FUSION Functional Specification

Locus Computing Corporation
IBM Corporation

April 26, 1991

TABLE OF CONTENTS

FUSION Functional Specification

Locus Computing Corporation
IBM Corporation

April 26, 1991

1. PRODUCT RATIONALE

1.1 BACKGROUND

The evolution of distributed computing technologies over the past twenty years has created a general consensus. Accessing remote resources should be no different from accessing local resources. In other words, the network environment should be "transparent" to the users and to the software, which should view a single, but more capable, system.

Early distributed computing solutions offered no form of transparency. They provided mechanisms to connect to remote nodes, and obtain services from those nodes, such as logging in, transferring files, and running programs. The users of these mechanism had to know explicitly which node was appropriate, had to explicitly connect to that node, and had to use special commands to do particular things. The software interfaces to these services were very different from local methods.

Second generation systems, most notably NFS, began to provide some level of transparency. It was possible to virtually "connect" remote resources such as file systems to a local machine. Once connected, the user could use that resource as if it were locally attached. In order to set up the appropriate connections for this level of function, the user or administrator had to know the location of the resources. In addition, the semantics of accessing remote devices were not always identical to that used for local accesses.

The third generation of systems provide a fully integrated transparent distributed operating system. IBM's AIX Transparent Computing Facility, or TCF, is the only commercially available system in this category. TCF provides a complete implementation of network transparency for programs and users. TCF provides the ability to access remote file system data while preserving correct Unix semantics. TCF also provides network process transparency, where processes can be started on any machine, and can move from one machine to another in the middle of operation.

The release of OSF's Distributed Computing Environment (DCE) is a major opportunity to provide an expanded version of the TCF technology on an industry standard base environment. The **FUSION** product (Full UNIX Semantics Integrated Over Networks) is a highly portable version of the TCF technology with expanded capabilities to operate across a wide range and potentially large number of network nodes.

With the **FUSION** product, networked systems can be grouped in a cluster to form a "network multiprocessor" or "loosely coupled multiprocessor" configuration. A networked MP system provides the following benefits to a potential customer:

a. Network MP systems are scalable and can be heterogeneous; they can be built incrementally with existing hardware or combined with new hardware.

b. Network MP systems can scale to a much higher number of processors, and can in fact be composed of symmetric MP systems. Memory contention is avoided by having private memory per node.

c. Network MP systems provide flexibility in local autonomy; a feature not typically found in the symmetric MP world.

d. Network MP systems provide flexibility in office proximity and physical dispersion. One can have clusters that are multiple CPUs in a single box as well as work groups that are cross-country.

The stage is set for an evolutionary growth in transparent distributed computing technology. LAN technology such as optical networks is becoming cheaper and more powerful. Consequently, less and less performance penalty is extracted when accessing remote resources. Processor and memory hardware are becoming cheaper. This makes extremely powerful multiprocessor systems more affordable than ever.

1.2 FUSION: PRODUCT DEFINITION

The **FUSION** Product can be viewed as containing three layers. The lowest layer extends the services available in OSF's DCE with a set of distributed file system enhancements and remote execution facilities that can be used within an environment of heterogeneous machines from multiple vendors. The highest layer builds onto the lower layer a clustering mechanism that provides a consistent name space between the machines, and a set of extensions to allow the processing load to be automatically balanced among the machines in the cluster. In a homogeneous environment, the load- leveling capabilities can be extended so that the collection of machines appear as a network multi-processor. A third layer which functions somewhat independently of the other two provides a file system replication service that is more general than that available from DCE. These layers are called the Distributed Execution Environment, the Cluster Environment, and the File System Replication Service.

The principal reason for separating the functions into these layers is the variety of requirements in the marketplace. There is strong demand today for products that provide enhanced distributed file system support and the ability to run processes on other machines from a variety of vendors, without requiring the

grouping of machines into clusters. Vendors are willing to support this type of product because it does not require close coordination with other vendors. There is also demand in the market for a more cohesive computing environment spread among many machines, but some vendors are not willing to provide such products at this time. Generalized file system replication is useful in both of the above scenarios but is not strictly necessary in either one. By separating the functions along these lines, the **FUSION** product can obtain a wide acceptance among vendors, and also provide a more complete distributed system solution.

1.3 GOALS

There are several goals driving the **FUSION** project. These include a set of customer goals, a set of architecture goals, a set of functionality goals, and a set of performance goals.

1.3.1 CUSTOMER GOALS

FUSION attempts to satisfy two major goals of the customer. These are achieving the correct ratio of price to performance, and achieving enhanced productivity of existing resources.

There are several ways that **FUSION** offers reasonable price/performance. It allows existing single site applications to take advantage of a distributed computing environment. No changes are required in the application software to operate in this manner. It provides a software solution to build a "network multiprocessor", where each node can have private memory, and the processors can be put in a single cabinet or physically distributed, even beyond a single building or even a single city. Such an architecture can be scaled to sizes much larger than a conventional multiprocessor, even to hundreds or thousands of processors. Load leveling can be easily built within the **FUSION** environment. Such load leveling can be automatic, application specific, and controlled by users and system administrators. Such functions allows hardware to be shared more effectively, thus serving greater computing requirements with a smaller investment.

The customer is continually searching for methods to enhance productivity. The **FUSION** system should thus support an incremental growth strategy so that additional computing resources can be added as the needs grow without a serious disruption of service. In a similar manner, the system should degrade gracefully if components of the system fail. This offers greater availability than a single large system which impacts all users if a failure occurs. The system should make the network transparent to users, both to support program portability as well as allowing users to move to the larger system without incurring additional training costs. The system should supply a high level distributed programming model that complements RPC mechanism. Finally, the system should provide a migration path, to new hardware, between different operating systems and also between different versions of the same

operating system.

1.3.2 CUSTOMER CONSTRAINTS

While the goals mentioned in the previous section are important for satisfying the requirements of the customer, these goals must be satisfied within a set of identifiable constraints. These constraints include binary compatibility, adherence to standards, acceptable performance, easy installation and administration, understandability, affordability, tunability for a variety of configurations, scalability to large numbers and wide areas, flexibility in configurations, availability from multiple vendors, and support within a broad range of environments.

Binary compatibility is an important constraint, because customers have made large investments in application software and need to continue to utilize that software as the environment evolves. Thus **FUSION** must not require changes to applications that wish to take advantage of the environment it offers.

Standards within the computing industry continue to appear. They are important because such standards define what interoperability and portability the users expect. The **FUSION** system must operate within the framework of existing and forthcoming industry standards. Equally important, **FUSION** must be suitable as an industry standard.

The **FUSION** system must perform well from the perspective of the users. Bad performance would limit the suitability of the system to most customers. This includes the impact on performance of existing operations as well as the performance of new functions.

The **FUSION** system must be easy to install. Complex installation procedures discourage any installation, and will impact the popularity of the software. Similarly, it must be relatively easy to administer. This includes making administration of many machines much easier than the administration of those machines running separate independent systems.

The **FUSION** system must be understandable by the users, system programmers, system administrators, and even MIS executives. To achieve this, the system should introduce few new concepts.

The **FUSION** system must be affordable. If the system is too expensive, it will not be a viable solution to many installations.

The **FUSION** system must be tunable to operate within a large variety of configurations. The default tuning should be suitable for most small configurations. When large configurations require special tuning, **FUSION** should supply the necessary data collection tools and tuning guidelines, as well as adequate mechanisms to allow tunging.

The architecture and design of the **FUSION** system should permit the number of processors to scale. This includes scaling to very large numbers of

processors. Similarly the design should not preclude a wide geographic distribution of the processors, which might entail greater network latencies.

The **FUSION** system should allow for flexible configurations. This includes the ability to join or not join with other nodes, and the ability for a node to change between clusters.

The **FUSION** system should be general enough to be supportable by more than one vendor. Today's customers are demanding such flexibility today, so that they can choose from a much wider selection of hardware and software options, without discarding much of their existing investment. They want to avoid being locked in by one particular vendor.

Finally, the **FUSION** system should provide support within a broad range of environments. This includes suitability for operating beyond the confines of the UNIX environment into proprietary OS offerings. It also should be suitable for a wide range of hardware options and also networking technologies.

1.3.3 ARCHITECTURE GOALS

From the above stated customer goals and customer constraints, a set of goals for the **FUSION** architecture have been identified. The architecture goals of **FUSION** include portability, scalability, modularity, flexibility, installability, and support of an open architecture.

One of the key goals of the **FUSION** architecture is portability. Portability implies that the system should be built with very few hooks in the base operating system and very few hooks in DCE. The system should be layered above DCE to the degree possible. The architecture should not require any assumptions about the underlying hardware, and the requirements on the underlying operating system should be as general as possible. Doing so would make it applicable to UNIX and non-UNIX systems that support DCE. Special dependencies should be extremely limited, and avoided when at all possible.

Another important goal is scalability. Configurations containing thousands of nodes are quite conceivable with existing and emerging technologies. The **FUSION** architecture should permit reasonable operation within such large systems. With such large systems, some of the components may be spread out over large distances. The architecture should support such widespread configurations. Also, in large possibly widespread configurations, some autonomy is required.

Modularity is a key goal of the **FUSION** architecture. The architecture should support a separation of function into individual modules. The modules should be able to plug and play together. The specifications should be clear and accurate enough to allow alternate implementations of the same technology to work effectively together.

Flexibility is an important architecture goal so that the system can be adapted to suit a wide variety of uses. The architecture should not be dependent on any particular hardware feature, but should still be able to take advantage of special hardware when it exists. It should be operatable over a wide variety of networking technologies, from Wide Area Networks to Local Area Networks and even high speed backplane networks. The architecture should be resilient to networking problems. A single node should still be fully functional for the user. The network protocols and data should be extensible.

The **FUSION** system should be easily installed. It should be possible to install it onto running systems. Limited human interaction should be required at installation time. The system should provide optional tuning capabilities. The architecture should provide for different subscription levels to obtain different degrees of cooperation.

Finally, the architecture for **FUSION** should be open. The goal for being an open architecture is essential for its widespread acceptance as an industry standard. To be open, the base hooks should be as general as possible, to allow for competing implementations. The required hooks into the underlying environments should be made available to the Open Systems Foundation, Unix International, and other interested standards bodies. Finally, the network protocols and semantics should be published and made available to the industry.

1.3.4 FUNCTIONALITY GOALS

Within the above framework, the following functionality goals have been identified. The operation within clusters that appear as a network MP environment, operation in an more autonomous environments, support of automatic load leveling, and reasonable autonomy and security in a networked environment. Other functional goals include providing a superset of TCF and DEcorum functionality, applicability to UNIX and non-UNIX environments, and providing the basis for highly available computer systems.

A principal functionality goal of **FUSION** is to provide software support of a network multiprocessor. This means that **FUSION** should provide location, name, access, semantic, and performance transparency for system resources that are spread throughout the nodes. Each node must be potentially full function when not cooperating with other nodes. The system should not require shared memory. Invocation and dynamic load balancing should be provided to spread the load among the processors. Single-site POSIX semantics should be maintained so the available existing software can be used. The system should be heterogeneous to provide an effective interoperation between diverse hardware. Finally, the system should scale to thousands of processors, something not generally achievable with shared memory multiprocessors.

Another functionality goal is to provide a more autonomous environment for some applications. This implies a relaxation of the file system name space consistency, while still providing most of the functions. Such an environment should at least scale to the size of a DCE cell. This environment should provide full process transparency with POSIX semantics, support for heterogeneity, and load leveling data and primitives. However, automatic load leveling would not be performed due to the potential name space inconsistencies on different nodes.

Load leveling is an important functional goal because it allows the users to make effective use of the available machine resources. The load leveling support should be automatic, but controllable by administrators and users. A programming interface is also provided so that particular applications can make their own load leveling decisions. Load leveling can be done at program invocation time as well as after the program is running. The **FUSION** system should permit alternate load leveling policies to be installed via extension mechanisms. Policies modules should be able to account for batch versus interactive machines and loads, fairness, and a flexible preference list. Query and diagnostic services should be provided with the load leveling. The load leveling functionality should be applicable to RPC servers as well as general remote processing.

To maintain autonomy, it the **FUSION** system should permit particular users to be restricted to a certain set of nodes in a fairly static way. Also, privileged users on particular node should be able to dynamically restrict use of that node.

In the area of security, the **FUSION** system should use authenticated RPC mechanisms for all of its services. This avoids problems of phony messages and spoofing. Also, certain features such as performing some kinds of mounts and the semantics of setuid programs should be optionally restricted.

Functionally, it is desirable to have **FUSION** provide a superset of TCF and DEcorum functionality for several reasons. One reason for providing a similar functionality is the existing user base for TCF. To provide these users a migration path a similar function is required. A reason for providing a superset of TCF functionality is that lessons learned from the TCF system can be applied to make an even better system. For example, having all nodes be updated on the status of all other nodes in the same cluster requires complex and time consuming algorithms that do not scale effectively. Limiting the number of nodes operating together in a cluster to thirty-one limits the configuration options in modern hardware environments. Similar arguments apply to DEcorum functionality.

Another functional requirement for **FUSION** is to be applicable to both UNIX and non-UNIX environments. While UNIX with DCE is the target of the initial work, it is desirable to consider expansion beyond the UNIX framework as DCE becomes available on other OS platforms.

A final functional goal of **FUSION** is to offer a high availability system. Where possible, the **FUSION** design should minimize the need for particular nodes to stay up. As an example, when a set of processes that are sharing an unnamed pipe that was created on node A and all move to other nodes, then node A should no longer be required to support the pipe. Only when specific resources of a node are involved should that node be required.

1.3.5 PERFORMANCE GOALS

A set of performance goals for **FUSION** has also been defined. The system with only base hooks added should see no impact. The system with the **FUSION** installed but working locally should see very little impact. The aggregate performance of a set of machines or processors on realistic benchmarks should be significantly improved. Small configurations should require no special tuning to operate effectively. Large configurations should have data, tools, and guidelines for performance tuning. Performance requirements are specified in Chapter 6.

1.4 ASSUMPTIONS

Some assumptions have been made for the specification and design of **FUSION**. This section outlines some of those assumptions.

1.4.1 BASE SYSTEM ASSUMPTIONS

The initial design work has focussed on a system that can be built on an AIX Version 3 or an OSF/1 base. While this does not impact the design, there are some underlying assumptions that these systems satisfy.

It is assumed that the base kernel provides common hooks for remote file system support, such as the vnode layer and file operations table. It is also assumed that the pipe implementation utilizes this interface.

Much of the design assumes that it is possible to load in kernel extensions to provide the extended **FUSION** function within the kernel. The interface between these extensions and the base kernel will be through well defined hooks. Many of these hooks (e.g., the vnode layer) exist already. Other hooks are new and are defined by **FUSION** in a way that is applicable to a wide base of kernels.

Similarly, it is assumed that it is possible to alter the kernel linkage to substitute routines for policy modules and system call entry points. It is also assumed that user supplied code may be substituted for **FUSION** supplied routines to alter policy.

It is assumed that the new hooks defined for **FUSION** will eventually be accepted by code owners. If this is not the case, part of the **FUSION** porting effort would be in providing these hooks. This is viewed as more intrusive than desirable.

Much of the **FUSION** function is implemented by shipping the functions from the node where the user of a resource is operating to the node where the resource is currently controlled, via the DCE RPC mechanisms. It is assumed that the RPCs will be serviced by kernel threads. It is thus assumed that these kernel threads are inexpensive. Since some of these operations may sleep during the course of satisfying the request, it is assumed that the kernel threads are plentiful so that deadlock is avoided. These assumptions permit the design to utilize the RPC architecture in the way it was intended. If these assumptions are not satisfied, other designs could be proposed that rely on message based semantics. Such a mechanism is used in the TCF system. However, building message passage systems with RPC mechanisms is probably more inefficient than a directly implemented message passing system would be.

1.4.2 DCE ASSUMPTIONS

Not many assumptions are made about DCE. The design is based on features found in the early releases of DCE. It is possible that late changes to DCE could adversely affect the **FUSION** design, but that is not anticipated.

The **FUSION** architecture assumes that as a practical limit, there are no more than 4096 nodes within a cell. This limit is not hard and fast, but supporting larger cells requires a process ID mapping mechanism that

There are hooks required in DCE to implement /*(Fn. This design assumes that these hooks will be accepted by OSF for inclusion in the first release of DCE.

1.5 DEFINITIONS

Within this document, several technical terms are used where a particular concept is intended. To avoid confusion, this section defines those terms.

cell — The term cell arises from the DCE architecture itself. It refers to several things, though primarily it is the domain of the DCE authentication server. The **FUSION** definition of cell is thus the same as the DCE cell, to avoid any confusion.

cluster — The term cluster refers to a set of machines that are operating in close coordination together. In particular they share a common name space (or a large portion of a common name space). Because of this naming transparency, it is possible to apply automatic load leveling functions within a cluster.

groups — Within **FUSION**, groups are an administrative domain for keeping track of node information. Groups are defined because it is neither practical to have all nodes store the information about all other nodes nor to have a single

node store the information about the node. Providing groups permits a two level mechanism to be defined. Membership in a group is defined in a relatively static way. Being in one group or another does not impair function in any way, nor does it imply or prevent membership in a cluster.

group

Each group as defined above has one node at any given time designated as a server for the group. This node stores information about the nodes within the group, and communicates with other group servers to exchange information about nodes in each other's groups when necessary. The group server is chosen in a dynamic way. If a group is partitioned then each partition has an active group server. When partitions are rejoined a single group server resumes service to the whole group.

mount

Within a cluster a common name space is maintained to the degree possible. Several factors prevent this name space from being 100% common. For example, various vendors are not likely to have a shared root file system. Some nodes may require private mounted file systems (such as /tmp). A mount context is defined to identify which name space is used by a process. If a process moves to another node, its mount context can be used to ensure that the file names are interpreted as they would be on the original execution node. This ensures that the program will produce the same results regardless of the node is happens to run on.

sphere

Several standard UNIX utilities provides information about a node or a set of nodes. For example, the *who* command lists the users on a particular node and the *mount* command lists the the file systems mounted. In a distributed environment such as that provided by **FUSION** these programs may need to display information about several of the nodes. However with a large configuration it is not practical to display information for all of the nodes. The concept of sphere of interest is introduced to limit the list of nodes to be considered for these types of operations. Each user has a sphere of interest which can be customized as required for that user.

2. FUSION: DESCRIPTION OF THE PRODUCT

As described in chapter 1, the **FUSION** Product consists of three layers, the Distributed Execution Environment, the Cluster Environment, and the File System Replication Service, This chapter provides a greater depth of description of the individual functions that make up each layer.

2.1 THE DISTRIBUTED EXECUTION ENVIRONMENT

The Distributed Execution Environment, or DEE, is a set of enhancements to OSF's DCE. It consists of a set of File System Enhancements to the distributed file system capabilities of DCE, support for remote processing operations and a Node Status Service which provides dynamic information about resources in the distributed environment. The remote processing operations make use of some of the file system enhancements provided by the DEE. However, the services of the Clustering Environment are not required to support basic remote processing.

2.1.1 FILE SYSTEM ENHANCEMENTS IN THE DEE

There are several enhancements within the **FUSION** product that fall into the category of File System Enhancements. Each of the enhancements described here are provided for one of two principal reasons (or perhaps both reasons). They are useful in their own right in a distributed computing environment, and they are the useful as the basis for a remote processing implementation. These are the following:

1. Access to remote devices
2. Access to remote pipes
3. Access to remote sockets
4. Select on remote devices, pipes, and sockets
5. File offset coherency
6. File reopen and re-lock capability

Following these items, possible future extensions are described.

2.1.1.1 ACCESS TO REMOTE DEVICES

FUSION provides support for processes to access devices that are attached to other nodes. This support is called the remote device function. There is an underlying assumption that the program can name the device, which would be possible, for example, if the device were accessible via the DCE global name space.

One reason this support is provided is because other distributed system environments have shown that operations such as using tar to read from or write to a tape drive are best done on the node storing the data rather than on the node with the tape drive. Remote devices are also required for support of

remote process execution, since a remote process could inherit an open device, such as a user's workstation. In this case it is not necessary to be able to name the device on the remote node, though it is desirable to be able to do so. The **FUSION** remote device support allows processes to open remote devices, read from and write to remote devices, and also to perform device specific ioctl operations on the device.

2.1.1.2 ACCESS TO REMOTE PIPES

FUSION also provides access to remote pipes. This is desirable because named pipes (often referred to as FIFOs) could exist in a globally visible place, and hence be opened on more than one node. A remote pipe is one in which the data is stored on a node different from where the reading process or the writing process is executing. Although the program does not see a data storage function, the data is temporarily stored between the time it is written to the pipe by the writer and when it is read from the pipe by the reader.

FUSION provides mechanisms to support the opening of, writing to, and reading from pipes that are stored at another node. This support allows the reader and writer to be on different nodes. In addition to the named pipe case, the remote unnamed pipes are supported by **FUSION** for when remote processes inherit open file descriptors for unnamed pipes.

2.1.1.3 ACCESS TO REMOTE SOCKETS

FUSION provides support for remote sockets in both the Internet Domain and UNIX Domain. This is used to support remote process execution, since a process running on the node where the socket was created may move to another node with the socket still open. The support for UNIX Domain sockets will also handle the case where two unrelated processes on different nodes bind to the same socket in the global name space.

2.1.1.4 SELECT ON REMOTE SPECIAL FILES

Relating to the support for remote devices, remote pipes, and remote sockets is support for the remote select function. The basic UNIX select function allows a program to determine if a particular file can be written to or read from. In the case of a remote device, pipe, or socket, this function must be coordinated between the node where the program is running and the node where the device is attached or where the pipe or socket is controlled. Since the methods for doing this are the same whether the special file is a device, pipe, or socket, this function has been separated out and provided using a single mechanism.

2.1.1.5 FILE OFFSET COHERENCY

In standard UNIX implementations, file offsets and certain file open flags are shared between parent and child processes. **FUSION** provides a mechanism to preserve this sharing when the parent and child processes are not executing on the same node. This mechanism insures that the file offsets and related information are consistent in all file blocks that represent the same open of a particular file. This mechanism is called fileblock tokens, and is required to

properly support remote processing operations.

2.1.1.6 FILE REOPEN AND RE-LOCK CAPABILITY

FUSION provides a mechanism to permit a file that is opened on one node to be reopened on another node, while preserving all of the characteristics such as the current file offset from the first node. This is useful for all forms of remote processing, since remote process creation operations and process movement operations both expect to be able to have the files that were open still open. For the cases of process movement, locks that were held by the process need to be preserved as well. **FUSION** also provides a mechanism for preserving held locks across node boundaries. These functions together are referred to as file reopen.

