
        struct afsFID   afsFID;
        u_long          occ;            /* original commit count */
        u_long          ccc;            /* current commit count */
        u_long          timestamp;
        struct  propq   * next;
};

struct frfslock {
        struct afsFID   afsFID;
        struct afshyper nodes[MAXLOCKHOSTS];
} frfslock[MAXLOCKS];
```