2.1.1.7 OTHER DISTRIBUTED FILE SYSTEM EXTENSIONS

The sockets facility as described above is restricted in that the node where the underlying service is bound remains involved even if all processes using the socket are operating on other nodes. No straightforward mechanism for moving the controlling node that is applicable to all domains has been identified. However, future work may determine solutions that are acceptable to a large class of problems. One possible extension to the file system enhancements in the DEE would be some form of socket endpoint relocation, to make remote socket support more robust.

The streams facility is available in some Unix implementations and full remote process support requires that processes with open streams can move. Providing this function is feasible within the **FUSION** architecture, but is not included in this specification.

The set of IPC mechanisms referred to as System V IPC consist of semaphores, message queues, and shared memory. Full remote process support needs distributed versions of these mechanism and prototype work suggests that implementation is quite feasible within the **FUSION** architecture. It might be particularly desirable for use with clusters built in a single box. However, full specification and design have been deferred.

The TCF system introduced a concept called hidden directories. These are special directories, which for most ordinary accesses are treated as ordinary files. In these cases one of the files within the directory is selected. These were used to store binary executable files that had to be different for different machine types under a common name. Hidden directories were deemed non-essential in the initial release of FUSION. At some future point such support might be desirable.

2.1.2 REMOTE PROCESSING SUPPORT IN THE DEE

The **FUSION** system provides support for basic remote processing at the DEE layer. Within the technical community, the term remote processing can encompass several different facilities, including RPC, remote command

execution, remote process creation, and process migration, as well as certain types of access to processes executing remotely. In the context of **FUSION**, RPC mechanisms from OSF's DCE (known as NCS) is an existing tool upon which **FUSION** remote processing services are built. A function related to the remote processing support in **FUSION** is the Node Status Service, which collects information about the available nodes that can be used to select potential execution sites. It is described in section 2.1.3.

At the abstract level there are two types of remote processing operations within **FUSION**. These are remote process creation and process movement. However, there are more than one function in **FUSION** to provide them. The following remote processing functions are provided by **FUSION** at the DEE layer:

rexec	remote versions of the standard UNIX exec family of system calls. The caller explicitly identifies the node on which the load module is executed. This is a process movement operation.
rfork	a remote version of the standard UNIX fork system call. The caller explicitly identifies the node on which the child process will run. This is a remote process creation operation.
migrate	a new system call that causes the process to change the node on which it is running. This is a process movement operation.
SIGMIGRATE	a new UNIX signal that when received causes the process to change the node on which it is executing. This is a process movement operation.

In addition to the remote process creation and process movement operations, **FUSION** also provides transparent access to remote processes. Remote process access require a process naming strategy which allows processes to retain their process ID regardless of which node they are running on, and to guarantee that a particular process ID is not used by more than one process at a time. **FUSION** provides a mechanism to ensure process IDs are unique within a DCE cell. The **FUSION** architecture can also support remote process access beyond the cell, but the full specification and design of this capability has been deferred.

The architecture for supporting remote processes is built using two pieces of function. The first involves a small restructuring of the base kernel process access code to add the concept of virtual processes, or vprocs. Vprocs are an abstraction of standard UNIX processes to recognize that the aspects of naming processes and identifying the relationships between processes can logically be placed at a higher level than the physical implementation of the operations performed on processes. The implementation of vprocs requires rearranging

some of the process related code in the kernel, so the code that implements the operations on physical processes is separated into service routines (called virtual process operations), and the code that performs the operations on the process calls the virtual process operations via a procedure switch. Within this architecture, it is possible to implement remote processes as an installable client and server. The client code is an alternate set of virtual process operations that ships operations on remote processes to the node where the process is currently executing. Each server function on the node where the process is executing calls the corresponding standard virtual process operation on that node.

FUSION provides user level commands to access the remote processing functions. The on command permits the user to run a command on a specified node. The migrate command allows the user to migrate a process to another node. For convenience, these commands are provided as extensions to the standard UNIX shells. **FUSION** also provides a command called fast that will select a relatively less loaded node to run a command.

The functional area related to remote process creation is that of execution permission, or xperm. This feature is used to restrict where a user can execute, either as a signed on user or via the remote processing capabilities of rexec, rfork, and either form of process migration.

2.1.2.1 REMOTE PROCESSING EXTENSIONS

As process threads become more commonplace the **FUSION** system must accommodate them. The most obvious area is with process migration. When a multi-threaded process migrates to another node, the migration mechanisms need to ensure that all threads migrate at the same time. This includes migrating kernel managed mutex locks.

The migration mechanism could possibly be used as a mechanism for building process checkpoint and restart. This is a possible extension to the remote processing facilities of the DEE.

2.1.3 THE NODE STATUS SERVICE

A major supporting service for remote processing is the Node Status Service (or NSS), which collects and disseminates static and dynamic node status information about nodes in a cell. Each node can register both static information about itself (e.g., instruction sets supported, machine cpu power, optional equipment available, memory size, and paging space) and dynamic information (system load, I/O rates, paging rates, free memory, free page space and related).

The NSS also maintains information on the users that are signed onto each node. This is useful for some UNIX system utilities such as who, last, talk, write, finger, and mail notification. **FUSION** will include modifications to these programs so that users have the option of seeing one large pool of signed on

users, rather than just that group of users that happen to be signed onto the same node where the program is running.

A **FUSION** feature related to the Node Status Service is the sphere of interest. This function permits a user or a process to limit the list of nodes to be considered in a given operation. This is very important when cells scale to hundreds or thousands of nodes.

2.2 THE CLUSTER ENVIRONMENT

The **FUSION** Cluster Environment is built on top of the DEE. The goal of the Cluster Environment is to make a collection of machines on the network appear as a single system, to the degree possible. This includes a common file name space that is consistent on all machines within a cluster, and a set of tools to automatically run some programs on machines deemed best suited to run the program, and among machines of the same type, to move processes automatically to balance the load.

The Cluster Environment can be viewed as three pieces of function. The first is the data clustering capability itself, which provides a consistent view of file system mounts and a common file name space among all of the machines in the cluster. The second is a heterogeneous cluster load balancing function, which, for appropriate applications, will select the node in the cluster most appropriate for executing the application. The third is a homogeneous cluster load balancing function, which will balance the load among machines of the same type, by moving process among machines. Each of these functions will be discussed individually.

2.2.1 THE FUSION CLUSTERING CAPABILITY

The Clustering Capability of **FUSION** is based upon the cluster mount model. The cluster mount model insures that all files are visible with the same name on all machines in a cluster. This function is called the clustermount function. It ensures that a file system mount performed anywhere within the cluster is visible to all nodes in the cluster, so that path names are resolved to the same file regardless of which node evaluated the name.

The most difficult aspect of providing file name space consistency is with the root file system. One approach to providing name space consistency in the root is to provide a single common root file system. With this approach the highest level of file name space consistency is achieved. However, there are three obvious disadvantages of the single common root. These are:

1. the risk of the root being unavailable
2. the potential performance penalties for accessing root files that are stored on another node
3. the difficulty in getting consensus on the contents of the root among multiple vendors and across different platforms, even from one vendor.

To address the first two disadvantages, **FUSION** supports a replicated root. The access and update semantics of a **FUSION** replicated root are the same as those of a single node and the local performance closely matches that of a single node. Also, the failure semantics are consistent with what users would expect.

The problem of getting consensus on the contents of the root is addressed by having the **FUSION** clustermount function also support multiple root file systems within a cluster, while still striving to preserve the highest degree of transparent naming possible. This function is termed extended clustermount services. Extended clustermount provides that cluster-wide mounts are visible from the same name on all different versions of the root, so that the name consistency is preserved. The potential for inconsistencies using the extended clustermount is increased, and must be managed by the cluster administrator. The extended clustermount services also allows for local only mounts, so that (for example) different machines can have different /tmp file system, even when the root it is mounted on is replicated.

One additional aspect of the **FUSION** Clustering Capability is supporting NFS in a cluster-wide manner. There are two aspects of the NFS function. One is making NFS mounts on one node visible to all nodes within the cluster. This function is provided by the clustermount function. When accessing an NFS file from a node in the cluster, **FUSION** is designed so that the using node will communicate with the NFS file server directly, without going through the node that performed the NFS mount. The other aspect of enhanced NFS support in **FUSION** is cache consistency among nodes of the cluster. NFS, unlike DCE's version of AFS, does not provide cache consistency between clients of the same file. Without providing this service within a cluster, serious errors could occur if inconsistent caches were accessed. Consequently **FUSION** has made changes to the NFS client code to allow all members of a cluster to be cache-consistent with respect to each NFS file. This does not require any NFS protocol changes and is transparent to the NFS server.

2.2.2 FUSION HETEROGENEOUS CLUSTER LOAD BALANCING

FUSION's heterogeneous cluster load balancing function is provided to allow the system load caused by particular programs to be balanced among nodes of a cluster. Because the name space of the cluster is consistent among all of the nodes in the cluster, it does not particularly matter which node executes a program in order to get the correct result. However, some machines may be more appropriate for running particular programs, perhaps because they are less loaded, are more powerful machines, they support some capability that is not generally available, or they execute the instruction set for which the program is compiled. The **FUSION** heterogeneous load balancing function takes advantage of the name space consistency, and can automatically select the execution node, without specific intervention from the user.

This function is provided by a kernel extension, which at the time of an exec will make use of system capability and system load information from the Node Status Service (NSS) of the DEE, and select a node that is best suited to run the program. This is not done for every program, but only for executable files marked as candidates for exec time load leveling and for programs that cannot run on the local node.

FUSION permits a replacement load leveling algorithm to be installed by the administrator. Such a replacement could access all of the NSS data that the supplied load leveling extension had available. **FUSION** also provides a set of library routines to access the NSS data, so that a particular application can make its own evaluation, and select a node explicitly using the rexec function.

2.2.3 FUSION HOMOGENEOUS CLUSTER LOAD BALANCING

The homogeneous system has the added ability to balance load within the cluster using the process migration facilities provided by the DEE. Again this function makes use of the fact that a **FUSION** cluster environment guarantees that the file name space is consistent among all nodes within a cluster, and consequently it does not really matter on which node a process runs. It also utilizes the fact that process migrations induced by a signal are not visible to the running program and no program modifications are required to move the program to another node. By spreading out processes among nodes of a cluster, the load can be balanced. This function makes the cluster appear as a multi- processor system, and consequently is sometimes referred to as network MP.

While this function is called Homogeneous Cluster Load Balancing, it is useful in heterogeneous clusters as well as homogeneous clusters. However, the migration functions used to balance the cluster load will only be attempted between nodes in the cluster that are of the same processor type and running compatible operating systems.

FUSION supports system daemons that monitor the load on the nodes within a cluster, and move certain processes around to balance the load. These daemons can operate on a per user basis or operate for all users. The base **FUSION** system will not provide complex programs to provide dynamic load leveling, as the policy decisions need to be quite elaborate and tuned to particular installations.

Automatic process migration is limited to executable files that are designated as candidates for load leveling. Within this environment, users and system administrators can choose to manually move processes between nodes as well.

2.3 THE FILE SYSTEM REPLICATION SERVICE

2.3.1 READ/WRITE FILE SYSTEM REPLICATION

FUSION provides replication capabilities that complement the functions and capabilities of the underlying distributed file system of DCE. The replication

is called single system semantics read/write replication because it provides the single- node/single-copy semantics of a standard UNIX non- replicated file system. The replication capabilities of base DCE are limited in several ways. The multiple copies are not able to be used in a true read/write manner, the copies are not guaranteed to always be up to date, and the mechanisms to bring copies up to date require cloning the entire volume.

FUSION replication is built upon the existing distributed file system capabilities provided by DCE. The replication support automatically manages the consistency of all replicas. To provide this function, update capabilities exist with only one of the copies. By providing this restriction, updates can be synchronized and conflicting updates are avoided in a network partitioned environment. The limitation of having a single update copy, however, requires the users to only have read/only copies of the data available if a failure occurs at the node storing the single update copy or if that node becomes partitioned from the user's node.

2.3.2 REPLICATION EXTENSIONS

In addition to the basic replication services specified in this document, there is additional functionality that may be added to the replication services in the future. These possible extensions are described here.

Replication within TCF provided an open-time option called O_REPLSYNC. When used in conjunction with O_SYNC, this mode guaranteed that file propagation to the replicas occurred as soon as the write operation completed. **FUSION** Replication does not currently support this mode. It might be appropriate to add this in the future.

As laptop machines become more commonplace, specialized support within the **FUSION** environment may be necessary. With laptop machines, the likelihood of shutdown and relocation is much greater. Consequently the replication mechanisms might need to be adapted to the disconnect/reconnect/update scenarios that would be more likely.

One of the limitations of the replication services specified here is the support of a single read-write copy. With this strategy if the site storing this read-write copy fails then the file system become read-only. In the case of network partitioning one or more of the partitions have only read-only copies. A possible extension to the **FUSION** replication service is to support multiple read-write copies of a replicated file system, and provide mechanisms for reconciliation of the multiple copies after they have been disconnected.

3. DESCRIPTION OF FUSION FEATURES

This section provides a detailed description of every major subcomponent of the product. These descriptions are divided into two categories, covering the three fundamental layers of the **FUSION** product as described in the previous chapter. These three layers are the Extensions to the services of the DCE, referred to as the Distributed Execution Environment, or DEE, the **FUSION** Cluster Environment, and the File System Replication Service. The features of the DEE will be provided in the first section. The second section presents the **FUSION** Cluster Environment. The last section describes the File System Replication Service.

3.1 THE DISTRIBUTED EXECUTION ENVIRONMENT

There are three major components of the Distributed Execution Environment, or DEE. Section 3.1.1 discusses the various enhancements to the remote file system provided with OSF's Distributed Computing Environment (DCE). Section 3.1.2 describes support for remote processing within the cell. Section 3.1.3 describes the Node Status Server.

3.1.1 FILE SYSTEM ENHANCEMENTS OF THE DEE

Several functional areas fall into the category of file system enhancements. Included are:

a. Remote device support, including character, raw and block devices along with special devices like /dev/tty and /dev/null;

b. named and unnamed pipes (FIFOs) with readers and writers on potentially different nodes;

c. remote sockets, both internet and unix domain; rexec and migration with open sockets is the principle function;

d. select on remote objects like pipes, devices and sockets;

e. Open file offset coherency;

f. Open file and lock movement, given remote execution;

3.1.1.1 REMOTE DEVICES

With DCE, regular files can be shared and distributed transparently across a network. **FUSION** extends this level of transparency to the sharing of most special files (terminals, disks, tapes, pipes, and sockets). This section describes the component of **FUSION** which makes devices fully transparent to the user. FIFOs and Remote socket endpoint access are discussed in Sections 3.1.1.2 and 3.1.1.3., and the select operation on all remote special files is discussed in Section 3.1.1.4.

3.1.1.1.1 MOTIVATIONS FOR REMOTE SPECIAL FILES

Scenarios in which remote devices are necessary include:

A process is started up on a remote machine, but is connected for input and output to a tty or console on the originating node.

A remote tape drive is read to extract a dump.

A remote process accesses its "/dev/tty" on another node.

Remote device support is key in order to enable and fully support process transparency.

3.1.1.1.2 REMOTE CHARACTER DEVICES

FUSION will support full functional access to remote character devices, although devices that use a non- standard programming interface, such as memory mapped i/o, will not be fully supported remotely. Processes that are operating with a memory mapped device will not be permitted to migrate away from the node controlling the device. Processes that are operating remotely from such a device will not be able to open it if it is always memory mapped, and will not be able to switch it into that mode if it is controlled by an ioctl.

Perhaps the most common use for remote devices is to support remote terminals. **FUSION** processes will be able to inherit terminals in a completely transparent fashion from anywhere in the cell. Programs will be able to issue terminal ioctls without concern for the location of the terminal. Operations issued against a remote terminal will result in the appropriate file system code being called through the VFS+ interface to perform the function. The server code at the device's node will then perform the requested function using the device vnode/gnode's VFS+ interface, returning the correct information to the client code which in turn passes it to the initiating process.

Other character devices, like raw tape devices, are also supported. The underlying mechanism handles the input and output buffering.

3.1.1.1.3 REMOTE BLOCK DEVICES

Full remote access to block devices will be provided. Users are able to perform all standard functions on block devices independent of whether that device is connected to another machine in the cell or cluster. Block devices look much like files when accessed. The primary differences are in the area of mapping offsets to blocks of the device and in the ability to issue ioctl operations against devices. Like remote terminals, remote block devices can be supported by the client machine talking to the server machine to perform the function.

The VFS+ interface is used to hide the mechanics of connecting the client to the proper server. VFS+ interface code is responsible for intercepting operations on remote special files and rerouting them to the remote node using the RPC mechanism.

3.1.1.1.4 REMOTE /DEV/TTY

In order to support remote access to a process's controlling tty, **FUSION** augments the controlling terminal information currently kept in the user structure (u.u_ttyp, u.u_ttyd, u.u_ttympx) with u.u_ttyrdevnode (node number where the actual controlling tty device actually exists). The driver for the /dev/tty pseudo-device, and other code which depends on u.u_tty* is modified or replaced with code which communicates with the node where the controlling- tty exists, if it is not the local node.

In order to support the modification of a process's controlling tty, the various controlling tty fields in the user structure will be passed and returned during RPCs which can effect them. The RPC servers for these operations will copy the fields into their user structure for the duration of the operation. If modified, they will be copied back into the calling process's user structure, thus preserving the single- system semantics of these operations.

3.1.1.1.5 REMOTE /DEV/NULL

As one might expect, writing or reading from /dev/null leads to rather predictable results independent of the /dev/null used. Consequently, code will be provided to map remote /dev/null requests to the local /dev/null where possible. This will be done on the basis of the major and minor numbers of the device, and not on the name of the device.

3.1.1.1.6 IOCTLS

UNIX I/O controls (IOCTLs) are a general mechanism provided to allow a wide range of operations which don't fall into the more common classes of reading, writing, etc. Due to their general nature, they differ widely in their functional interfaces. Some take simple values as arguments, others take pointers to complex trees of data structures.

FUSION will support all host IOCTLs by providing a vnode ioctl operation which takes advantage of NCS RPC's NIDL (Network Interface Design Language) to completely specify the ioctl interface and the data structures it operates on (struct sgttyb, etc.). This will allow each ioctl performed on a remote object to be implemented largely via function shipping. A list of the ioctls supported will be found in Section 3.1.1.1.

A mechanism will be provided by **FUSION** to add remote support for additional ioctls that may be required by new device drivers. This will be implemented by defining a secondary ioctl NIDL declaration and RPC client set that are initially empty. The base **FUSION** ioctl support will search this secondary set if no match is found in principal ioctl support. System administrators can augment this secondary client as required to support new ioctls and link them in with the **FUSION** support.

3.1.1.1.7 IOCTLS BETWEEN INCOMPATIBLE SYSTEMS

When **FUSION** support extends across product lines which are not completely compatible with respect to IOCTLs, it must be possible to link in new RPC clients and servers for the new IOCTL commands, using the mechanisms described in the previous section. Where possible, the server code would translate incompatible ioctl commands into their local equivalent, but there will probably be ioctls which can't be supported on certain foreign devices.

3.1.1.1.8 NAMING AND OPENING REMOTE DEVICES

Cellwide, devices for particular nodes may or may not be nameable, depending on what is placed in the global file system naming tree. Initial opening of devices named via on-disk junctions will be based on the DCE approach to diskless machines. That approach is expected to be based on parameters associated with the mount or a new system call to identify devices in a volume as belonging to a station.

Once remote devices are opened, they may be inherited across remote process operations are accessible via the Reopen code described in Section 3.1.1.6.

Within a cluster, all files are nameable; therefore all devices are nameable. In the absence of other device node specification, device inodes will be assumed to apply to the node which did the /etc/mount of the file system containing the device inode. Device inodes in replicated File Systems will be assumed to apply to the node which has the writable copy of the file system. If the process becomes partitioned from the device, subsequent I/Os to the device will fail.

3.1.1.1.9 SIGNALS FROM DEVICES

Devices can generate signals, either in response to external events or due to explicit programming. These signals are always be seen by whichever processes would have seen the same signals in a single-machine environment, assuming that the machines on which those processes are executing remain available.

3.1.1.2 REMOTE PIPES

Throughout this document, both named and unnamed pipes will be referred to as pipes unless an explicit distinction is made between the two.

The implementation of remote pipes will provide single-system UNIX semantics for pipes used in the distributed environment. This functionality is necessary to allow users to move a process throughout the cell or cluster without concern for where the input or output of the process is directed.

Pipes are supported when the reader and writer are not local to the machine supporting the pipe. In addition, multiple readers or writers may exist and they may all be on different nodes.

Pipelines will be supported in **FUSION** without sacrificing single-system Unix semantics. Each such pipeline may be composed of an arbitrarily long series of processes connected by pipes. Each of the processes may be running on any of the active nodes.

The current design of **FUSION** pipes does not try to recover from a broken pipe condition due to the loss of a controlling node. If the node where a pipe is being controlled fails during a pipe operation, the system generates a broken pipe error message, generates the SIGPIPE signal, and terminates the pipe operation. Furthermore, if the node where a named pipe inode is stored (or the read/write node if that named pipe is stored in a replicated file system) goes down during while the pipe is open, the pipe is broken in the same way.

In **FUSION**, pipes are implemented as a virtual file system. The design uses a client/server paradigm, but the client and server are really parts of the same virtual file system code. The **FUSION** remote pipe virtual file system code is designed to utilize the local implementation of pipes on the controlling node of the pipe.

PIPE STORAGE

The **FUSION** remote pipe support utilizes the underlying pipe implementations to store pipe data. This leaves vendors free to chose the storage method most appropriate for their system. Vendors often have strong reasons for choosing a particular storage method. Conventional systems use a file system and the buffer cache to store data. A real-time system might require all pipe data to be kept pinned in memory so that no disk activity will occur during time-sensitive operations. Another vendor that supports a paging kernel memory might wish to simply call a kernel malloc() routine to allocate space for pipe data.

From the **FUSION** perspective, the underlying pipe storage module presents a simple first in/first out byte stream abstraction. The vendor supplied storage module needs to provide routines to allocate pipes, deallocate pipes, read from pipes, write to pipes, and to test for blocking conditions (such as an empty pipe on read and a full pipe on write). The storage module requires no knowledge of any aspect of the distributed environment; it simply acts as the local repository for pipe data.

FUSION REMOTE PIPE POLICY MODULE

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

The policy module maintains a pool of FIFO descriptors. A FIFO descriptor contains all the state information that the policy module needs for managing a

remote pipe. Information in the descriptor indicates whether it represents a locally or remotely stored pipe.

In the case of an unnamed pipe, the node that controls the pipe is the node where the pipe was created with the pipe() system call. If the processes using the pipe change execution nodes, then the controlling node of the pipe is moved when the last processes using that pipe moves to another node or exits. When that occurs another node is selected from the set of using nodes. This scheme allows the set of cooperating processes to continue even if the node where the pipe was originally created fails.

In the case of a named pipe the official control point of a pipe is the node that stores the file system where the pipe inode exists (or the node where the read/write copy is stored if the file system is replicated). For efficiency, a state transition protocol is used to allow the node storing the pipe inode to temporarily lend a pipe to another node when all using processes reside on the second node. If a process from a third node opens the pipe, the controlling node can reclaim responsibility for managing the pipe without any interruption of service.

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, that is without being interleaved with data from other WRITE requests. The value of PIPE_BUF can vary from node to node in a heterogeneous environment. To comply with the standard, the policy module must choose the largest PIPE_BUF value from among all using nodes that have a pipe open for writing. Thus the client node's operative value of PIPE_BUF may be different from descriptor to descriptor on the same server node.

If a using node's value of PIPE_BUF is greater than the amount of data that can be stored in a pipe by the server node's implementation (i.e client PIPE_BUF > server PIPE_MAX), the pipe will be lent to the node with the largest PIPE_BUF value. If the client PIPE_BUF is larger than the server PIPE_BUF but not larger than the server PIPE_MAX, the server will implement the larger PIPE_BUF value in the policy module.

PIPE SYNCHRONIZATION

In a nonpreemptable uniprocessor UNIX kernel, pipe 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, the process sleeps and is later reawakened by the occurrence of some event, (e.g. another process riting data to an empty pipe) and it can then proceed. It is assumed to be acceptable for kernel threads servicing the pipe RPCs to sleep at the server end just as local pipe operations do.

3.1.1.3 REMOTE SOCKETS

The Berkeley socket abstraction defines an interface for interprocess communication with a socket as an end- point of communication. There are two socket domains commonly provided. The UNIX domain of 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 use sockets controlled on a remote nodes transparently. 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 protocols.

Remote socket support will be provided for both the datagram (SOCK_DGRAM) and connection oriented services (SOCK_STREAM) for the local unix (AF_UNIX) and DARPA internet (AF_INET) protocol families. Since raw sockets (SOCK_RAW) are only used to directly access the underlying communication protocols on the local node, they cannot be accessed remotely.

It should be noted that only processes with "Well Behaved" sockets will be allowed to fork, exec, or migrate to another node. A well behaved socket is a socket in a known state where system calls upon it will result in known results regardless of the underlying protocol. A precise definition of what it takes for a socket to be well behaved is provided in a few paragraphs. If this restriction was not enforced it would be possible for applications to create one or more sockets, migrate and not understand why the communications failed.

The remote socket layer will be implemented with a client/server model using NCS RPC calls. When a remote socket is required, a server process is created on the machine where the socket was created. It then transfers the node-independent fields of the socket (socket type, socket options, and socket state) to the new node. The new node allocates a socket structure for each remote socket, and attaches a file structure with a new set of fileops to the newly created sockets. A flag is set in the socket structure indicating the endpoint is remote.

After the remote socket is established, the server keeps track of all nodes that have the socket open. When the process running on the remote node accesses the socket, the socket system calls detect that the socket is remote and ship the socket requests to the original host. The remote socket server in the original host identifies the individual sockets by a bound handle and issues the requested socket operation on the individual socket.

If a process with an open remote socket moves to or creates a process on another node, the new remote socket is linked back to the original node. This minimizes the multiple hops required to support remote sockets. If however this node is the original node, the process will access the socket directly.

WELL BEHAVED SOCKETS

Using node-dependent information in creating sockets can cause migration of partially connected sockets to fail. This would most commonly occur in AF_UNIX domain sockets where the connection name is derived from some node-specific information. If the node endpoint of a socket were changed before this information is obtained from the original node, the subsequent connection would fail. To minimize the effects of this problem, the following restriction will be applied; AF_UNIX domain sockets cannot become remote until either the connect() or bind() socket call is performed.

Note that this only protects sockets created before the migration. Processes which acquire node dependent information such as the nodes network address after becoming remote may still produce incorrect results.

REMOTE SOCKET SYSTEM CALLS

The remote socket client intercepts all socket system calls and forwards them to the remote socket server at the original host.

The socket system calls that send data (send(), sendmsg(), sendto()) are intercepted at the sosend() routine.

The socket system calls that receive data (recv(), recvmsg(), recvfrom()) are intercepted at the soreceive() routine.

The file operations ioctl(), close(), and stat() are intercepted and forwarded to the original host by the substituted file operations. The read() and write() file operation will be intercepted when they call soreceive() or sosend(), respectively. The fcntl() operations are not intercepted by the socket layer. Instead, the flags field from the file descriptor is included in all RPCs forwarded to the original host.

The select() file operation is handled by the generalized support for select on remote special files, which is described in Section 3.1.1.4.

3.1.1.4 SELECT ON REMOTE SPECIAL FILES

The select system call allows a process to wait for data or other events on multiple file-descriptors simultaneously. Remote select allows a process to wait for data or other events on file descriptors which include files which correspond to remote objects like devices, pipes, and sockets.

INTERFACES OF REMOTE SELECT

Like all open files, a file which refers to a selectable remote object points to a set of functions through the file operations switch. These are the fileops for objects of the given type. (There are several different sets of fileops in the system, but a given open file only refers to one of them at a given time).

Among the operations for each file type is a select (or poll) operation, generically known as the type's fo_select routine. For files which refer remote objects for which the select operation is legal, the select operation will be rem_select().

The rem_select function will provide a standard select- file-op interface:

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

This means that the main select code in the kernel is unchanged, since it only requires that the underlying file object implement the fo_select functionality. The location of the actual resource is hidden from this layer.

MODULES THAT COMPRISE REMOTE SELECT

The remote select functionality will be implemented as two main remote procedure calls (RPC's). The RPCs are remote_selpoll, and remote_selnotify. These two RPCs are in some sense symmetric. Remote_selpoll is a call made from the process's node to the object's (FIFO, SOCKET, DEV) node. Remote_selnotify is the opposite, coming from the object's node to the process's node. In addition, each RPC can be thought of as having two sides: a client side (work done at the caller), and a server side (work done at the called side). remote_selpoll

Rem_select will call remote_selpoll (RPC) to do its work. The server code for the remote_selpoll RPC (at the object's node) will simulate a file struct for the specified object, and then call selpoll() (the guts of select) to setup the necessary select event blocks. It will also pass a pointer to a special notification routine. That notification routine will know how to notify the remote selecting process via remote_sel_notify(). The RPC will then pass back the return value and modified reqevents and rtnevents.

remote_selnotify

As mentioned above, the remote_selnotify routine will be used to notify remote processes selecting on an object when an awaited event happens to the object (data becomes available to read, ...). The remote_selnotify RPC will be invoked by the remote notification routine passed into selpoll on the object node. The remote_selnotify server code (executed at the node where the select

call is waiting) will then call the standard AIX selnotify routine to notify the selecting process of remote events.

CLEANUP OF REMOTE SELECT

Due to the large potential overhead (due to multiple RPC's), and to the non-procedural interface, select cleanup will be left up to timeouts or subsequent selects. Hence if a process is selecting on several devices, and one device has data ready, the process will awake and his select will return. At this point, no attempt will be made to cleanup the data structures associated with the previous selects. Timeouts (if specified) will be used to cause these selects to eventually be cleaned up. On last close of the file object (due to normal closes, or to "node-down") any associated select resources will be released.

In the case where a remote select has terminated, but the select data structures at the selected object node have not been cleaned up, it is possible that spurious notification RPC's may be made. These will be ignored at the selecting node due to the uniqid in the pending- select data structures.

OTHER SELECT IMPLEMENTATIONS

The remote select RPC protocol is independent of the underlying select implementations, provided that the underlying implementation provides and uses the select fileop. A select implementation like OSF/1 may require a little more code at the remote select's server (the node where the selected object is stored or exists) than AIX 3.1. This is primarily due to the inferior performance of OSF's select code. Once this is done, however, the various implementations of select will be completely interoperable.

3.1.1.5 FILE OFFSET COHERENCY

One of the requirements for true process location transparency is that processes on multiple machines be able to interact with the same semantics as if they were running on a single node. UNIX processes on a single node can share open files (inherited across forks). The semantics of this sharing requires that the file offset and file flags (NDELAY, etc.) be shared. Thus if one of the processes reads a byte from the file, the file offset will increase by one, and the next process that reads from the file will see the next byte. This must be maintained even if the processes are on different physical machines.

Fileblock tokens are the mechanism currently used by AIX/TCF to synchronize access to a shared fileblock. **FUSION** will employ a similar mechanism. The fileblock is the kernel data structure which contains the file offset and flags.

Support for the sharing of file offset and flags between nodes requires relatively few "hooks" in the upper levels of the operating system which supports **FUSION**.

These hooks are calls to acquire and release the fileblock token for any file being operated on by a system call which modifies or interrogates that file's

offset or flags. Examples of such calls are read, write, fcntl and lseek. The exact places for these hooks will vary from OS to OS, but the calls should be the same:

```
fbtoken_lock(struct file *fp);
```

operation which uses or modifies the file's offset or flags.

```
fbtoken_unlock(struct file *fp);
```

The implementation of fileblock tokens will use Remote Procedure Calls, as provided by NCS.

Among the primary functional requirements in the design is a large measure of Token Managing Node independence. To provide a high level of robustness, the underlying fileblock token protocol will provide an automatic mechanism to move fileblock token management to one of the nodes using the fileblock.

The fileblock tokens implementation will provide a means to allow underlying protocols (like remote pipes, sockets, terminals, and other devices) to take advantage of the synchronization provided by the protocol. This will be implemented as a new operation in the file_ops structure, the fo_fbtok_revoke operation.

```
void
fo_fbtok_revoke(
struct file *fp,
token_seq_t tokseq
);
```

File-types which provide the fo_fbtok_revoke operation will use it to guarantee that all data that they've written to the underlying object (socket, pipe, ...) has been flushed to the object's storage node. They might also return unconsumed (unread) data at this point.

The token-sequence number in fo_fbtok_revoke is provided to allow the underlying remote-object access protocol to implement flushing of data in a non- synchronous manner to minimize the performance impact of supporting fileblock tokens. A simple protocol could just provide a version of fo_fbtok_revoke which doesn't return until all data is guaranteed to have been flushed back to the storage node's real object. A more complex protocol could simply begin the flush procedure when its fo_fbtok_revoke is called, and use the token-sequence number at the storage node to guarantee that no other data is allowed into the object until that token sequence has been completed.

In addition, the file structure will be extended with a small data structure to maintain the state of its fileblock token, if any. The exact format of this data structure will be specified in the Detailed Design Specification for Fileblock Tokens.

3.1.1.6 FILE REOPEN AND LOCK INHERITANCE

Transparent remote processing while preserving UNIX and POSIX semantics requires the following:

1. The inheritance of open files when processes are created on remote nodes, and,
2. The preservation of open files and locks when processes are moved between nodes.

FUSION implements the movement of open files by a protocol which involves a "prepare" phase, done on the originating node where the files are open, and a "reopen" phase which is performed on the destination node.

INTERNAL INTERFACES TO FILE REOPEN

When performing one of the major Process Transparency operations which creates a process on a destination node, the destination node will perform the GET_LOCK_FILE_DATA RPC to the originating node, specifying the source process. Despite the simple name, the client and server for this RPC are responsible for setting up all file related state for the new process. These includes:

— reestablishing open files.

— migrating any file locks.

— setting up "fileblock tokens" for shared file "blocks".

— setting up current working directory, current root, and the mount context (See Section 3.2.1.1.6 for a description of the mount context)

INTERFACES USED:

The file structure in VFS based kernels contains a pointer at a set of operations which can be done on that file. These are normally:

```
fo_rw
fo_ioctl
fo_select
fo_close
fo_stat
```

To allow the GET_LOCK_FILE_DATA server to remotely reopen files, **FUSION** will add the following operations to this set:

```
fo_prepare_for_export
fo_reopen
```

The fo_prepare_for_export has two responsibilities:

1. perform the modifications that are necessary to the local file and its underlying data structures to allow the file to be shared with a remote node.

2. provide a reopen handle which can be used at the destination node to reopen the file.

The fo_reopen operation for a particular type of file does the reverse; it takes a prototype fileblock and reopen handle, and performs the actions appropriate to the file type to make the fileblock refer to the same object as the original open file.

As mentioned above, each file type must provide functions to implement fo_prepare_for_export() and fo_reopen() for the particular type.

FUSION provides these file-operations for both of the two types of files which are visible to users, vnode- based files and non-vnode-based files.

For vnode-based files (e.g.. NFS, AFS, and local physical file system files), the fileops require support from each virtual file system type. This support consists of extensions to the standard set of virtual-file-system operations. These extensions are:

ORIGIN NODE:

VN_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).

VN_PREP_EXPORT_LOCKS

Creates a data structure which describes any locks that the process held on the file, and, again, does whatever token magic is necessary to allow exclusive locks to be reacquired remotely (since the local locks won't have been removed by then). This is only called for migrate and all remote execs, since file locks are per process.

DESTINATION NODE:

VN_REOPEN

Uses the reopen handle produced at the origin node to get a vnode, and opens it.

VN_RELOCK

Reacquires any locks which the process held on the file.

For non-vnode based files (e.g., some flavors of sockets) which are not normally accessed remotely, the file-op extension routines are responsible for invoking the new mechanisms which **FUSION** provides for remote access to these objects.

3.1.2 REMOTE PROCESSING ENHANCEMENTS OF THE DEE

The remote processing functionality is described via seven subsections: remote process programming primitives, cellwide process name space, virtual processes, remote process client/server function, execution permission, remote processing commands and shell enhancements.

3.1.2.1 REMOTE PROCESS PROGRAMMING PRIMITIVES

The programming level interface to the remote processing functions of **FUSION** provides the programmer a wide range of flexibility to control processes. New system calls are provided which allow programmers to explicitly indicate remote nodes or otherwise take advantage of the distributed system.

FUSION provides a new form of the fork system call called rfork. Rfork takes arguments to specify the environment in which the child process should be created. **FUSION** also provides new versions of all forms of the exec system call (execl, execv, execle, execve, execlp, and execvp) which together are called the rexec system calls. These new calls are rexecl, rexecv, rexecle, rexecve, rexeclp, and rexecvp. They take the same arguments as the equivalent exec call, but in addition take arguments to indicate which environment the process should execute.

In addition to the capabilities of rfork and rexec, it is also desirable to be able to replace the current process with an exact copy on a new node. This is provided with the migration functionality. The migrate system call takes the same arguments are rfork, indicating the new environment in which the process should run. The migration of a process can also be initiated through a new signal called SIGMIGRATE. This signal, when delivered, has a default behavior of migrating the process. Programs may catch this signal and implement any desired behavior. If either form of the migrate operations fail the process continues to execute on the original machine.

Migration of processes with multiple-threads will migrate all of the threads of the process to the new node. All kernel supported mutex locks will also be migrated in a transparent manner.

3.1.2.2 CELLWIDE PROCESS NAMESPACE

Central to all aspects of Process Transparency is the name space for processes. The name space for processes is the set of process identifiers (pids). The algorithm used to generate pids varies from UNIX system to UNIX system, but UNIX guarantees at least uniqueness for each pid with respect to other processes in the system.

For Process Transparency, this guarantee must be extended into the network environment. In a single machine environment, a process doesn't change its pid during an exec and a user or process can signal or otherwise access any process to which they have permission. Expanding to the distributed environment should not unnecessarily break these semantics and with **FUSION**

there are unique pids cellwide. A wider level of distribution is desirable but difficult to manage, given the restricted nature of pids.

FUSION assumes that all participating nodes can support 31 bit pids. Within this constraint, **FUSION** can support 16,383 (16k-1) participating nodes per cell. Each node will have a minimum of 128k pids to use and control, with the option of many more 128k chunks of pid space. The cost of each range is that one fewer total nodes can be supported. For example, if one had three machines in the cell which each needed over one million process ids (and thus an extra seven ranges each), while the others needed no more than 128k, the total supported number of nodes would drop from 16383 to 16362.

To support this cellwide pid capability, each node, as part of **FUSION** installation, is given a primary pid range. Ranges can be added during system operation. Releases of ranges can be done only while the machine is disabled. PID range allocation is kept cellwide via the directory name service. In the unlikely event that conflicting updates are done to the pid allocation data, the Node Status Service (See Section 3.1.3) will detect the problem and isolate an offender so as to avoid duplicate pid name confusion. To allow processes to be created before the **FUSION** extension is added to the system and a primary pid range is allocated, the pid range of 0-(128k-1) is reserved as a special case: processes in this range are "local only" - they are not allowed to become remote processes. These processes may also no be parents of remote processes, with the special case exception being the init process.

3.1.2.3 VIRTUAL PROCESSES

Virtual Processes are a generalized mechanism to support remote process operations. They represent a simple restructuring of the kernel so that code to access a process is directed through a virtual process switch. This restructuring is possible because the name of the process and the grouping with its relatives are logically at a higher level from how the operations are performed on the processes. For local processes, the routines below the process switch manipulate the physical process directly, as before. Once this restructuring is done, one or more remote process implementations can be installed within the kernel.

Users, processes, and kernel code can access local processes and remote processes transparently. The virtual process concept does for processes what vnodes do for files. Vnodes and vnode operations allow users, processes, and kernel code to access local and remote files transparently, and also allows different remote file implementations to be installed in the kernel at the same time, simultaneously.

The virtual process level represents names and relationships of processes. The latter is necessary because certain system calls reference processes not by their name but by their relationship to some process. For example, a process can find out the process ID of its parent via a getppid system call. Another

example is the signaling of a process group, which is a collection of related processes.

The virtual process design divides the data at the virtual process level into to structures: The implementation independent vproc structure and the implementation dependent pvproc structure. The vproc structure points to the pvproc structure. It also has a pointer to an array of function pointers, which establish the operations which can be performed on the underlying physical process implementation. The vproc structure also has a pointer to the physical process structure, which is valid if the process is executing locally.

The specific fields included in the pvproc structure are defined by the particular implementation. In general, however, these fields are typically those that specify relationships between processes, such as sessions, groups, parents, children, and siblings.

When a process moves from one node to another the physical process entry is destroyed on the source node and new physical process entry is created at the destination node. The array of function pointers on the originating node is also changed from physical process ops to client code functions. However, the vproc remains on the source node, and a new vproc is created on the destination node (if one does not already exist there). The array of function pointers on the originating node is also changed from the local physical process operations to a set of remote client functions.

WHERE VPROCS EXIST

Like vnodes, a vproc for a given physical process may exist on several different nodes simultaneously. There are several reasons a vproc for a process may exist on a give node. Some of these reasons are given below. When more than one of these applies, a use count field in the vproc structure is used to keep track of the number of references to the vproc.

A vproc is needed for a process that is currently executing on this node.

A vproc is needed for 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 for a set of processes.

A vproc is needed for the parent of a process currently executing on this node.

A vproc is needed for a child of a process currently executing on this node.

A vproc is needed for the process group leader of a process currently executing on this node.

A vproc is needed for the session leader of a process currently executing on this node.

A vproc is needed for a process being "temporarily" referred to. A "temporary" referral is a referral in which a process ID is referred to (e.g. in a system call) and the process being referred to 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 referral (i.e. for the duration of the system call).

3.1.2.4 REMOTE PROCESS CLIENT/SERVER FUNCTION

Remote processes are implemented as a client/server model using NCS RPCs. The remote process client takes virtual process operation calls from the local kernel and translates them to RPCs addressed to the proper remote node. The remote process server handles all such RPCs for processes physically residing on its node, or returns information to help the client locate the proper node for the involved process(es).

The client also implements the Process Transparency system calls - migrate(), rexecve(), and rfork() - and handles default SIGMIGRATE behavior. The remote process server, when it receives a virtual process request, uses the same local virtual process operations that the local caller would use for a local process.

Remote process client and server support can therefore be seen primarily as a location, routing and pass- through mechanism to deliver process context and parameters from a client to the current physical location of a process, whereupon existing local virtual process code is used to complete the request.

The Remote Process Client checks several statuses to determine if further action is required. If the Remote Process Server return value indicates that the requested process has moved, a query is sent to the original execution node to obtain the current execution node.

If the original execution node is unavailable, a surrogate origin node is used instead. Surrogate nodes are assigned dynamically based on a preferred order as established by the network administrator. Remote Process Servers are obligated to tell the surrogate origin node of any processes they are hosting as soon as they detect that the origin execution node of that process is no longer operable. Thus, if the query to original node fails, the Remote Process Client contacts the surrogate for the execution node information. Error recovery continues until a bad RPC status is detected or the maximum number of tries of different execution nodes has been reached.

When a vproc operation fails, the Remote Process Client updates the u_error field in the user block with an appropriate error code.

3.1.2.5 EXECUTION PERMISSION

In going from a single machine to a distributed environment, authorization controls are required be added to restrict which users can execute on specific nodes. The authorization data should be managed by the DCE Authorization Services. To minimize the cost of queries to the DCE Authorization Services,

this information will be cached the kernel.

Both positive and negative caches are needed. The positive cache is the actual tickets to do remote operations. A negative cache would record nodes a given user can't execute on. The negative cache is important in relation to the user's sphere of interest (SOI). The cache avoids repeatedly picking a node from the SOI for which execution is not permitted.

Besides the static data kept in the kerberos database, it is also important that users with appropriate privilege can dynamically restrict execution on particular nodes. A flexible mechanism to accomplish this is provided.

3.1.2.6 REMOTE PROCESSING COMMAND INTERFACE

FUSION provides three commands for users to access the remote processing features from the command line. The onnode command allows the user to specify a particular on which to run a command. The migrate command permits the user to migrate a process or a process group to a particular node. The fast command allows the system to pick a site on which to run a program based on system load.

SHELL ENHANCEMENTS FOR REMOTE PROCESSING

There are two primary reasons for enhancing the shells: to provide superior semantics for context manipulation operations like sphere, and, to initiate, monitor and control remote processes within the shell environment.

KORN SHELL

An overview of the extensions to the Korn shell is given below:

The jobs built-in command is altered so that the execution node name is displayed for each job if the "- l" option is used.

The onnode control command is provided. The command is used to initiate a job on a remote machine. The remote machine can either be specified by a node name or a set of node attributes. Input-output redirection is performed prior to doing any file system context changes. Options are provided to display the destination node name.

The migrate built-in command is provided. The command is used to synchronously migrate the shell or to asynchronously migrate a process or process group. The destination node is provided either as a node name, or a set of node attributes. The process or process groups to be migrated are specified via process identifiers. For the migrate to succeed the destination node must be the same node type with a compatible operating system as the current execution node of the process or process group.

Note that a process does not migrate until it starts to run. Stopped processes must be started before they migrate. In most signal implementations, a signal sent to a stopped process will be stored. If the shell is migrated, then it will

change its mount context (as described in Section 3.2.1.1.6) after completing the migrate. In addition, it will rebuild the hash table for the search path.

Note: if the migrate fails the process continues on its original node, retaining its original context. The sphere built-in command is provided. The semantics of this command are equivalent to those defined in Section 3.1.3.3, except that a new shell is not created. The change is performed directly on the shell process.

BOURNE SHELL AND C SHELL Enhancements similar to the above will be made to the Bourne and C Shells.

3.1.3 NODE STATUS SERVICE

The Node Status Service is intended to provide clients with at least the following information regarding nodes wither currently participating or previously participating in the cell:

1. whether they are currently participating and for how low;
2. whether they have ceased to participate and how long ago;
3. the node attributes list, including OS vendor, OS version, instruction sets supported, load average scaling factor, preference level, cluster membership, process id ranges;
4. the load average and other dynamic data, including network cost data with respect to the local machine;
5. which users are currently logged in for each node
6. /etc/mount information if the node is part of a cluster

Commands and the kernel which require information concerning other nodes participating in the cell or cluster will use the Node Status Service. This service will reduce retry delays by knowing which nodes are currently participating. The Node Status Service is extensively used by the process transparency cluster function for automatic node selection. It is also used for presentation of a Single System Image view of logged on users via extended version of standard commands like who, users, talk, write, wall finger and mail notification.

Since cells and clusters may be large, the users of these commands may not be interested in information about all the participating nodes within the cell. For this reason, the key commands normally operate upon information about the nodes within one of the process's or user's Sphere Of Interest.

The Node Status Service is actually six services working together to gather processor, mount and signed on user information from participating nodes and to selectively disseminate this information. The services are:

— Local Node Attribute Service;

— Local Cluster Mount Service;

— Local Node User Service;

— Group Node Attribute Service;

— Group Cluster Mount Service;

— Group Node User Service;

Each participating node runs the local version of the Node Status Service (and thus the three local services listed above). The local services keep information about the local node (which they push to the group versions) and cache information obtained from the group versions about other nodes. The local services never communicate with anyone other than group services or potential group services. Very few nodes run an active Group Node Status Service. Each such group service can support dozens of nodes.

The Group Node Status Service communicates primarily with the Local Node Status Service on a defined set of nodes, and with other Group Node Status Services. It collects the information about each of its clients and answers queries from either those clients or other group services. The queries from local clients can be about other local clients or about nodes supported by other group servers. In the later case, the group service will contact the relevant other group services to obtain the requested information. The definition of which nodes are in which groups and which nodes are to run a version of the group service is available through a DCE globally accessible set of files.

Below are discussions first of the Local Node Status Service function, then the Group Node Status function and finally the sphere of interest capability.

3.1.3.1 LOCAL NODE STATUS SERVICE FUNCTIONS

The Local Node Status Service is a daemon (together with a kernel extension) which is started after the node is booted. Below we look at the activities of this service from the standpoint of startup, ongoing background activity, update requests from local clients, queries from local clients, requests from the Group Node Status Service, and actions to take when the Group Node Status Service becomes unavailable.

STARTUP

When the Local Node Status daemon is initiated, it obtains the local node attributes, the local mount information and information about any currently signed on users. Next it must find its Group Node Status Service. A query to the DCE Cell Directory Service gives an ordered list of nodes (RPC handles) to try to find where the Group Node Status Service is executing. When the GNSS is found, the LNSS registers with it and pushes node attribute and node user information to it. When requested to join a cluster (a separate function

from just joining the group), the LNSS will push mount information to the GNSS.

ONGOING ACTIVITIES

Independent of any query or update activity involving the LNSS, this service will do a keep alive with it's GNSS by sending dynamic load and other resource utilization information with at least a minimal frequency. This information includes the current set of signed on users, the system load, I/O rates, paging rates, amount of free memory, and amount of free swap space. A more rapid frequency is used if significant changes occur. Both frequencies and the definition of "significant resource utilization change" are tunable, subject to group wide upper bounds.

UPDATES

Updates to the Node Attribute Service could be changes either to the static local node attributes or to local loads. Only users with appropriate privilege can change the static node attributes and only the Local Node Status Service or the kernel can update resource utilization data. Updates to the Node User Service would be the result of users logging in or out of the local machine. Updates to the Cluster Mount Service would be the result of local file system mounts or unmounts, including NFS mounts.

QUERIES

Queries to the Node Attribute Service are typically attribute based queries with a scope limited by a Sphere Of Interest (SOI), returning a set of node attribute records. Node attribute records will accommodate expansion and the protocol will have data versions capability.

One example query might be for

All nodes with attribute X in SOI A

while another might be

All nodes with attributes X and Y and Z in SOI A.

Example queries to the Node User Service include:

— All logins of user X in SOI A

— All users on node Y

— All users on all nodes in SOI A

Cluster Mount Service queries include:

— All mounts on site X

— All mounts in SOI A

— The mount which is mounted on mount point K

— A pathname query to get into site X's mount context (see Section 3.2.1.1)

INTERFACE BETWEEN LOCAL AND GROUP SERVICES

There are two interfaces from the Group Service back to the Local Node Status Service. The first is a simple "are you there." This can be sent at any time, but more specifically, will be sent:

— by a new GNSS

— by a GNSS who has reason to believe the LNSS is unavailable

The other interface from the Group Service to the Local Service is to push mount information as a result of mounts or umounts done by other nodes in the local node's cluster.

LOSING THE CONTROLLING GROUP NODE STATUS SERVICE

One can lose one's Group Node Status Service either suddenly (not responding to messages) or via a redirect to use a different GNSS. The redirect would typically be the result of a more preferred GNSS node becoming available.

Changing to a new GNSS simply requires pushing all local node attribute and node user information. Local cluster mount information will be pulled from the new GCMS if necessary (he may get it from another GCMS). The Local Node Status Service cache is not affected by changing to a new Group Node Status Service.

Selecting a new GNSS if the current one becomes unavailable can be done via the algorithm described in the STARTUP section above.

To deal with Group Node Status Service transient availability, each protocol to the GNSS has both a "retry later" and a "try him" response.

3.1.3.2 GROUP NODE STATUS SERVICE

Groups are named in the Directory Service name space and Group Servers locate other Group Servers by using the Directory Service, relating a group to the ordered list of possible group servers. A query concerning a node outside of the local group requires knowing the group of the node in question. Group servers will tend to cache such information but if needed, there is information in the Directory Service which relates nodes to their group.

Note that race conditions concerning either cached data or Directory Service replication are caught with retry return codes on messages to the Group Node Status Service.

UPDATE

The general update interfaces for the Group Node Status Service are:

— I'm servicing group XYZ

— I'm shutting down as a group service

— Node X is servicing XYZ so you shouldn't

The Cluster Mount update interfaces are:

— Adding site X to cluster A

— Deleting site X from cluster A

— Adding/deleted file system Z from site X, cluster A

QUERY

The general query interface for the Group Node Status Service is:

— Are you the group server for group XYZ ?

— The Node Attribute Service Query interface is:

— Ship node attribute records for sites X,Y,Z

— Ship node attribute records for all active sites in your group

— Ship node attribute records for nodes with attributes X in SOI A

The Node User Service Query interface is the same as the one from the LNSS to the GNSS.

The CMS Query interface is:

— Ok to add/delete site X to cluster A ?

— Ok to add/delete file system Y from site X on cluster A ?

— Read complete CMS data for cluster A

DECLARING A NODE UNAVAILABLE

Group Node Status Services are responsible for knowing if each of the nodes in their group is available. It is impractical and inefficient for each participating node to discover the state of all other nodes independently, especially since with **FUSION**, any given node may have a reasonably intimate interest with quite a few other nodes (transparent site selection to those nodes, replication of data with those nodes, etc.).

The Group Node Status Service (GNSS) should not be spurious about declaring a node unavailable but some algorithm is needed. The general strategy is:

a. Wait for the LNSS on the node to do its keep alive. There is a maximum timeout for this message, which can be configured.

b. After this timeout the node in question is marked as indeterminate. Requests with respect to it are delayed, however, it is not selected as a

good candidate for load leveling.

c. If the node fails to send its keep alive in the designated time, send a probe to that LNSS and again wait a specifiable time, doing retries as necessary. The node will respond even if it is no longer in the same group.

d. With no response to this message, the GNSS can optionally ask other Group Servers if they have any objection to declaring that node as unavailable. One reason one may wish to make this request is to discover the possible topology problem where another GNSS for your group is being created and the other GNSS servers for the other groups are aware of it.

e. At this point the node is declared unavailable and the GNSS can optionally tell the other GNSS so they can update their status caches and not assume the node is still available.

Steps d) and e) are optional both because operations will work even if the messages are not sent and because sending these messages may be a scaling problem in very large cells.

Note that if the GNSS is told that a node is suspected of being unavailable it can skip step a) and go directly to sending the probe in step c). Nodes rejoin the group by sending status to their GNSS. Once a node is declared unavailable, it is excluded from whatever cluster it was in before by telling the Group Cluster Mount Service (GCMS). That service will initiate cleanup activities with respect to the node which became unavailable.

GROUP SERVICE SCALING

The objective of the overall Node Status Service design is to handle several thousand nodes. To do this, we expect each Group Server to handle up to 100 or more nodes but probably not up to 500 nodes. The number of groups which could be supported is, in general, bounded only if we insist on having agreement between all the Group Servers as to who the active Group Server for each group is. Such a requirement helps to simplify some complicated network partitioning situations described below. Very large scale cells would also require that the optional messages sent between group servers and outlined above are not sent. Even if some small amount of traffic does go between Group Servers, 100 or more such servers can participate in a cell concurrently.

The real scaling constraint for very large environments is making the definition of each group optimal with respect to the Sphere Of Interest of the users on nodes in that group. If many users have spheres which span many groups, and they execute operations which require interrogating the loads or status of the machines in their sphere, the overhead will grow quickly. Thus, providing tools to help tune the number of groups, the size of each group and the list of

nodes in each group is very important in the large scale environment.

NETWORK PARTITIONING

The goal of the **FUSION** software is to be no more dependent on DCE services (Name Service, Kerberos, VLDB), than DCE itself. Being partitioned away from one or more of these services will no cause any additional limitation due to **FUSION**.

With respect to **FUSION** function, network partitions can come in two flavors. The first is a partition on a group boundary. The other is a partition inside a group. Below we look at each in turn.

PARTITIONS BETWEEN GROUPS

If a group is isolated from the other groups then it will work independently. Periodically each side of the partition will try to reach the various Group Node Status Services in the other partition so that reconnection can take place automatically. On reconnection, the GNSSs will merge, recording the existence of each other, exchanging a list of nodes supported by each Group Server and the list of clusters supported the GCMS in each GNSS. If any of the clusters overlap, a cluster merge is attempted.

PARTITIONS INSIDE A GROUP

Inside the group one could end up with two or more independent instances of the group, each with an active GNSS. As with the inter-group case above, each of these group instances will be periodically trying to find the rest of their group but, until they do, they operate independently. When the subgroups are able to communicate, they join into a single group, with a single GNSS. This presents no particular problem unless either some of the nodes in one of the groups cannot communicate with this chosen GNSS or there is a problem in trying to cluster merge because of some mounts conflict.

The proposed resolution of the first problem is that nodes which cannot communicate with their GNSS will either be isolated or will form a subgroup of nodes with the same problem.

The proposed resolution of the second problem is that a unified cluster will be created, but the node causing the conflict (by definition the node in the subgroup whose GNSS is becoming inactive in favor of the other subgroup's GNSS) will not be able to join the unified cluster and will become isolated.

Both these policy proposals could be altered if necessary.

3.1.3.3 SPHERE OF INTEREST

In a large cell or cluster, it may be important to be able to limit the list of nodes to be considered for various operations. Automatic load leveling is one such area. Additionally, it is desirable for applications which display information about more than the local machine to restrict their scope to a small

set of nodes about which the user really cares. This functionality is called the Sphere of Interest.

The Sphere of Interest (SOI) is a list of nodes associated with each process. The SOI is purely advisory. The presence or absence of a node on the list in no way allows or prohibits any kind of access to the node. The list is used purely to narrow down the list of nodes which would be considered under certain default conditions (distributed who, automatic node selection, etc). For convenience, a sphere of interest aliasing mechanism is provided so that names can be given to commonly used spheres.

The SOI may be changed arbitrarily by any process, though the most common case would be for a login shell to set the Sphere and leave it alone after that. Some system utilities, including those modified to consult with the Node Status Service such as who, finger and mount, will search the user's home directory for a file named ~/.sphere/PROGNAME. If a such a file is found, its contents will be used as the sphere for that program. This allows users to customize their SOI on a program by program basis. If the file is not found, the SOI associated with the process and used for node selection will be used for any application making calls to NSS.

3.2 THE FUSION CLUSTERING CAPABILITY

The **FUSION** cluster environment relies heavily on most of the DEE function of Section 3.1. Augmenting that function are the clustering functions, which are the clustering or Single System Image of data and the automatic load balancing of processing, both heterogeneous and homogeneous. Below there are major sections on clustered data, invocation load leveling and dynamic load leveling.

3.2.1 CLUSTERING OF DATA

Clustering of data has two major functional components. First is the cluster mount capability, which, using the underlying DCE distributed file system transport, allows all mount file systems to be visible cluster wide. This provides a uniform file name space. All machines in the cluster have the same view of the file system, going beyond that provided in base DCE.

The second major function is cluster wide NFS coherency, which includes not only mounts but also client cache consistency.

3.2.1.1 THE CLUSTER MOUNT SERVER

The Cluster Mount function complements the DCE distributed file system global naming function in the following way.. In a base DCE environment, only File Systems registered in the VLDB (DCE may have a new name for this) are visible beyond the machine on which they are stored. File Systems mounted by /etc/mount are not visible beyond the machine on which the mount operation is done.

Clearly, to provide a Single System Image out of a cluster of machines, the /etc/mount must be visible cluster-wide (if /etc/mounts are to be allowed at all). There are several possible reasons why an /etc/mount would be used even after DCE is installed. They are:

1. Historically, /etc/mounts are the way File Systems were made visible, so some administrators may wish to continue with it.
2. One may not wish the visibility of a particular file system to be galactic. Currently the only export restriction one has is to /etc/mount rather than use DCE file system registration.
3. One would prefer not to have to rely upon the existence of the VLDB to attain clustered capability.

This section is organized as a series of 8 subsections. First a simple or completely transparent global name space is introduced (including a cluster wide replicated root). Within this model the scope of which objects can be mounted on which other objects is described. Then the functional components of the Cluster Mount Service are introduced, followed by the specification of normal file system operations. Next is the specification of how the Cluster Mount Service works in the context of the Node Status Service. Then the general mount model is presented. The general model relaxes the restriction that all file systems must have a cluster wide unique mount point and allows independent root file systems. The final two sections consider multiple roots per node (useful in preparing for a system upgrade) and the authority required to do various mount operations.

3.2.1.1.1 SIMPLE MODEL FOR CLUSTER MOUNT

The ideal cluster consists of machines with a replicated root and no conflicting mounts, so that file system naming can be constructed into a Single System view. However, the **FUSION** architecture supports a much more general mount environment.

To ease in the explanation, the next few sections are written with the cluster wide replicated root environment in mind. This is the most transparent and simplest environment. Section 3.2.1.1.6 describes the truly general environment supported, together with the associated extended semantics.

3.2.1.1.2 WHAT CAN BE MOUNTED ON WHAT?

The intent is that anything a single-system UNIX could mount can be mounted in the clustered environment and be visible cluster wide.

This includes:

1. any form of physical file system mount,
2. single system semantics read/write replicated file system mounts (see Section 3.3.1),

3. DCE read/only replicated File Systems,
4. NFS remote File Systems, and
5. any namable subtree (directory mount).

Note that mounts of files and mounts of remote devices are not permitted. The remote devices restriction is imposed because, if the functionality was useful, it is simpler to have the mount program function-ship the system call to the appropriate machine (i.e. the machine which has the named device) rather than having much of the function-shipping inside the kernel. However, this means that the remote device must be on a node in the cluster, because the mount done on that node will be visible only within that cluster.

Mounted objects will, in general, have a physical file system id. In addition, as part of the mount, a virtual file system id will be assigned so that two virtual File Systems can refer to the same physical one. Both file system ids will come out of the DCE file system id space so remote open, read, write etc. can operate using the standard DCE remote file system code.

As a default, a mount will be undone either when a umount operation is done on the node where the mount was done or when the node which did the mount becomes unavailable. However, it is sometimes desirable to have a mount persist even after the mounting node becomes unavailable. An NFS mount onto a directory in a replicated file system is one such example. The persistence is accomplished by having the same mount operation done on multiple nodes and by having that mount take a permanent file system id as a parameter (this is one of the changes to /etc/mount referred to in Section 3.2.1.1.3). The functionality is very similar to the concept of administered mounts in AIX/TCF. Besides having a rich set of mounted objects, **FUSION** allows these objects to be mounted almost anywhere. In particular, they can be mounted on:

a. locally stored non-replicated directories;
b. non-replicated directories stored elsewhere in the cluster;
c. non-replicated directories stored outside the cluster and named through DCE on-disk junctions;
d. non-replicated directories stored outside the cluster and named through NFS;
e. DCE read/only replicated directories mounted via /etc/mount;
f. DCE read/only replicated directories accessed via DCE on-disk junctions;
g. **FUSION** replicated directories accessed via /etc/mount;
h. **FUSION** replicated directories accessed via DCE on- disk junctions;

Note that, in the above scheme, the writable replica in DCE replication is considered a nonreplicated directory.

3.2.1.1.3 CLUSTER MOUNT FUNCTIONAL COMPONENTS

The Cluster Mount function is realized by a collection of sub-functions which execute in six different areas. Working together, this code presents a Single System Image mount model to applications and users. The code consists of:

a. hooks in the base Operating System;

b. the Cluster Mount Kernel Extension (CMKE)

c. hooks in the base DCE distributed file system

d. the Local Cluster Mount Service, which is part of the Local Node Status Service (LNSS) (Section 3.1.3.1)

e. the Group Cluster Mount Service (GCMS), which is integrated into the Group Node Status Service (Section 3.1.3.2).

f. system utility enhancements, specifically to create the /etc/mountx program.

The hooks in the base operating system center around the system calls mount and umount, as well as a test in the lookup function. These hooks enable entry to the Cluster Mount Kernel Extension (CMKE). It may be the case that the mount and umount extensions can be integrated via a layer in the vfs code and thus not require any base changes.

The Kernel Extension caches mount information both about the local node and about other nodes. Other node information is obtained either by querying through the NSS to the CMS on the Group Server or by having the CMS tell the Kernel Extension via the local NSS. The Kernel Extension is also used to create mounted on vnodes in the base so subsequent lookup operations will invoke the Kernel Extension.

The hook in the base DCE involves the Protocol Exporter. In the base DCE the Protocol Exporter does not look for mounted-on status in vnodes and thus does not inform the client that a requested vnode is mounted-on. For clients which are clustered, the Protocol Exporter must return this information so all cluster sites see a consistent name space.

The Local Node Status Service is non-kernel code which, with respect to the Cluster Mount function, is largely a conduit from the CMKE to the GCMS. It supports and passes on requests for mounts or unmounts and queries about what is mounted. It also pushes GCMS requests into the CMKE.

The GCMS is also non-kernel code. It is a service which maintains all the /etc/mount information for the entire cluster, so it can:

a. Do consistency checking on subsequent mount requests;

b. Inform nodes when something is mounted on a directory they store;

c. Answer queries about what is mounted where and who stores it.

The CMS can be a replicated service. If it is, the data in each replica is identical (except, perhaps, when the network is partitioned).

The extensions to the /etc/mount program and thus the inclusion of the /etc/mountx program are in two categories.

1. Additional options and parameters which must be passed the CMKE. Uses of these extended options was discussed in Section 3.2.1.1.2 and is referred to in Section 3.2.1.1.6.

2. Display options. Either by default or under optional control, the /etc/mount program is able to display the mount information for all or part of a cluster. This is done, in part, by having /etc/mount gather information from the CMS (possibly through the local NSS) rather than just displaying /etc/mnttab information or whatever /etc/mount was doing in the single-system environment.

For each mount, the CMKE tells the local Node Status Server (NSS) which in turn tells the GCMS. The GCMS determines if the mount is consistent and, if so, tells the other GCMS replicas. The GCMSs tell the necessary NSS(s) so the incore mounted-on bit can be set.

3.2.1.1.4 CLUSTER MOUNT NORMAL OPERATIONS

In this section we describe how the Cluster Mount function operates to:

a. join a cluster;

b. leave a cluster;

c. mount a file system;

d. unmount a file system;

e. do a lookup through a mount point;

f. do mount queries from /etc/mount;

g. support a mount or umount done on another node.

h.

a. Joining a cluster involves:

    1. Having the CMKE push the request to the NSS along with all the local mount information;

    2. Having NSS negotiate with CMS. CMS gets the requested local mount information from the NSS, analyses it for inconsistency

with existing mounts in the cluster, and if there are none, propagates the information to any other CMS's for the cluster.

3. Any of the CMS's may, as a consequence of the requesting machine joining, tell other nodes to keep some mounted-on vnodes incore. This is done by the CMS pushing information to the NSS on the relevant nodes, and those NSSs pushing the information into the CMKE, which in turn sets up the mounted-on vnodes in the base operating system.

b. Leaving a cluster cleanly simply means telling the CMS, which in turn will tell the other CMSs. Together, the CMSs will inform any CMKE on any other node if some cleanup is necessary. Additionally, the local CMKE will clean up unnecessary incore mounted-on vnodes. Unscheduled departures from the cluster are triggered via the Group Server (Section 3.1.3.2) but are semantically the same as scheduled departures. When a node leaves the cluster, processes from that node that are now on other nodes will be permitted to continue executing, since the remote processing support is enabled outside of a cluster. However, the processes mail fail if they were dependent on resources on that node that had been available only by clustering mechanisms. Also, parts of the mount context of that process may become unavailable.

c. The mount operation is a subset of the joining the cluster operation described above.

d. The umount operation is a subset of the leaving the cluster operation described above.

e. The lookup or pathname resolution software is altered only slightly to work in the clustered environment. Upon finding a mounted on vnode (either locally or from a file system server in the cluster), the lookup code must not only check the standard kernel mount table but also call the CMKE which might redirect the lookup to a new file system, potentially on a different node.

f. Mount queries from /etc/mount will go directly to the NSS, which can either answer the query directly or will get the necessary information from the CMS at the Group Server to answer the query. These queries will be influenced by one of the Spheres Of Interest associated with the user or process (see Section 3.1.3.3).

g. Lastly there is supporting the effect of a mount or unmount done on another node. Many remote mounts and unmounts will not require any activity on the local node, but some will. In particular, mounts done on directories stored locally or not stored in the cluster at all will involve a message from a CMS to the local NSS and down to the CMKE. If the operation is approved, the CMKE will then record the operation and

create an incore vnode with a mounted-on bit set.

Unmounts will undo what mounts have done.

CACHING MOUNT INFORMATION

For most non-local mount points encountered during pathname lookup, the CMKE will obtain mount information from either the NSS or directly from the CMS. It then caches this mount information to avoid requesting it again later. The CMS need not keep track of which clients have cached which mounts because stale mount information will be detected if it is used and corrected at that time due to the token mechanisms. The performance in the case of crashes of mounters or clients is no worse than with the DFS in DCE.

3.2.1.1.5 CLUSTER MOUNT OPERATIONS WITH NODE STATUS SERVICE

There are several types of operations that involve the NSS and CMS. First, the NSS must find its controlling CMS. The operation is done as a side-effect of finding the Group Node Status Services since the CMS function is done as a subservice of the GNSS. By definition, one's Group Service Node is also one's CMS Node. The Group Service Node is found by searching an ordered list kept either by the DCE Directory Service or in a highly replicated DCE file. More detail on Group Service interaction is given in Section 3.1.3.2.

Having found the CMS, the NSS does the Cluster Join described in Section 3.2.1.1.4. The Cluster Leave, mount, umount and lookup operations were also discussed in Section 3.2.1.1.4, as were the effects of other nodes doing mounts and umounts.

If the NSS detects the loss of a Group Server (and thus the CMS), it attempts to find a replacement using the ordered list just described. Joining the new CMS is very similar to the initial join except the new CMS is told that this node was already a cluster member. To avoid arbitrarily complicated interconnection topologies, the selected backup CMS will only agree to be CMS if it agrees that all the higher priority nodes are unavailable. The general principle is to have only a single active Group Server (and thus a single CMS per group).

A situation can arise where two or more Group Servers for the same group can't communicate and thus each believes it should be the Group Server. If more than one such Group Server can communicate with another group's Group Server, that other group's Group Server will help to arbitrate to the more senior local Group Server. Another group's Group Server should never see more than a single Group Server per group.

In one of these unlikely scenarios, a given node may become isolated with respect to clustering and other group services.

If a backup has become Group Server, it periodically attempts to find the principal Group Server for the group (or any more senior Group Server node).

In addition, the principal Group Server will, on startup, attempt to contact the backup and the other group's Group Servers. Once one of the schemes determines that the principal is back, the backup Group Server tells all its NSSs to move back to the principal Group Server. The backup then ceases to be a Group Server and starts using the principal. The CMS service moves with the other Group Server services.

INTRA CLUSTER MOUNT OPERATIONS

This section outlines the operations which occur between replicas of the Group Cluster Mount Service. Since the Group Service can have a scope broader than a given cluster, it is possible for a Group Server to be running without the CMS for a given cluster. When the first member of that cluster joins the group and subsequently does a cluster join, the Group Server initializes the GCMS for that cluster and group and finds any other instances of the CMS. Having found an instance, the new CMS downloads the CMS replicated data and then attempts to add the new client to the existing cluster.

Network partitions due to gateway failure make it possible for two active CMSs to be operating in a stable state for the same group. When the gateway reappears, the two CMSs will suddenly see one another. The goal at this point is to have the more senior Group Server take over and the other Group Server to stop functioning. There could, however, be mount conflicts which could hamper a smooth takeover. The resolution of this class of conflict is that the conflicting groups will not join until the conflicts are resolved.

Gateways becoming available can also cause two or more principal Group Servers for different groups to join. This is done, in the most general case, by having one CMS from one partition negotiate with one CMS from the other partition. The pair exchange client mount information. If there are conflicts, the partitions remain as they were, effectively ignoring the fact that they can communicate. Periodically they will try to rejoin. Any conflict preventing the rejoin will be logged in system error logs. If there are no conflicts then a merged set of data is sent to all CMSs. They, in turn, disseminate the necessary information to the appropriate NSSs, as was done for any given mount on a remote machine.

CLUSTER MOUNT OPERATIONS DURING UPDATES

Adding a single node to a cluster is done in a two- phase, serialized manner. Between phases, questions concerning mounts on the node(s) joining are deferred until the completion of the second phase. Queries about all other mounts on all other nodes are answered with no delay. Losing a node is handled similarly.

Losing a CMS would mean losing a whole group of nodes when perhaps only the CMS node itself became unavailable. Consequently all the nodes which were in the cluster and which were being serviced by that CMS are marked as

neither up nor down. Mount queries concerning them are deferred back to clients and retried until the new CMS for that group is fully operational.

To become fully operational a new CMS first talks to the existing CMS servers, learns which nodes in its group it should be servicing, polls those nodes and then waits for replies. As each node is found it is marked as operational. The loss of the CMS does not cause any other nodes to be marked as unavailable. If indeed some machines are unavailable, the new Group Server will discover their unavailability just as any Group Server would.

3.2.1.1.6 THE GENERAL CLUSTER MOUNT MODEL

While the replicated root model outlined in Section 3.2.1.1.1 provides an important data point for distributed systems, reality is that some very interesting environments exist and will continue to exist where the file system naming scheme across a collection of machines is not fully integrated. For example, machines not already clustered may have conflicting directory hierarchies and mounts. Machines from different vendors will have slightly different root file layouts.

Clustering, load leveling and remote execution are very interesting in these environments. The model presented here deals with some of the areas where integration may not be practical, namely:

a. Independent root File Systems on different machines or more than one independent replicated root;

b. Independent /tmp mounted File Systems;

c. Mounts of machine-type dependent binaries;

d. Mounts of node specific data (/var, perhaps);

e. Multiple independent user File Systems with the same name (e.g. /u).

Again, while it is recognized that a completely integrated naming environment improves transparency and is encouraged, the architecture supports a much more general environment.

This model is designed specifically for a collection of machines, some of which have an independent root and some have replicated roots. However, the model really treats the root like any other file system and thus allows independent versions of other File Systems as well, while striving to retain the Single System Image. The mount model allows two flavors of mount and an interface to uniquely name all data in the cluster. Specifically:

1. If a mount is applicable only to a specific machine and would conflict with similar mounts on other machines, it can be mounted "locally". The root file system is considered to be mounted this way, with all roots being "local." /tmp and /var are other examples of mounts which would qualify as being "local". Mounts of this variety would issue warnings if

a cluster-wide mount had already been done at that mount point. "Local" mounts are not the default.

2. By default, mounts are cluster-wide. If the mount is on a directory in a mounted file system which is visible cluster-wide, only nodes with copies of that directory need be notified. If the mounted- on directory file system is itself "locally" mounted, all nodes must know about the mount. Note that, using these semantics, one wants to be sure that all nodes are prepared for the requested mount at the mount point indicated. If some machine had a subdirectory it was using at that mount point directory, that subdirectory would disappear.

   Section 3.2.1.1.8 explains that this form of mount should require special privilege. Avoiding this form of unintended behavior could optionally be done by only allowing mounts on empty directories or by using a directory attribute bit which indicates "mounted on" and only allowing mounts on marked directories. While either of these restrictions could be useful in a given implementation, neither is required in the architecture. These cluster-wide mounts would fail if there was already a cluster-wide mount at the same mount point.

   If the cluster-wide mount can not be successfully executed on each machine (perhaps because the mount point is a file on some machine), a warning will be issued to allow intervention to eliminate the problem.

   Note that a mount point not existing in some environment will often be solved by having it created automatically.

3. All data in the cluster is uniquely nameable within the cluster via a CMS interface. For instance,

   ```
   /.../cellname/clustername/CMS/nodex
   ```

   would name the root on Node X.

Note that this "name" for the root of any given node is not like a second mount of that root. Multiple mounts of the same data, if supported by the underlying base operating system, do not in general preserve nested mounts.

Said differently, if you mount on one pathname and the mounted-on directory is available via a different name, the mount will not be seen via that different name. This is the case whether the mounted-on directory is nameable via other /etc/mounts or via DCE on-disk junctions. The CMS interface "name", however, is a service which directs naming to the appropriate file system, consequently submounts are seen.

Below we consider the feasibility and semantics of this model.

IMPACT ON THE CUSTOMER

This model potentially requires virtually no disruption to the single machine environment to become part of the cluster. In the simplest case the administrator need only change his /etc/mounts to do "local" by default. If mounts aren't made "local", however, they may get conflicts or may mount over a subtree on another machine (/bin on one machine is mounted, while it is part of the root on another machine).

To attain the greatest degree of Single System Image, one would want all mounts to be cluster-wide. To do so would probably require at least some reorganization and/or integration (e.g.. the TCF <LOCAL> context dependent symbolic link).

The only other impact on the customer is the necessity of determining the permissions necessary to do various types of mounts. The system administrator must set up these permissions (see Section 3.2.1.1.8 for details).

IMPACT ON THE SINGLE SYSTEM IMAGE

The more "local" File Systems there are, the less transparent the environment becomes. The straightforward "make everything local" installation strategy could get a cluster off the ground quickly, but an effort should be made to allow some File Systems to be cluster-wide.

It would be advantageous, for example, to integrate the various /u File Systems that might exist on different nodes in the cluster so that a single, perhaps replicated, /u could be seen cluster-wide and different contents were not seen from different machines.

SCALING CONSIDERATION

For "local" mounts it should not be necessary to alert every member of the cluster. However, cluster-wide mounts done on directories which are themselves in a "local" mounted file system will, in general, require alerting each cluster node.

A given machine can avoid being notified when there are mounts done on its locally stored File Systems by preissuing mounthint calls on the directories other machines might do mounts on.

The effect of these calls is twofold:

a. It puts a mounted-on vnode for the designated directories incore so that subsequent lookups investigate to see if a mount does exist at this potential mount point.

b. The CMS is informed that the hint is registered. Thus, when a cluster-wide mount is done, the CMS can determine that no message need be sent that node.

MOUNT CONTEXT

Given that there are potentially different root layouts on different machines and that one would like to be able to look at a different node's mount tree and perhaps change to that node's mount tree for remote execution, we have added a general set of mount contexts to each process. A mount context specifies which node's mount context a lookup is to evaluate when a mount marked "local" is encountered. The full generality outlined in this section may seem excessive for the non-replicated root case but will be needed when replicated roots are used.

There are actually three mount contexts for each process.

The first is the root relative mount context. It is needed to determine which node's mount context to use when a pathname begins with /.

Explicit changing of the root relative mount context (or RRMC) is made available via a Cluster Mount Kernel Extension (CMKE) chmntcontext call. If this call succeeds, it changes the RRMC to the indicated node (chmntcontext takes a node identifier) and changes the u.u_rdir (root directory used in root relative pathname lookups) to the root in the RRMC. The call fails if the node is unavailable or if a preexisting chroot had already set the process's u.u_rdir somewhere other than the root of the node in the current RRMC.

The other mount contexts are the dot-relative mount context and the lookup mount context. These are not changed explicitly. Their need and use is outlined below.

As we have outlined, all data in the cluster is cluster-wide visible via the CMS interface and name structure like

```
/.../cellname/clustername/CMS/nodex/.......
```

In the rest of this section we shorthand this syntax to

```
"CMS/nodex/....."
```

The semantics of naming through this pathname into other node's mount context should be:

1. A single lookup through CMS/nodex/... should do lookups with respect to Node X's mount context so one can say

   ```
   ls -l CMS/nodex/tmp
   ```

   and get the obvious result.

2. A change directory (cd) to CMS/noden should allow dot relative naming to be in Node N's mount context while retaining root or / relative naming to stay as it was (for finding binaries, for example). Thus a

   ```
   cd CMS/noden
   ```

   followed by

   ```
   ls -l /tmp
   ```

yields the same result as

```
ls -l CMS/noden/tmp
```

3. While cd'd to CMS/noden, a reference to CMS/nodex/tmp should be done in the mount context of Node X.

A reference to /bin/grep should be done in the same mount context the process had before the cd. To accomplish the semantics outlined above, each process should have three mount contexts which are defined to operate as follows:

a. A root relative mount context, which changes only explicitly via the chmntcontext call described earlier or via some remote process operations described in the next section. This context is used for all root relative naming (by putting the value in the lookup mount context described below).

b. A dot relative mount context, which is used in lookups which are dot relative (again by putting it in the lookup mount context described below). This value is inherited in fork, exec and migrate and should represent the mount context of the current working directory (cwd). It is changed by:

   1. cd through CMS/nodex will change it to Node X's mount context.
   2. cd /anything will change it to the value in the root relative mount context.
   3. cd .. from CMS/nodex will change it to the root relative mount context. Note that cd /.../cellname/clustername/CMS is illegal.

A lookup mount context (LMC) which is set equal to either the RRMC or the DRMC at the beginning of each name expansion. It changes during the name expansion only if the name goes through the CMS (e.g.. /.../cellname/clustername/CMS/nodex). If the name does go through the CMS, the LMC is changed to Node X's RRMC for the duration of the current name expansion.

REMOTE EXECUTION SEMANTICS

In this section we discuss the desired semantics of various forms of remote execution with respect to the root relative mount context. For this discussion there are three classes of remote execution -

1. explicit command, via on,
2. explicit system calls, and
3. implicit or load leveled system calls.

The on program will cause the exec to end up with a root relative mount context of the destination node. Explicit rexecs, rforks and migrates (the

system call, not the signal) will also change the root relative mount context (RRMC) by default to the destination node. However the capability is provided for the caller to override these semantics and retain the originating node's context.

Normal load leveling operations, which include implicit remote execs, remote execs with the load leveling options, migrate with the load leveling option, and migrations induced via SIGMIGRATE are handled differently. The default behavior is to retain the root relative mount context of the process. This insures that the program will produce the same results if the load leveling operation had not been performed. However, a flag bit in the load module header will be assigned to override this behavior. If this bit is 1, then the root relative is set to the new node where the process is running. This bit would be set for programs that are computationally intensive and favor long term availability over complete single system semantics.

Depending on the policy chosen, there is the possibility that the node corresponding to a given process's root relative mount context could be down.

One would expect that the process would subsequently get failures for root relative pathname lookups. This would not be the case if the root from that node is replicated. In that case lookups can continue to occur successfully until a "local" mount point is encountered.

A mechanism to allow the lookup to continue to succeed (assuming the data in question is still available, for example through dual path disks, replication or because the data was not on the unavailable node) could be created by extending the CMS to retain mount information on nodes determined to be unavailable. Specification of this capability is has been deferred.

3.2.1.1.7 MULTIPLE ROOTS ON A SINGLE NODE

An interesting case of multiple roots in a cluster could occur when one wanted two roots on the same node. The rationale for wanting such an environment concerns operating system upgrades.

In some environments, the system administrator may wish to retain the old root and system utilities for users while simultaneously a new root is tested and possibly customized. In such an environment, one would expect the new kernel to be running and the assumption is that, not only can that kernel support the dual root environment, but it can also run the older versions of the binaries successfully. If a problem with the new kernel arises, the old kernel can easily be rebooted with just the old root environment.

With respect to the cluster mount model we are well poised to support such a capability with the function outlined above. The major extension needed is the ability for one machine with one kernel to act as two "nodes" with respect to the cluster mount function and for operations directed to a machine to be identified as to which "node" they pertain to.

Specification of this capability has been deferred.

3.2.1.1.8 MOUNT AUTHORITY IN THE CLUSTER

This section discusses the issues of which users and what authority are needed to do the various mounts discussed in this section. The architecture does not require any particular privilege, but a given implementation may find the guidelines below important.

First, one should expect the agent doing a mount to have write access in the directory being mounted on, since one is effectively changing the contents of that directory. One would expect this rule to apply to all mounts. It is not specific to clustering or DCE. Second, for local mounts, one shouldn't need any more authority than that imposed by each local machine. Similarly, cluster-wide mounts done onto objects which themselves (and their parent tree) are not local, should not require special privilege. Cluster-wide mounts on local objects may require additional privilege as described below.

The reason for special restrictions on cluster-wide mounts on an object that has any local parentage is that the system will attempt to create a similar mount in each instance of the local mount, causing potentially significant effects. As an example, imagine if a user had permission to mount on his local /bin. If that mount were made cluster-wide, then the entire cluster might become non-functional. It seems appropriate to require greater authority for such potentially dramatic effects.

3.2.1.2 NFS INTEROPERABILITY IN THE CLUSTER

With respect to NFS 4.0, there are three areas where NFS functionality must be enhanced to meet clustering goals. The first area is the mount model, which must be expanded to allow cluster wide visibility. The second area is file access in the face of remote processing. The third area is added cache consistency for the cluster so cooperating processes on different nodes see single system semantics. Below is a discussion of each of these areas of enhancement.

3.2.1.2.1 NFS CLUSTER MOUNT MODEL

The NFS cluster mount model goal is that any node within the cluster can do an NFS mount that will subsequently be seen by all nodes within the cluster. Each node communicates directly with the NFS server, without the need to double hop through the mounting node.

Normally, when the node mounting a file system goes down, the file system will no longer be mounted throughout the rest of the cluster. This is not always the appropriate action for an NFS mounted file system. The design therefore provides for NFS mounts to optionally persist if communication is lost with the original mounting node.

The potentially large number of nodes within a cluster make it impractical to notify each node when an NFS mount is done. Instead the mounting node will

contact the cluster mount service with the mount information. There are two methods by which other nodes within the cluster can acquired the mount:

1. A local process traverses the mounted on vnode during path name evaluation.
2. A process which is accessing the file system is migrated onto the node.

In either case the needed mount information is obtained from the cluster mount service. Setting up the mounted-on vnode can, in some cases, require communication through the CMS to each cluster node (see Section 3.2.1.1).

Without special token code, mounting anything on an NFS directory would require communication with each node. This overhead is avoided by having the NFS token manager (Section 3.2.1.2.3) keep track of NFS directories which are mounted-on, thereby allowing each node to discover the mount only when needed.

3.2.1.2.2 REMOTE PROCESSING AND OPEN NFS FILES

Whenever a process migrates from one node to another the vnodes for the open files will be moved along with the process if they don't already exist on the migrated to node. This movement of vnodes is required because the only way an NFS vnode can be created is by looking up the name of the file/directory in the directory that contains it.

The first time a given user migrates a process from one node to another it is likely that the mount information for the vnodes that have been migrated will not exist on the migrated-to node. Then it will be necessary to do the mount without having the mounted-on vnode for the root. It is retrieved from the CMS only when necessary. If having the mounted on vnode were to be required, then a single mount could require an unknown number of mounts to get back to a file system that was already mounted. The vnode for the current working directory will also have to be moved to allow lookups that do not start at the root to function.

File locks are supported in the kernel. The base code is modified to support moving a process from one node to another. In addition single-system semantics within the cluster are fully supported.

AUTHENTICATION

The NFS DES conversation key encrypted verifier contains a time stamp that must always be chronologically greater than the previous transaction. This makes the use of a common conversation key between multiple nodes impractical from a performance point of view. Any node desiring to use the key would have to synchronize every request with a common registry. If a migrating process accesses data that is mounted via secure NFS, it will be necessary to transport the user's login string to the new node. The login will already be available, as the local NFS security routines will have an ongoing

need for it. This data will be transported in a secure fashion.

3.2.1.2.3 USING DCE TOKENS TO SUPPORT NFS COHERENCY

NFS has no built-in mechanism to provide single-system semantics across multiple nodes. Said differently, there is no strict cache coherency between NFS clients. While one might argue such a capability is inherently important, it is even more important in the clustered environment. In order to provide this, tokens will have to be added to the NFS client, similar to those in the current versions of TCF and DCE.

There are three major differences between the tokens required for NFS and DCE.

1. 1) The first major difference between DCE token management and NFS token management is that the NFS token management cannot happen at the NFS storage site as it can in DCE. In DCE requests for tokens go back to the token manager that serves the physical file system that is exported. There is no token manager for an NFS exported file system from a non-**FUSION** node.

   The **FUSION** code will provide a token manager for each NFS mount.

   Because the token manager node is not the storage node, there are times when it will be necessary to merge multiple token managers into a single token manager. In general this will happen whenever two token managers within the same cell and for the same file system notice the existence of each other.

   There are two primary events that lead to token consolidation. The first is when a gateway connecting a cluster together comes up. The second is when a process is migrated to another cluster that already has the same file system mounted. In either event consolidation will be by recalling all the read/write tokens. This will force all data to be written to the NFS server. If multiple write tokens for the same file existed, data might be lost, as in normal multi-node NFS operation. If open conflicts exist they will be allowed to remain until the file is closed.

2. The second difference between DCE tokens and those needed for NFS concerns the way directories are treated. DCE locks directories much the same as they do files. As NFS caches only the existence of directory entries it will be necessary to get the write token only when entries are to be either deleted or changed. In NFS, directories cannot be locked by range. It will therefore be necessary to recall the read token for an entire directory whenever the write token is given out.

3. The third way tokens have to be treated differently is in the area of timeouts. This is the result of the token manager and the storage node not being the same node. If an NFS server goes down after a client has

been requested to give back a read or write token the client may not be able to comply for an indefinite period of time. In the DCE case, if the server goes down the node requesting the token back has also gone away so an analogous situation cannot occur. For NFS tokens, it will therefore be necessary to have a WAIT reply to a request for the return of a token. Clients receiving a WAIT reply must retry their requests after the wait interval returned in the WAIT reply has expired.

The DCE token mechanism will be used to support single- system semantic file locking within the cluster as well as cache consistency.

There is no provisions in the **FUSION** NFS cache consistency support for interaction with DCE client mounts of NFS file systems. If the same file system is mounted with NFS via DCE mounts and a cluster mount, cache inconsistencies may result.

3.2.2 INVOCATION LOAD BALANCING IN THE CLUSTER

With the **FUSION** cluster environment, the automatic application of the **FUSION** remote processing functions becomes feasible. Once a clusterwide name space is established, programs can run on any node within the cluster and see the same computation environment as any other node. So if the system decides to run a process on another node to help balance the load, the user will not be given inconsistent results.

Making intelligent automatic load leveling depends heavily on the services of the Node Status Server, which was described in Section 3.1.3.

Load leveling services can be obtained in **FUSION** by a several different methods. A Kernel Extension supplied by **FUSION** can be used to automatically select an execution site within the cluster. **FUSION** also supports having a node administrator install an alternate user supplied kernel extension to support alternate algorithms. **FUSION** supplies a set of library routines so that new applications can be written to make application specific load leveling decisions. The system calls rexec and rfork take an explicit parameters to cause the new node to be selected automatically. Below we consider each of theses in turn.

3.2.2.1 LOAD LEVEL KERNEL EXTENSION

The **FUSION** Load Level Kernel Extension is invoked if any of the following events occurs:

a. a load module marked for load leveling is encountered during an exec call;

b. an rexec or rfork call is done with special parameters that designate an automatic load leveling should be performed;

c. an exec is done of a load module that cannot be run locally and thus must run remotely and should thus be load leveled.

One hook in the base exec code, inserted after the load module has been selected and its header has been read, is enough to invoke the Kernel Extension to select the node for execution. In selecting the node, the Kernel Extension considers at the least the following:

— machine attributes

— sphere of interest of the process

— node execution permission of the process

— loads and other resource utilization on nodes

— load scaling factor and preference.

3.2.2.2 USER INSTALLED LOAD LEVEL EXTENSION

A customer can install an additional Kernel Extension which the load level Kernel Extension will recognize and call. The added Kernel Extension, which has full access to the NSS information, can chose to do node selection in some or all cases. For cases it does not want to handle it can defer back to the supplied Kernel Extension.

3.2.2.3 LOAD LEVELING LIBRARY ROUTINES

FUSION provides a set of library routines so an application can make its own load leveling decisions. These routines provide a uniform interface for accessing the load leveling information from the Node Status Service. In addition to the information provided, the application may wish to take other factors into account, such as time of day, availability of other machines or resources, or information that is pertinent to that application.

After making a decision on which node to run, the application can use the rexec, rfork or migrate with an explicitly node argument to effect its own load leveling.

3.2.2.4 LOAD LEVELING VIA SYSTEM CALLS

As outlined in Section 3.2.2.1, rexec and rfork system calls can take a parameter which indicates that loads balancing should be considered in doing the operation. The load level kernel extension actually does the decision making based on Node Status Service information.

3.2.3 DYNAMIC LOAD BALANCING IN THE CLUSTER

Much emphasis has been put (and rightly so) on node selection at process invocation, or exec, time. Nonetheless, dynamic load balancing via process migration is the correct load balancing mechanism for some long running processes. Since process migration is restricted to occur between homogeneous nodes. Those nodes, however, could be part of a heterogeneous cluster so process migration is not limited to homogeneous clusters.

Outlined below are the functions and facilities in **FUSION** for dynamic load balancing. Included are sections on marking load modules for load balancing,

system call extensions for load balancing and load balancing daemons.

3.2.3.1 LOAD BALANCING SPECIFIC PROCESSES

As was mentioned in Section 3.2.2.1, executables in the **FUSION** environment can be marked as load leveling candidates. Such a mark (to be extended to general attributes in the future) is used not only at invocation time but also during execution. The process is a candidate for process migration.

To facilitate this form of load leveling, the load level indication from the load module header is stored to the vproc so process migrators know which processes are the best candidates.

3.2.3.2 SYSTEM CALL EXTENSIONS FOR DYNAMIC LOAD LEVELING

There are basically two system call extensions for dynamic load balancing - the migrate system call and the SIGMIGRATE signal. These can be used in a couple of different ways. One way is for a process to provide a specific node to which that process is to migrate. The other is to specify a load level parameter and cause the load leveling kernel extension to select an appropriate node as a destination for the process.

3.2.3.3 PROCESS MIGRATION LOAD BALANCING SERVICE

A user or system administrator can do manual load leveling by just sending SIGMIGRATE signals to processes or process groups.

Additionally a user could run his own load leveler. It would consult NSS data and move his processes around to optimize their elapsed run time. On a cluster wide basis a system administrator could enable the load leveling service that consulted NSS data and moved eligible processes around to optimize processor utilization and consequently reduce task run time.

While the details of a process migration load leveling service are not included in this specification, a sample dynamic load leveling program will be provided as part of the **FUSION** offering.

3.3 THE FUSION FILE SYSTEM REPLICATION SERVICE

The third major area of **FUSION** is provision of a single-system semantics read/write file system replication service. This service leverages off the DCE distributed file system capability to provide a richer and more transparent form of file system replication than is available in base DCE. This section specifies the services of the **FUSION** File System Replication Services.

3.3.1 FILE SYSTEM REPLICATION SERVICES

Data replication allows multiple storage nodes for a given file system. There are several reasons why such replication is important in a distributed file system environment.

1. The first involves availability needs. It is unacceptable for the failure of a single node to put thousands of other nodes out of operation because of

the unavailability of required data files and programs. Replication allows all critical user and system files to be simultaneously stored on multiple servers. Thus, if one node goes down, another node immediately resumes service, without any apparent disruption.

2. Another key reason for replicated data is performance. Having more than one node service a file system decreases the load on the servers. This provides increased performance for the system. However, writes to the data are more expensive because of the cost of synchronizing multiple copies.

LAZY REPLICATION

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 infrequently done. There are three primary limitations of "lazy" replication:

1. A different pathname is necessary to access the read/write replica than that used to access the read-only replicas, and
2. The read/only replicas are not guaranteed to be current but simply no more than "n" minutes or hours old. Sometime within "n" minutes or hours, the read/write volume will be cloned and the clone copied to the local storage node, thus updating all files.
3. The full functioning lazy replication requires support for the Episode physical file system.

The advantages of DFS replication are:

1. The amount of network traffic is limited.
2. There is less latency seen by programs modifying the replicated file system.
3. The cloning mechanisms provides a good way to make backups.

FUSION Replication Additions

In addition to DCE lazy or read/only replication, **FUSION** offers two other forms of replication, tight and loose. Both the new forms allow access to the read/write and read/only copies via the same pathname and neither require clone capability on the underlying physical file system.

The primary difference between tight and loose types of replication is latest version guarantee. "Tight" replication guarantees UNIX semantics and access to the latest available version, while "loose" replication does not.

"Tight" replication guarantees UNIX file semantics. It is provided for files which will be updated more often than the "read-mostly" file systems, and for

which up- to-date versions are essential. In "tight" replication the replicas are updated on a file-by-file basis as soon as modifications have taken place. Access to file is handled by the local server only if that server's copy of the file is up-to-date and the modifiable replica is not currently undergoing modification. Otherwise, the file access will be handled remotely, either by a server which stores the latest version or by the server on which the modifications are currently being made.

In the most general case, "tight" replication does not scale gracefully beyond a few dozen replicas (The TCF replication scheme is similar to **FUSION** tight replication and was capable of supporting up to thirty-one replicas). After that the read/write replica can become a bottleneck. With less than general update frequency, however, scaling levels can comfortably increase. Nonetheless, scaling to hundreds or thousands of replicas requires "loose" replication.

"Loose" replication does not guarantee that changes are seen as they are happening. The changes are seen shortly after they are completed. System files, including program binaries and manuals are good candidates for loose replication. As would be expected, loose replication will scale more gracefully than tight replication, primarily because access is always local if a copy is stored locally, without needing to ensure that the latest version is being accessed.

3.3.1.1 CREATING FUSION REPLICATED FILE SYSTEMS

A file system is designated as either "tight" or "loose" either when it is created or when it is converted from a non-replicated file system. The initial replica is also designated as the read/write replica (this can be changed later, see Section 3.3.1.6). The rest of the replicas are created as read/only versions of the read/write replica. As a scaling aid, two types of read/only versions can be designated, principal and secondary. The optional designation can allow hierarchical propagation of changes, as described in Section 3.3.1.4.

Although subscription into the VLDB is not necessary, a permanent volume number must be assigned so that replicas can be identified as such by having the same volume number.

Any underlying physical file system type can be used for **FUSION** replication and replicas need not all have the same underlying type.

3.3.1.2 THE FUSION REPLICATED MOUNT MODEL

Any node within a cell can function as a replication storage node for any file system within the cell. There is no requirement that all replicas be within a single cluster. However, for replication to work outside the cluster, the replica must be mounted using DCE on-disk mounts. Using /etc/mount to mount a replicated file system will only work within the cluster.

In addition, only nodes running **FUSION** can function as storage nodes for replicas. The propagation routines are part of the kernel and a non-**FUSION**

kernel will not know how to update the replica.

Access to a replicated file is not restricted to **FUSION** nodes, however. The DCE remote file system client will be slightly enhanced so that, if a non-**FUSION** client wishes to access a **FUSION** replicated file system, it will access the read/write replica as though it were the only storage node of the file system.

3.3.1.3 ACCESS TO FUSION REPLICATED FILE SYSTEMS

Section 3.3.1.2 above outlined the form of access available to non-**FUSION** clients. For **FUSION** clients the access model is slightly richer. The access model for regular files and directories is based on how the file is opened, what other activity is ongoing for that file and what subsequent operations are done. Access to special files are directed through the node storing the read/write copy. Note that support for special files in replicated file systems does not result in replication hardware or reference to devices on more than one node. This is supported only to maintain UNIX File System semantics within **FUSION** Replicated File Systems.

More specifically, for tight replication, if the file is currently undergoing modification, all accesses (read, read/write and write) will use the read/write copy. For loose replication, only the opens which are actively modifying the file will use the read/write copy. More specifically:

OPEN FOR READ (Tight Replication)

Opens for read will be assigned to a replica server based, to some extent, on load and locality of the server. If there is another ongoing modification, however, all opens will use the read/write replica.

Even though the open can be assigned to any replica, a replica is chosen only if it has the latest version of the file (see Section 3.3.1.4 for Version Management). If, after the open has been assigned a server, a modification is done by another open, the server for this open for read will be changed to be the read/write server in a way transparent to the user, so strict single system semantics are preserved.

OPEN FOR READ/WRITE (Tight Replication)

Opens for read/write are initially treated just like opens for read only, on the argument that processes often open for read/write but never write. The difference between read only and read/write is that, if the process with this open does a write operation, opens to that file move to the read/write replica.

OPEN FOR WRITE (Tight Replication)

Opens for write use the read/write replica at all times. Whether other opens move to the read/write replica when this open is done or only when the first write is done is a design detail.

OPEN FOR READ (Loose Replication)

Opens for read are assigned a server independent of whether there are ongoing modifications and independent of whether the server chosen has the latest version. The open will stay with that server unless it becomes unavailable. If the file is being propagated to that copy at that the time of the open, the contents of the file may be unpredictable. However, this is consistent with the semantics of existing UNIX systems.

OPEN FOR READ/WRITE (Loose Replication)

Opens for read/write are very similar to opens for read, except that, again, when a modification request is made by the process with the open, the server is changed to the read/write replica. No other opens are moved as a result.

OPEN FOR WRITE (Loose Replication)

Opens for write use the read/write replica at all times. Neither the open nor any subsequent writes, however, cause any other opens to move to the read/write replica.

In any of the above scenarios when a read access is performed, the access time of the file is updated only on the copy that is actually read.

Note from the above description that sharing a file between cooperating processes is not practical in the loose replication case. Programs that attempt to use loose replication in this manner may result in unpredictable errors. Loose replication is intended for read-mostly applications.

Open Shared and Open Exclusive for both tight and loose replication use the read/write replica at all times.

File Locking

Read locks can access any valid replica, while write locks will access the read/write replica. However, all locks will be coordinated cell-wide via the token manager. If a request for a write lock is made at the read/write replica and a user on any local replica holds the read lock token, the attempt to get the write lock token will fail. This ensures full Unix semantics.

3.3.1.4 VERSION MANAGEMENT

It is important to accurately determine, under many failure conditions, whether two copies of a file have the same content, without actually comparing the files. Consequently something in each replica must uniquely describe the version of the data contents in that replica. Modification to the underlying physical file system, as was done in AIX/370, can simplify this task. However, **FUSION** does not use this approach to avoid any changes to the physical file system.

The strategy is to use hidden files to store file system and replica specific information. The file .frfsinfo contains file system implementation specific to replication and is the same on all replicas. When the file system replication services updates this file it is propagated to all copies. The file .replinfo contains replica specific information. Consequently this file is maintained independently by each replica and is not propagated. This file is used to indicate what version of each file is stored in that replica. It should be noted that both of these files are treated specially by the **FUSION** File System Replication Service. They cannot be modified or removed by any user, including the super user.

The general principle is that, on the read/write replica, a new version is declared and that declaration committed before any data changes are made. (Note that a new version could just be a new ctime value and committing is just writing out the inode.) Then any number of changes can be made until another node starts to propagate that version. After the propagation, a new version must be declared before any changes can be made.

These simple principles allow versions at any two nodes to be compared without requiring data to be compared.

3.3.1.5 PROPAGATION

All modifications to files within a replicated file system use the read/write replica, although this is transparent to the user and to the process doing the modification. Below is the propagation model for tight and loose replication, followed by a scaling discussion, and the description of how read/only replicas are synchronized after a disconnect with the read/write replica.

TIGHT REPLICATION PROPAGATION

At the point when a modification request is made, the implicit read token is taken back from all replica servers. This forces all requests to the read/write replica. Each replica records this state so all read or read/write opens can accurately determine whether they can be serviced by a read/only replica or if they must use the read/write replica. When a modification is completed, each replica is notified and can commence pulling of that version of the file. Each read/only replica service keeps a queue of outstanding propagations so bursts of changes will not force repeated unnecessary propagations.

LOOSE REPLICATION PROPAGATION

In loose replication a modification request does not revoke the implicit read token from all replicas; thus, during the update, clients using read/only replicas see the unmodified data. Only at the completion of the update are the read/only replicas notified of the change. After the notification but before the propagation, the read/only replica server continues to export the local version. Access will be switched to the read/write replica during the actual propagation to ensure a consistent version of the file while the local version is being

overwritten.

FUSION REPLICATION SCALING

To accommodate scaling the number of replicas, a hierarchy of read/only replicas can be defined, with principal and secondary read/only replicas. If this is done, only the principal read/only replicas will propagate directly from the read/write replica. The principal replicas are responsible for servicing the secondary replica's requests for propagation.

REPLICATION SYNCHRONIZATION AFTER DISCONNECT

FUSION replication also provides a mechanism by which the read/only replica can obtain from the read/write replica all the modifications it may need in order to come up-to-date. This mechanism is used when a replica has been off the net or has been unable to keep up with propagation and is extremely out-of-sync.

3.3.1.6 ADMINISTRATIVE CONTROLS

FUSION provides functions to create **FUSION** Replicated File Systems, and to add and delete replicas, and to perform other services pertaining to the management of **FUSION** Replicated File Systems. Some of these additional services include the ability to convert read/only replicas to read/write replicas, and back. Additional information is provided in Section 4.3.1.

If possible in the context of DCE and the underlying base operating system, tools will be provided to monitor the distribution of replica copy access to thus allow replication related tuning.

3.3.1.7 FUSION REPLICATION AND NETWORK INSTABILITY

A distributed network is a dynamic environment in which nodes may join and leave a cluster at any given time. Replication allows the disappearance of a server to have minimum impact on the user. The read/write replica manages all reconciliation between servers which are just joining the network. If the read/write replica is unavailable, a read/only replica will serve this function.

With the exception of the read/write replica, the disappearance of a storage node is invisible to the user. If the node goes down, extensions to the DCE cache manager will automatically select another server and the user will continue undisturbed.

If the read/write replica becomes inaccessible and a user is modifying a file, the operation will fail with EIO. If the user is accessing a file, the extensions to the cache manager will automatically select another server and the user will continue. However, the version the user will access may not be the latest version on the read/write replica. It will be the latest version which has propagated from the read/write replica at the time of the it's unavailability.

If the read/write replica becomes inaccessible, the read/only replicas do synchronize to ensure that all replicas have all the latest data.

4. COMMAND AND PROGRAMMING INTERFACE DEFINITIONS

This section describes the command and programming interfaces of the basic **FUSION** components. It includes both system programmer interfaces and system command interfaces.

For the reader's convenience, the subsections of SECTION 4 have been numbered in a sequence consistent with the numbers used in SECTION 3. This allows the reader to page back and forth between the two sections. For any **FUSION** component, the reader can easily find discussions of FEATURES in SECTION 3 and INTERFACES in SECTION 4.

Please note, however, that some numbers appear to be missing from SECTION 4. All subsections in SECTION 4 appear in sequential order, but not all topics covered in SECTION 3 have corresponding headings in SECTION 4. Where a subsection number appears to be missing, it means either that the concept of interfaces does not apply to that topic, or that interfaces have already been sufficiently covered. The interfaces are organized below into two major subsections - those for the Distributed Execution Environment and those for the Cluster Environment.

4.1 THE DISTRIBUTED EXECUTION ENVIRONMENT

The interfaces for the Distributed Execution Environment are described in three sections - the file system enhancement interfaces, the remote processing interfaces and the node status service interfaces.

4.1.1 FILE SYSTEM ENHANCEMENTS OF THE DEE

The file system enhancement interfaces are described in seven sections below: remote device interfaces, remote pipe interfaces, remote socket interfaces, select on remote object interfaces, file offset coherency interfaces and interfaces for file reopen and re-locking.

4.1.1.1 REMOTE DEVICE INTERFACES

No specific command or programming interface changes or additions are need to complement this functionality. However, we are expecting a DCE /etc/mount interface to allow device inodes to be declared as describing devices on the local machine (this is functionality for diskless machines).

The ioctls which will be supported between **FUSION** machines include:

IOCTYPE	IOCINFO
IOCCONFIG	TXISATTY
TXTTYNAME	TXGETLD
TXSETLD	TXGETCD
TXADDCD	TXDELCD
TIOCGSIZE	TIOCSSIZE
TIOCGETD	TIOCSETD
TIOCHPCL	TIOCMODG

TIOCMODS	TIOCGETP
TIOCSETP	TIOCSETN
TIOCEXCL	TIOCNXCL
TIOCFLUSH	TIOCSETC
TIOCGETC	TIOCLBIS
TIOCLBIC	TIOCLSET
TIOCLGET	TIOCSBRK
TIOCCBRK	TIOCSDTR
TIOCCDTR	TIOCGPGRP
TIOCSPGRP	TIOCSLTC
TIOCGLTC	TIOCOUTQ
TIOCSTI	TIOCNOTTY
TIOCPKT	TIOCSTOP
TIOCSTART	TIOCMSET
TIOCMBIS	TIOCMBIC
TIOCMGET	TIOCREMOTE
TIOCGWINSZ	TIOCSWINSZ
TIOCUCNTL	FIOCLEX
FIONCLEX	FIONREAD
FIONBIO	FIOASYNC
FIOSETOWN	FIOGETOWN
SIOCSHIWAT	SIOCGHIWAT
SIOCSLOWAT	SIOCGLOWAT
SIOCATMARK	SIOCSPGRP
SIOCGPGRP	SIOCADDRT
SIOCDELRT	SIOCSIFADDR
SIOCSIFDSTADDR	SIOCGIFDSTADDR
SIOCSIFFLAGS	SIOCSIFBRDADDR
SIOCGIFADDR	SIOCGIFBRDADDR
SIOCGIFFLAGS	SIOCGIFCONF
SIOCGIFNETMASK	SIOCSIFNETMASK
SIOCGIFMETRIC	SIOCSIFMETRIC
SIOCADDNETID	SIOCSIFMTU
SIOCSIFREMMT	SIOCSARP
SIOCGARP	SIOCDARP
SIOCSNETOPT	SIOCGNETOPT
SIOCDNETOPT	SIOCSIFSECURITY
SIOCGIFSECURITY	SIOCSIFAUTHORITY
SIOCGIFAUTHORITY	SIOCSARP
SIOCGARP	SIOCDARP
SIOCSARP	SIOCGARP
SIOCDARP	SIOCSX25XLATE
SIOCGX25XLATE	SIOCDX25XLATE

4.1.1.2 REMOTE PIPES AND FIFOS

No command or programming interface changes or additions are need to complement this functionality.

4.1.1.3 REMOTE SOCKETS

No command or programming interface changes or additions are need to complement this functionality.

4.1.1.4 SELECT ON REMOTE SPECIAL FILES

No command or programming interface changes or additions are need to complement this functionality.

4.1.1.5 FILE OFFSET COHERENCY

No command or programming interface changes or additions are need to complement this functionality.

4.1.1.6 FILE REOPEN AND LOCK INHERITANCE

No command or programming interface changes or additions are need to complement this functionality.

4.1.2 REMOTE PROCESSING ENHANCEMENTS OF THE DEE

Remote Processing Interfaces are described in six subsections below: programming primitives, cellwide process name space, virtual processes, client/server function, execution permission and remote process commands.

4.1.2.1 REMOTE PROCESS PROGRAMMING PRIMITIVES

Programming Interface

The execl, execv, execle, execve, execlp, and execvp system calls have the same interface as standard Unix but their semantics are extended in **FUSION**. The Sphere Of Interest and execution permissions are inherited on exec. If the load module is not marked otherwise, the mount context is also inherited.

The rexecl, rexecv, rexecle, rexecve, rexeclp, and rexecvp system calls are used to initiate a program on either the local node or a remote node. Each system call form has semantics similar to the corresponding exec form. In general, the arguments are the file name to initiate, the arguments for the program, the environment variables, the destination node, and a flag used for Mount Context (see SECTION 3.2.1.1.6). The first three arguments match their meanings for exec. The destination node may be used both to determine the execution node and to determine the execution object, by searching in that nodes mount context when evaluating the file name. The mount context flag is a binary value. If non-zero, the mount context is not changed. If zero, the mount context is changed to that of the new execution node.

The rfork system call creates a new process on the destination node specified. The new process is a copy of the current process. The initiating process's properties are preserved or altered in the same ways as a fork system call. rfork's two arguments are the node specifier of the destination node, and a flag

indicating the Mount Context to use for subsequent name lookups.

The existing fork system call is extended by **FUSION** to pass additional context information to the new child. In particular, the execution node permissions, Sphere Of Interest and Mount Contexts are all inherited by the child process.

The new migrate system call moves the current process to the destination node specified, altering the Mount Context if so indicated. Its arguments are the destination node and a Mount Context flag. The destination node must be the same or compatible type as the current execution type. Its value may also be a special indicator to do a load leveled migrate automatically, or it may be a special indicator to do a load-level migrate automatically and ensure that it does not remain on the current node, if possible.

SIGMIGRATE is a new signal. The default behavior is to attempt to migrate the signaled process or process group to the specified node. The node is specified via the argument of kill3. If no argument is specified then the destination node is the execution node of the kill3 system call that sent the signal. The SIGMIGRATE signal delivery does not interrupt executing system calls so EINTR will never be returned.

Note: if the migrate fails the process continues on its original node without impact.

Kill3 is a new system call. Kill3 is similar to kill but permits an additional 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.

The time accounting information is managed as follows:

1. If the process remains on the local node, the information is unaltered.
2. If the process moves, an accounting record is written on the local node. Then the process's user and system times are added to the process's accumulated child user and system times, and the process's user and system times are reset.

This mechanism is provided to provide accurate accounting records of resources used on each node. This is important because time on differing machines may be very different in cost. Adding the time to the child user and system time is suboptimal. However it appears better than discarding it and another approach would require changes to the base.

4.1.2.2 CELLWIDE PROCESS NAMESPACE

This function requires no command or programming interface. There are, however, interfaces in the INSTALLATION, PACKAGING AND ADMINISTRATION SPECIFICATION (TBD).

4.1.2.3 VIRTUAL PROCESSES

No specific command or programming interface changes or additions are needed for this function other than augmented error codes to handle network failures.

4.1.2.4 REMOTE PROCESS CLIENT/SERVER FUNCTION

No specific command or programming interface changes or additions are needed for this function other than augmented error codes to handle network failures.

4.1.2.5 EXECUTION PERMISSION

No command or programming interface changes or additions are identified at this time. The execution permission data should be kept by the DCE Authorization Services. It may be necessary to provide the DCE Authorization Services interfaces.

4.1.2.6 REMOTE PROCESSING COMMANDS INTERFACE

There is a set of new commands to access the remote processing functions. These include onnode, fastnode, fast and migrate. Besides being available commands, some of these are also provided as shell builtins, as described in SECTION 3.1.2.6.

The onnode command allows remote execution selection to be done in two ways. A specific node may be selected, or an attribute or set of attributes can be specified, leaving the actual node selection to the on command. The -v option directs the command to display the name of the node chosen.

The fastnode command selects and then returns the node name of the least loaded node that has compatible machine attributes with the current node. The fastnode command may also broaden the selection beyond those nodes with the same attributes or it may limit the selection to nodes with a specific set of attributes. In all cases the choice is limited to nodes within the current program's Sphere Of Interest and, of course, to nodes which are up.

The fast command is similar to fastnode, except a specified command is run upon the selected node. The migrate command permits a user to migrate a process or process group to a particular node.

4.1.3 NODE STATUS SERVICE

As described in CHAPTER 3, the Node Status Service has two major pieces, the Local Node Status Service, which runs on each participating node, and the Group Node Status Service, which runs on a subset of the nodes, managing the status of a group of nodes. Within each of the Node Status Services, there are three subservices, Node Attribute Service, Node User Service and Cluster Mount Service. Below we look at the interfaces Node Attribute Service and Node User Service. The Cluster Mount Service is discussed in section 4.2. In addition, interfaces to the sphere of interest function are specified.

4.1.3.1 NODE ATTRIBUTE SERVICE

Command Interface

There are two new commands that make extensive use of the data provided by the Node Attribute Service. These are loads and node.

The loads command displays average load information about all nodes in the current program's Sphere Of Interest. It can display the load average information about the current node, a specific node, or a selection of nodes with specified attributes.

The node command is used to report on static information about nodes. This information is obtained from the Local Node Status Service. The command line options and the 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.

An administrative function to allow tuning the load delta for retransmission of load information, as well as the maximum time interval for pushing information will be provided. This will be specified in the

ADMINISTRATION SECTION (TBD).

Programming Interface

There are interfaces to get and set the local machine's attribute list and there is an interface to cause the Local Node Status Service Kernel Extension to check and possibly update its cache of attribute data with respect to a given set of nodes.

The setmachattr system call sets the local machine's machine attribute list. This system call can only be used by a privileged process. No validation is done by this system call on the contents of the attribute list, except making sure that the components are separated by slashes and the list is null terminated.

```
setmachattr(attrlist)
char *attrlist;
/* ptr to the input machine attr list */
```

The getmachattr system call places the local machine's machine attribute list in the provided output buffer.

```
getmachattr(attrlist, len)
char *attrlist;  /* address of buffer for output attr list */
int len;         /* length of output buffer, in bytes */
```

The cachattrdata call indicates to the Local Node Status Service Kernel Extension that attribute data about certain sites should be checked to see if it is current and if it is not current, the Group Node Status Service should be queried to ensure current data locally. The cmd argument allows the caller to

request either a specific site, specific sites, or to use the process's Sphere of Interest.

```
cacheattrdata(cmd, params)
```

There will also be library routines to interrogate the Local Node Attribute Service data, specifically to select an appropriate node to execute on.

4.1.3.2 NODE USER SERVICE

Command Interface

This service has data provided to it by several commands and is interrogated by other commands. Commands which modify the local /etc/utmp file have been changed to also notify the Node Status Service of the changes. This includes the commands login, init, rlogind, and telnetd.

The comsatx command differs from the standard comsat command in that it notifies the user's login sessions on all nodes within the comsatx command's Sphere Of Interest that mail has arrived.

The fingerx command differs from the standard finger command in that it lists the idle time and login time for a particular user name for all nodes within the current user's Sphere Of Interest on which the user is logged in. The option '-L' allows only the current nodes login sessions to be listed.

The talkx command is a modification to the standard talk command. When no node name is given, all nodes within the current user's Sphere Of Interest will be searched. An additional nodename parameter can be used with the line parameter. The line parameter, when issued WITHOUT the nodename parameter, causes the search for the user being contacted to be done on device line within all nodes in the current user's Sphere Of Interest. When the line parameter is issued WITH the nodename parameter, only the specified node will be searched. In the case where neither the nodename parameter nor a node is specified, if the user has multiple logins, and one is the current user's node, then that login will be notified. If none of the logins are the current user's node, the most recently accessed node will be notified.

The usersx command differs from the users command such that if no options are given all login names of users logged in on nodes within the current user's Sphere Of Interest will be listed. If this form of the command is used, the name of the node on which each user is logged in is prepended to the output of each line. The node. The '-n' option allows the user to specify a node. In this case only users on the specified node will be listed.

The wallx command differs from wall by sending the specified message to all users logged in on all nodes within the current user's Sphere Of Interest. The '-L' option has been added to send only to user's logged onto the current user's node.

The whox command differs from the who command by displaying information about users logged in at all nodes within the current user's Sphere Of Interest. The node name is listed along with the user information. A '-L' option has been added to list only those users on the current node.

The writex command differs from the write command in the same manner in which the talkx command differs from the talk command.

Programming Interface

Updating signed on user data will be done via an enhanced version of the setutent library routines. For display commands (e.g. usersx, whox, talkx, comstatx, fingerx, writex, wallx, etc), there are two interfaces which force the Local Node User Service to have timely data which can be interrogated. They are:

```
CACHEUSERDATA(CMD,PARAMS)
```

The cmd argument can specify either a specific node, a Sphere Of Interest, or indicate that information about all nodes serviced by the Group Node User Service.

```
GETUSERDATA(USERNAME)
```

This interface ensures that data about the user, within the calling program's Sphere Of Interest, is cached for interrogation.

New functions will be added to interrogate Local Node User data, to be used instead of the getutent library routines. The getutent library routines will still work as they do now, accessing only the local signed on user data.

4.1.3.3 SPHERE OF INTEREST

The sphere of interest is used for remote execution load leveling as well as for limiting the search scope for queries about logged in users, mounted file systems and possibly other resources.

Command Interface

The sphere command allows the user to add or delete nodes or groups to/from the sphere of interest (SOI) and to display the current SOI.

For the convenience of the user or administrator, a Sphere of Interest may be specified by referring to "groups" of nodes. Groups are defined by files containing the list of nodes. They may be defined either by the system administrator or by the user. The sphere command built into the shells would manipulate the SOI. A separate command is also provided which changes the sphere and then execs $SHELL (this could be used from user supplied shells which don't have a built-in sphere commands). The sphere command would allow nodes or groups of nodes to be added or subtracted from the SOI as well as allowing the entire list to be redefined.

```
sphere                  # print SOI
```

```
sphere [node|group] ....   # reset entire SOI
sphere [+|-][node|group] # modify existing SOI
```

The default for the Sphere Of Interest is set by the administrator.

The who and loads commands would pass the SOI to the Node Status Server to limit the response to information concerning the specified nodes.

The mountx and dfx commands will pass the SOI to the Cluster Mount Server to limit the response to information concerning file systems on the specified nodes.

The fast and fastnode commands default to choosing from among the intersection of the set of nodes of the "same type," Xperm, and SOI.

Programming Interface

The getsphere and setsphere system calls would manipulate the SOI. The user process setting the SOI is responsible for expanding the group lists. These system calls take pointers to the expanded lists.

```
result = setsphere(char **nodelist, int nnodes)
(result == 0 for success, -1 for failure)
result =  getsphere(char **nodelist, int nnodes)
```

(on success, "result" is the number of entries in "nodelist", but no more than "nnodes" entries will be returned)

In all of the RPCs which send spheres of interest (except for Soi_to_UUID), the sphere of interest is passed as unique identifier (UUID) representing the SOI rather than passing the entire list. The Local Node Status Service and Group Node Status Service will cache the mapping of these UUIDs t the corresponding SOI. If the LNSS or GNSS does not recognize a given UUID, it can request the full list by sending a Get_Soi request to the site which presented the UUID.

```
Get_Soi(UUID)
```

This request is used to get the full SOI list for a SOI known only by its UUID. It can be 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 padded 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 GNSS when a request is sent to it with the UUID of an unknown SOI.

Sphere of interest identifiers are obtained by registering a sphere of interest with the GNSS.

```
Soi_To_UUID(SOI)
```

This request is sent to the GNSS to obtain a UUID for a SOI. The GNSS returns either a new UUID or one obtained from its SOI cache.

4.2 THE FUSION CLUSTERING CAPABILITY

The clustering capability has interfaces for data clustering a process load leveling.

4.2.1 CLUSTERING OF DATA

Included in this section are interfaces to join and leave a cluster, mount interfaces and any NFS coherency related interfaces.

Command Interface

The clusterjoin command will issue a clusterop system call to join a specific cluster.

The clusterleave command will issue a clusterop system call to take the local node out of the cluster. The mountx command is very similar to the standard mount command but is extended in several ways. First, a display option of -L is added to permit local only display. If that option is no used, the semantics are to display all mounts in the Sphere Of Interest. For non-display options, there is mount hinting, an option to specify a file system id, and an option to specify scoping of the mount (see the mountx programming interface below).

The umountx command is an extended version of the umount command. Added is the capability to force unmounts, using the umountx system call described below.

The dfx command will be modified to contact the CMS for mount information.

The NFS coherency function requires no command changes or additions other than extensions to mountx used to aid NFS system administration.

Programming Interface

The clusterop system call extension is used both to join a cluster, which involves contacting the GCMS and negotiating mount information, and leaving a cluster, which involves telling the GCMS and then cleaning up local mount information.

The mountx system call, used by the mountx program above, has extensions to allow mount hints, assignment of file system ids and flags to indicate the scope of the mount (this is relevant when multiple root file systems are involved; the scope can be either restricted to the local machine or clusterwide). The umountx system call, used by the umountx program above, has an extension to allow forced unmounts. While this is a generally useful kernel extension, it is included here because remaing quiescent so as to allow unmounting in the clustered environment could be very difficult.

The semantics of the standard mount and umount system calls are extended to negotiate with the CMS in the case where the node is a cluster member.

If DCE does not provide something, an interface may be needed to allow determination of which node or nodes a given file is stored.

No programming interface changes or additions are needed to complement the NFS coherency functionality.

4.2.2 CLUSTER LOAD BALANCING (INVOCATION)

Many of the basic load balancing primitives were described in the previous section. This section will identify those remaining interfaces.

A load module marked for exec time load leveling will have its execution node selected automatically by the system.

During the exec operation, the load module object type is examined and compared with the acceptable local values. If the object is not compatible with the local machine, **FUSION** will attempt to execute the load module on an appropriate node in the cluster.

Command Interface

A command will be provided to mark executables to be automatically load leveled. Some of the fields that may be set include suitability for load leveling, don't preserve mount context when load leveling, I/O intensive, CPU intensive, and memory intensive.

Programming Interface

The are load level options on rexec and rfork.

4.2.3 CLUSTER LOAD BALANCING (MIGRATION)

Command Interface

There will be a sample load leveling daemon which will load level by migrating processes between compatible architectures.

Programming Interface

There are load level options on migrate.

4.3 FILE SYSTEM REPLICATION INTERFACES

Command Interface

The fs whereis command is used to determine where in the file system a file is stored. This DCE file system command will be modified to accommodate replicated file systems.

There will be administrative interfaces to create and manage replicated file systems. Included will be the capability to convert a read/write replica to read/only and convert a read/only to read/write.

Programming Interface

There will be a library interface to retrieve information on where a file is stored.

5. INTERFACE TO OTHER PRODUCTSS

The overall goal of the **FUSION** software is to provide a portable transparent distributed operating environment on top of Unix systems with OSF/DCE. In order to be portable, intrusions into the existing unix base must be minimal, and those that are required must be done in a clean, well defined way. Below is the outline of the interfaces and hooks needed in Unix and those related to DCE.

5.1 INTERFACE TO UNIX

Where possible, **FUSION** components will exist outside of the standard Unix base. Standard interfaces to the systems will be used to the degree possible. When extensions to the kernel are required, they will be made using well defined hooks.

Below are the interfaces and required changes in the areas of installation, administration, system utilities, base kernel and NFS extension.

5.1.1 INTERFACE TO INSTALLATION

There are several pieces of setup which must be done at the time the **FUSION** extension is installed. At installation time, a primary process id (pid) range must be assigned to the node. This will be done in a transparent manner.

At installation time enhanced versions of login, init, telnetd and rlogind must be installed so signed on user information can be gathered cellwide.

At installation time, the node must be assigned to a group for collection of node status information. This will be done transparently, at least until the environment is larger than a couple of dozen nodes.

At installation time the Local Node Status Service must be registered and set to start execution during single- to-multi.

At installation time the installer is given the option of having each foox command replace the corresponding foo command in the base.

5.1.2 INTERFACE TO ADMINISTRATION

Several administrative capabilities are needed to support **FUSION**.

It will be possible to request additional process id (pid) ranges and to relinquish any no longer needed, except the primary range.

It will be possible to rearrange the group (for node status information) membership and the preferred Group Node Status Service nodes.

It will be possible to determine, programmatically, the active Group Node Status Server for any given node, so as to obtain status information about that node.

It will be possible to tune the reporting of node status information.

It will be possible to register cluster membership and to join or leave a cluster.

It will be possible to specify the mount characteristics desired for particular file systems.

It will be possible to create replicated file systems using any physical file system layout.

It will be possible to monitor and tune replicated file system activity.

It will be possible to convert read/write replicas to read/only replicas and to convert read/only replicas to read/write replicas.

5.1.3 INTERFACE TO SYSTEM UTILITIES

The **FUSION** function depends on very few changes to base system utilities. As outlined above in section 4.1.3.2, only commands which modify /etc/utmp need be augmented and thus overlay the base versions. In addition to updating /etc/utmp, those commands will also update the Local Node Status Service component (it may be the case that no system utility code need change and the added code can be in a library). In AIX these changes may be placed in the setpcred system routine.

As was described in chapter 4, enhanced versions of a dozen or so utilities are provided to allow viewing a collection of machines as a single machine. The standard version will function, however, on the local node.

5.1.4 INTERFACE TO BASE OPERATING SYSTEM

The **FUSION** function is largely a set of installable kernel extensions. However, a few hooks and a small bit of restructuring is important to allow the extension to function.

Without having completed the detailed design and exhaustive analysis of various Unix kernels, it is difficult to lay out a complete list of changes needed to those kernels. Below is an attempt to capture all the changes needed, organized by the function which requires the change.

Vprocs

Analogous to vnodes, the proc structure is split into a vproc structure and the regular proc structure. The vproc structure has identification fields and relationship pointers. The physical proc structure is largely unchanged. The changes needed to support and use the vproc concept are less than 1024 lines of code, involving small changes to 20 to 30 routines that access the proc structure currently.

Process Id (PID) Allocation

FUSION introduces the idea of cellwide process ids. Each machine can use the range 0 --> 128K for special processes (like, init, pager, swapper, kprocs, etc.)

but most processes should be assigned from the one or more pid pools assigned to that node. The name service should keep track of the ranges assigned to each machine but that machine can also cache that information across reboots as it is static data. There is no implication that the ranges should be given out in a sequential manner.

EXEC

A hook or two are needed in the exec strain of code to allow certain marked load modules to be load leveled. A bit (at least) is needed in the executable header to be used as an indication that the load module can/should be load leveled.

File Offset Tokens

The struct file should be extended to at least point to a a token structure, which is used via some extension code to maintain a single logical fileblock between processes sharing that fileblock across multiple machines. In addition to the structure enhancement, each place a fileblock is interrogated (maybe 10 places) needs two lines of hooks to call the token management routines for the object.

File Reopen and Lock Movement

New struct file operations (fo_reopen and fo_prep_export) and new vnode operation (v_reopen and v_prep_export) are needed to allow a remote exec to keep its files open and any locks intact. The vnode changes should be part of DCE+. Support for the extensions is not complicated and not physical file system dependent.

Read/Write Replication

At this point it looks like all the necessary hooks will be in the DCE distributed file system code and not in the base. However, only early versions of DCE distributed file system have been researched. In case the DCE does not provide the necessary file versioning capability, it is described here.

To correctly assure that two copies of a file labeled the same have the same contents, each "version" of a file must be uniquely marked. A "version" is any snapshot of the file that can be replicated. Without atomic updates via shadow paging (as was done in TCF), versioning must be very carefully done to avoid the possibility that two replicas think they have the same contents when indeed they don't. Versioning can be done using the ctime inode field with the following conditions and hooks:

1. time must never go backwards;
2. there is an interface to force an inode write with a specified ctime;
3. ctime of a one second granularity may be unacceptable

If condition 1 cannot be met, a new commit count mechanism should be added that is guaranteed to not decrease.

Pipes and Fifos

Some small restructuring of the pipe storage code may be necessary to allow the code to handle both local and remote requesters. Special bookkeeping will be done using a new structure pointed to by the vnode in the pipe VFS.

Sockets

Remote socket support requires that alternate system call entry points be linked for the socket related system calls. Ioctl, close, stat, and select are intercepted with substitute file operations in the file ops table.

Remote Devices

The user structure should be extended to describe remote controlling ttys.

The incore inode should be extended to add an rdevnode field, which could be a pointer to a host table entry. Additionally some small changes may be necessary to initialize the vnode operations correctly to allow operations like read,write,ioctl to go to the device node while chmod,chown,etc. go to the inode storage node.

The /dev/tty and /dev/null devices may need extensions but this could be done by just replacing them in the kernel extension.

Select

There may not need to be any changes to provide select on remote objects because select is done largely through the fo-select file block operation, which, if the object is remote, will be a kernel extension routine. A small hook may be necessary to set this up, however.

Clustermount

The mount and umount system calls need hooks to call the kernel extension to negotiate the mount or umount and to inform any other nodes necessary. In addition, lookups will have to recognize mount points which do not have local mount table entries. Finally, the kernel extension needs a way to set up vnodes with mounted-on bits.

The entire set of base kernel changes is projected to be less than 2000 lines of code, with vprocs (already built) being the major piece.

5.1.5 INTERFACE TO NFS

To provide the correct coherency of NFS mounts within a cluster, minor modifications to a node's NFS support is required. This modification involves adding a token mechanism similar to that provide in DCE distributed file system, with some differences. The addition of the token mechanism should

not require any changes to the NFS code. A small change to allow ".." evaluation to work correctly outside the cluster may be needed.

Secure NFS requirements may mean that remote NFS user logins be shared between cluster nodes. Accomplishing this should not require changes to the base NFS code, however.

5.2 INTERFACE TO DCE

FUSION will be built using OSF's DCE as a base operating environment. DCE provides the basic mechanisms for inter-machine communication with NCS, directory services and naming, remote distributed file systems, and authentication services with Kerberos. **FUSION** will interface with each of these components of DCE.

5.2.1 INTERFACE TO NCS

All inter-machine communication will be accomplished using NCS remote procedure calls, or RPCs. NCS provides mechanisms for in-kernel RPCs and application level RPCs. No non-standard interfaces to NCS are required. However, it is assumed that the in-kernel NCS can communicate with the application level NCS.

5.2.2 INTERFACE TO DIRECTORY AND NAME SERVICES

It is planned that several pieces of information will be stored in the directory service but there are no plans to request changes or additions to this services.

5.2.3 INTERFACE TO DCE DISTRIBUTED FILE SYSTEM

DCE provides the basic mechanisms for remote file access and limited read/only file replication. **FUSION** will use DCE as a basis for all file related activity, including cluster wide mount visibility and file replication.

There are three areas where modifications to DCE distributed file system code are required. One set is to support reopening of files and movement of file locks, needed for remote execution and process migration. Another set is to support the single cluster wide view of file system mounts. The final set is to support tight file system replication. Some of these changes are required in the initial release of DCE, not just in **FUSION** provided versions. However, the changes will be developed as part of **FUSION**.

File Reopen and Lock Movement

To support remote process execution and process migration, hooks are required to permit an opened file to be reopened on another node. This must be supported even when multiple opens are not normally permitted. In addition, file locks must be able to move with remote execution.

Cluster Wide View of File System Mounts

Three small changes are needed in DCE to allow the support of a cluster wide view of mounted file systems to operate successfully. The objective is that,

within a cluster, all file system mounts will be visible to all other cluster nodes. First, a way is needed to generate volume or file system id numbers without registering them with the VLDB. Since the file system id is 64 bits, some range reservation for this purpose seems practical. Second, there must be a way to add entries to the cache manager's volume or file system table. Lastly, a small changes must be made to the protocol exporter so that it will pass on "mounted-on" information, so it can be inspected during lookups within the cluster.

Read/Write Replication

The read/write replication service depends on almost all the DCE distributed file system mechanism. In particular, it depends on the documented features of the VLDB to support large numbers of replicas. It also depends on the ability to switch flow of lookup from the "glue" code to the cache manager and back and in fact it depends on being able to switch an already open file from local to remote service and from remote service back to local. If any of these dependencies are not in the base DCE code, additional code will be needed to provide the service.

Several hooks are needed to fully support read/write replication. They include:

File System Location Data Base (VLDB):

Support is needed for a new file system type, which should be transparent to the VLDB other than the type indication itself. In addition, an added field may be needed for the file system data structure and this field would need to be sent from the VLDB to the client code.

Client Code (Cache Manager)

Most of the changed needed for read/write replication are in the client code. The first change would be to allow base DCE code access to read/write replicas, even if locally the **FUSION** code was not installed. The proposal is, under this condition, to just access the read/write replica as if the file system was not replicated at all. The second change would be to have hooks so that if the **FUSION** code is installed, full access to the replicated file system would be supported. To do this, the volume or file system registry may need additional fields to support the new file system type. Also, the storage node must be put in cache manager part of the inode so different files in the same file system can be serviced by different storage nodes.

Token Processing

The plan is that each call to the token manager will be intercepted by the replication token manager, which will only analyze requests on read/write replicated file systems. All requests will eventually be passed to the regular DCE token manager.

The replication token manager may, however, pass back a new return code which indicates the request should be retried at the node storing the read/write replica. The cache manager and glue code need a hook to check for this return code, passing control to the kernel extension when this happens. None of the requested changes are of any significant size and none pose any risk or performance penalty to the overall DCE.

5.2.4 INTERFACE TO DCE SECURITY SERVICES

Kerberos is a network authentication subsystem used by all DCE RPC services. It operates using a ticket- granting mechanism. Passwd.etc is a related service to provide authorizations for logging in. These services will also be used by **FUSION** services, including remote processing and remote device kernel extensions.

There is one known area where Kerberos tickets and **FUSION** remote execution need to coordinate. Kerberos has the concept of a ticket-granting ticket, which is sort of a supper ticket and is used to obtain specific tickets for access to services. Clearly a remote process execution or a process migration would have to either take this ticket along with it or be able to reobtain it. Our understanding is that to reobtain it, the user would have to "login" again, which of course is unacceptable. Thus we must be able to copy the ticket granting ticket to another node. At present it is not clear if this will require any changes to the kerberos code.

6. ERROR HANDLING

Error handling will be provided as a basic extension to the existing Unix error handling mechanisms. In most cases no extension will be necessary. The areas to be addressed are at the kernel level, at the application level, and at the remote system level.

6.1 ERRORS DETECTED BY THE KERNEL

Errors detected by the kernel are indicated through a few well defined interfaces. The most common errors are indicated via return codes from system calls. Some abnormal conditions are indicated via special signals, but these are usually somewhat hardware dependent."

Some kernel errors are indicated via console log entries or messages directed to users."

Most additional errors detected by the kernel will be surfaced via additional return codes. The following error return indications will be added.

```
ESITEDN1   Required node is not available
ESITEDN2   Operation terminated due to node failure
ENOSTORE   File or working directory is unavailable
ENLDEV          Not a local device.
EBADST          Bad node specification
ELDWRG          Load module not for this machine
ELOCALONLY      Operation restricted to local node
```

The supporting perror(S) library routine will also be enhanced to display reasonable strings for those errors. In addition to this list, some detected errors will be indicated using already existing error indication codes.

No special generation of signals for error indications will be introduced as part of this functionality. The addition of console log or special user messages will be kept to a minimum. Additionally, new system panics will be kept minimal as well. 6.2 APPLICATION LEVEL ERROR HANDLING Application errors are usually indicated via an error message from the program, Occasionally, application programs indicate errors via the system error log facilities. Some applications experience errors and exit suddenly, perhaps with a core dump file.

It is expected that most new error indications at the application level will be made via error messages. This is the most portable form of error handling, and the one most widely used. For some critical system applications, the ability to log errors via a system logging mechanism would be extremely desirable.

Messages of this type will be entered into the log via the syslogd mechanisms, which incorporate a time stamp with the message.

For this system, applications are expected not to exit mysteriously or produce a core dump.

6.2 REMOTE SYSTEM ERROR HANDLING

Remote system error handling will be handled much as local system errors are handled. Indications of problems at remote systems will typically be obtained from the RPC layer. In some cases, the higher level protocols may detect an error. Some errors pertaining to security will be detected by the Kerberos mechanism.

Errors detected by the RPC layer will typically be errors where the other node fails to respond in a timely or appropriate way. The most common of these is the case where the remote node has failed in the middle of an operation.

Errors detected by higher level protocols will be indicative of state inconsistencies between the two nodes. For example, two nodes may have different notions of the present state of a process, which would be detected at this level.

The DCE Security Mechanisms mechanisms will detect errors that involve user authorization to perform functions or obtain services from a particular system. Examples of such a failure would be a user who did not have correct permission attempting to read or write a copy of a replicated file.

The nature of reporting errors pertaining to remote systems will be the same as reporting local errors to the user, as described above in SECTION 6.1 ERRORS DETECTED BY THE KERNEL and 6.2 APPLICATION LEVEL ERROR HANDLING.

6.3 NETWORK PARTITIONING ERRORS

At some point during normal system operations the system may experience failures in the underlying communications. This may be a network hardware failure, a routing failure, a failure of a gateway device, or one of several other causes.

A principal goal of **FUSION** design is to minimize the impact of such failures. At most, the failure is to be restricted to the nodes that are providing resources and nodes that are using those resources. When such a failure occurs, the failure is to be limited to the failure of particular operations on those nodes, and not to the nodes in general. The design is to be resilient to intermittent partitioning errors as well as longer term problems. An important goal is to avoid long, chaotic recoveries.

6.4 DEBUGGER SUPPORT

Some extensions to the existing dbx interface to support debugging of processes on different nodes are required.

Changes to the crash command are required to operate with the restructuring of the kernel into vprocs.

7. PERFORMANCE

There are three areas of performance to be considered. These are the performance impact of the **FUSION** hooks on existing Unix software, the performance impact of the **FUSION** hooks on the DCE software and the performance impact of the **FUSION** software itself.

7.1 FUSION HOOKS IN THE BASE OS

The changes made to the base Unix to support **FUSION** shall not result in performance degradations for existing Unix services when compared to the same system running without the **FUSION** modifications. The goal is to preserve the execution path for the strictly local case; nothing will be added except a simple test to determine if the operation is, indeed, strictly local.

7.2 FUSION HOOKS IN DCE

The changes made to OSF/DCE to support **FUSION** shall not result in performance degradations for existing OSF/DCE services when compared to the same system running without the **FUSION** modifications.

7.3 FUSION CODE IMPACT

Performance of the new software is critical. For a distributed system solution to offer true transparency, Performance Transparency is an important goal. If obtaining remote services takes substantially longer or consumes substantially more system resources, then users would prefer to not utilize the remote resources.

Without DCE performance numbers, however, it is premature to specify performance requirements.

8. IMPACT OF FUSION ON EXISTING CUSTOMERS

The addition of the **FUSION** software will not affect any existing binaries or source, either syntactically, semantically or with respect to performance.

APPENDIX A: RATIOS, RELATIONSHIPS TO DCE, AND ALTERNATIVES TO DCE

This appendix provides a discussion of parameters and ratios of services within the **FUSION** system, the relationship of the **FUSION** components to DCE, and some alternatives to DCE.

RATIOS

In order for the **FUSION** design to scale, several hierarchical structures have been created. This section is provided to discuss the ratios that are assumed within these structures.

The concept of Sphere of Interest has been introduced in **FUSION** to permit various commands and other activities to restrict their concern to a user controllable set of nodes that they are interest in, because it is relevant to them. For example, the 'who' command examines the sphere of interest and only identifies the users on the nodes within that sphere. For typical users, it is expected that the sphere of interest would be less than fifty nodes within the cell. However, there is not limit enforced by the design.

In order to manage node status information, the concept of node groups was introduced. A group of nodes will share a single group server (though that server is assigned in a dynamic way). This group server acts as a repository for information about the nodes in that group. When a node requires information about another node, it inquires the group server. If the node in question is within the group the information is returned right away. If it is within another group, then this group server contacts that node's group server. It is expected that the groups will typically be less than 100 nodes. Furthermore, group membership should be defined in a logical way, such that the groups are not strictly arbitrary. For example, at a University the Electrical Engineering Department might be a group, and the Mechanical Engineering Department might be another group. It is hoped that within a typical group, most users of the nodes in that group will have sphere of interests that are mostly if not completely contained within the group. This will minimize much of the inter-group network traffic.

The Group Servers interact closely with the name service and the network security server. It is expected that in a typical installation, there would be one group server per name server (maybe even the same node).

A **FUSION** Cluster could potentially contain as many as 4096 nodes, although most clusters will probably be much smaller than that. Clusters are conceived as not spanning a DCE cell.

The concept of a cluster and a group are only marginally related. For administrative reasons, however, it might be convenient that if members of a group are members of some cluster that the be members of the same cluster.

Since clusters can be larger than groups, it may be made up of several groups.

The replication design for scaling supports a hierarchical propagation model. There is a single read/write copy from which all changes originate. The hierarchy is to define some of the replicas as principal replicas, which propagate directly from the read/write copy. Secondary copies then propagate from the principal replicas. The rule of thumb for allocating principals and secondaries is to have the read/write copy serving as many principals as each principal serves secondaries. For the pathological case of 111 replicas, one would be the read write replica, which would serve ten principals, each of which would serve 10 secondaries. While these ideal ratios will not always be achievable, this is the target model.

Another issue affecting all of the above services is network location of critical resources within the cell. In a complex network topology with multiple networks, gateways, and even slow point-to-point links, a reasonable configuration goal is to centrally locate resources critical to the system. For example, group servers and read/write copies of file systems should be roughly equally distanced from all remote nodes. It would be a poor idea from a performance perspective to put these resources at the far end of a remote link.

RELATIONSHIPS TO DCE

The **FUSION** product as defined within this specification is built upon DCE. This section describes the particular aspects of DCE that are required.

All of the **FUSION** functions that require communication between nodes use the NCS RPC mechanisms to obtain that communication. Consequently, NCS is required for all aspects of **FUSION.**

Several of the **FUSION** functions use the services of the DCE Distributed File System. These services are remote processing (for file reopen support), the Node Status Service, the Cluster Mount Service, the clustered NFS service, and the **FUSION** File System Replication.

The Node Status Service uses the DCE Name Service and Directory Service.

The security of **FUSION** is to be based upon the security services of DCE. This includes the authentication server and the authorization services. Also, some security is achieved by using security features of NCS. It is possible for **FUSION** to run without these services, but the security features will be absent.

ALTERNATIVES TO DCE

As specified, the **FUSION** requires DCE as an underlying mechanism. This is beneficial because DCE is an industry standard. However the **FUSION** architecture is general enough to not require DCE. The system could be specified to use alternate underlying services.

The use of NCS could be supplanted by some other RPC mechanism. NCS is nice because easily handles some of the heterogeneity issues, such as byte ordering and data formats. However, additional software such as that provided in TCF could be substituted, to be used with an alternate RPC mechanism. With additional restructuring, **FUSION** could be built using a message passing protocol instead.

The DFS portion of DCE could be replaced with other distributed file system services, and provide functionality. This would perhaps be at a reduced level of functionality, depending on what was chosen. Additional software (such as a token manager) might have to be built, which is not necessary with DFS.

Another name service could be substituted. This should not pose any difficult technical problems.

The DCE Security services could have substitute services provided or built as required. Or, as stated in the previous section, could be eliminated if security concerns were not an issue.

APPENDIX B: OPEN ISSUES

At the design review and from individual comments, a set of open issues with respect to **FUSION** has been identified. These issues need to be discussed and resolved before the **FUSION** functionality is fully defined. The remaining issues are on extensions and other unspecified functionality that may be considered now or at some time in the future.

This appendix lists those issues in detail.

1. The general **FUSION** security model needs to be resolved, both in terms of the DEE as well as inside the cluster. The issues to be resolved are listed here. Security issues are listed here because that area is perhaps the most significant to be addressed. The second set of items are comments pertaining to specified functionality, in the order they were presented in the body of the specification.

    a. Authentication should be done by Kerberos. This is consistent with the DCE Security Mode. The authorization should be managed as part of the DCE Authorization Services. The services from DCE are not well understood at this time. Perhaps an alternate mechanism should be investigated.

    b. There is an open issue with Kerberos, with regard to whether users want restrictions on where their Ticket Granting Ticket (TGT) can be moved.

    c. More investigation of Kerberos is required to understand if we have to move tickets other than the TGT.

    d. The issue of what authorizations are required to perform mounts that are visible to the cluster needs to be resolved. The current approach is limited to a single super-user per cluster model.

    e. At the review the issue was raised as to the possibility of creating a finer grained privileged user model, where an intermediate level of privilege would allow a particular user (or set of users) to have effectively root authority on their particular node, but that authority would not extend beyond the node. This needs to be resolved, and if the answer is yes, the function needs to be defined a design proposed.

    f. Programs that are marked setuid create a unique set of problems in a distributed environment. If multi-granular privileges are created, it becomes even more complex. The DCE model for this also needs to be understood, because globally visible setuid binaries could be set up even without the option of remote execution.

g. The issue of network security has been raised several times. Rather than use a piecemeal approach, a unified approach to network security needs to be addressed. This should include encryption and authentication at or below the RPC layer.

2. Sockets and Streams currently have limited support.

   a. Some concern was addressed regarding the performance of double hop socket support. Investigations into the X-windows use of TCP/IP indicate that the double hop should not be a big performance impact.

   b. Additional concerns were address regarding the impact of double hop socket support on availability. The initial node of the socket is fixed with the current design. Redesigning sockets to support redirection would require some significant amount of study and experimentation. There are also issues of what the failure modes would be with respect to applications that were not prepared for the remote endpoint to be relocated. This is probably more significant with network protocol domains than with UNIX-domain sockets, particularly in a cluster where the view of the file system is maintained.

   c. The potential to support streams is sketchy at best, and further design and specification work has been deferred. It may be appropriate to reconsider the priorities of remote streams support in **FUSION**.

3. There are some specific issues pertaining to process migration that require final resolution.

   a. It was suggested that having migrate take a pid argument would be useful functionality. While this could be implemented trivially by having this routine send SIGMIGRATE with the appropriate argument to the process specified, the goal was to provide a call that would not return until the migration had completed, and would indicate whether the migration had succeeded. The initial investigations indicated that this would be very difficult to build in a reasonable way.

   b. The current plan for specifying what process can migrate to which nodes is based on static attributes of the node where it is running. Some dynamic constraints have been identified, such has a 370 vector process with a fixed section size. This could perhaps be resolved by supporting some form of process attribute, which is set when particular features are used. The mechanisms to build this are not understood.

4. There are some outstanding issues regarding the Node Status Service.

   a. The current set of statistics kept at the Node Status Service are only loosely defined. It is doubtful this will be enough. What is being proposed needs to be reviewed. A mechanism to easily expand the information stored there is also desirable.

   b. Bruce indicates the algorithms need to be enhanced to allow "group: to be bigger than three nodes. Is there really a problem this serious?

5. There are a few issues open pertaining to the cluster mount function.

   a. It was suggested that making mounts persistent for the Directory and NFS mount cases would be better than providing both types of mounts. However this must still work in the case of partitioned operation. One possibility is to pick the volume ID based on the mount string so the same mount would use the same volume ID when partitioned.. Any such algorithm would need to work in heterogeneous environment.

   b. It is conceivable at an abstract level that multi-ported disks could be used in some interesting way to keep a file system mount available even though a failure occurred. This requires additional study to propose a mechanism for doing this. The semantics under these circumstances would also have to be defined (such as what would happen to files open for modification at the time of the failure)

   c. The significance of linking with shared libraries and the effect on mount context decisions needs to be investigated. This is particularly relevant if the base executable comes from a different mount context than the one the process runs in. The shared library mount context probably needs to be the same one as where the executable came from.

   d. There has been a suggestion made that clusters are not that meaningful in the expanded context of the DEE. It might be more reasonable to have users define their own cluster by allowing SOI to control load leveling.

   e. Supporting TCP/IP operations to a cluster was expressed as a desirable goal, having the name server do load leveling. The work on this has been deferred up until now.

   f. There is some concern that the interaction between mounted-on vnodes and DFS tokens might be a problem. This should be investigated.

6. The NFS function is fairly well understood. There is one issue remaining involving interaction with DFS support for NFS.

   a. Should we do NFS the way DFS is planning? The current scheme proposed would likely perform better but may be more complex.

7. The **FUSION** function includes support for automatic load leveling at execution time and while processes are running. There are a couple of open issues remaining in this area.

   a. The plan for **FUSION** is to supply a sample load leveling daemon. However little **FUSION** work has actually be done to provide reasonable load leveling algorithms. Perhaps some work from a previous LCC R&D project is applicable.

   b. A set of load leveling information access library routines are described. This description needs to be fleshed out to a specification level of detail.

8. More of the review comments pertained to the **FUSION** replication function than any other single area. Replication has been taken out of the DEE layer since the other **FUSION** functions do not strictly depend on it. There are still several issues for discussion.

   a. Jim has suggested a variation of replicated file systems that are used primarily as a backup mode. This needs to be thought about further, and if agreed as a proposal, the functionality and interfaces need to be described.

   b. It is conceivable at an abstract level that multi-ported disks could be used in some interesting way to keep the read/write copy of a replicated file system available even though a failure occurred. This requires additional study to propose a mechanism for doing this. The semantics under these circumstances would also have to be defined (such as what would happen to files open for modification at the time of the failure)

   c. An alternative to the multi-ported disk that requires no special hardware is to provide a mechanism to convert a read-only copy to the read/write copy when the original read/write copy becomes unavailable, and to convert the previous read/write copy to a read-only copy when it returns. There is a potentially difficult problem here that could result in two writable copies if a network partition occurred. The resolution is no less difficult than supporting multiple read/write copies in the first place.

   d. The replication portion of **FUSION** has been moved to a separate layer in the architecture. There was some discussion that maybe it ought to be a stand-alone product. This is probably not a big

technical issue, but an agreed position is required.

e. When a process is reading from a file in a replicated file system and the read/write copy goes away, if there is no up to date read-only copy, the current specification says that the read will still switch to one of the read-only copies. There is a suggestion that this is not the correct thing to do, or at least it should be a user controlled policy.

f. The current replication version information is designed to be based on the inode ctime. There is some concern that the time granularity is not good enough, that time going backwards would be a problem with this design, and that ctime may not be updated for all file modifications. These need to be resolved, perhaps by proposing an alternate file version mechanism to initiate propagation.

g. The current design for file propagation is to read the whole file. This can result in unacceptable propagation times when small changes are made to large files. Some form of page level or range oriented propagation might be desirable.

h. In the current design, a file with unallocated blocks on the read/write node will propagate as files with zero-filled blocks in the other copies.

i. If a binary in a replicated file system is replaced, the current specification is not clear as to how propagating the new version affects current running versions, particularly with respect to demand paging of load modules. Returning ETXTBSY is undesirable as it would restrict replacement of programs like 'init' and 'shells' which tend to always be running. Keeping shadowed versions may not be supportable in all underlying physical file systems. One interesting point to investigate here is what DFS does in its replication scheme. One possibility would be building a shadow mechanism in the kernel extension, using the same mechanism that unlinking a running binary uses.

j. Some comments expressed doubts about the semantics for loosely replicated file systems relating to file access and read locks. Agreement is required on the suitability of the current loose replication design.

k. If an executable is replaced in a loosely replicated file system and then 'execed', the user would probably expect to get the new version. The current semantics do not guarantee this. A suggestion was made to provide a mechanism to insure it had propagated. It is not clear exactly how this would work.

l. The names of .replinfo and .frfsinfo files in the root directory of a replicated file system could cause name space collisions, which would be a minor deviation from UNIX semantics. This solution or some other one needs to be agreed upon.

m. Much speculation was made as to just how well the **FUSION** replication designs will perform and to how many copies they will scale. We need to propose some simulations to model these designs and get a better handle on this.

n. A feature was added to TCF after its initial release to cause propagations to occur synchronously. Such a feature might be useful in **FUSION** replication. This needs to be decided, and if it is to be added, the functionality and design need to be specified.

o. At some point multiple read/write copies of a replicated file system should be supported. This requires substantial design work.

p. It was noted at the review that access times are managed on a per-copy basis. Some believed that it might be good to keep these synchronized, though the cost of doing so would likely be prohibitive.

9. In order for **FUSION** to support multiple versions and releases simultaneously, support for more general expandability needs to be addressed. Expandability

   a. Critical Protocols and visible global data structures need to supply version information. Also a mechanism to guarantee upward compatibility and interoperability is needed.

   b. Some features of **FUSION** may be optional in the first release. Additional features might be added later. A mechanism to support negotiation of common function between participating nodes is probably required in this complex environment.

   c. The current design does not support hidden directories but still support heterogeneity. Without hidden directories, the execution search path could be much more complicated and searching it could become inefficient. The best way to do this within the constraints needs to be determined.

10. One of the goals of **FUSION** is availability in the face of failure. Another is robustness in non-transitive and non-symmetric situations. The current failure modes need further investigation with respect to these goals. In some cases changes may be required.

    a. There are places where a non-transitivity failure mode could cause significant problems.

b. Similarly, non-symmetric transient problems could result in inconsistent results.

c. Methods to support clean shutdown of processors or machines with minimal impact to ongoing computations need to be provided.

11. Some of the services of DCE may limit overall performance.

    a. A mechanism to support parallel concurrent RPCs from a single thread is highly desirable. An alternative of using multiple threads to do them in parallel might offer a satisfactory substitute approach.

12. Several extensions to **FUSION** might be appropriate now or in the future.

    a. Supporting a pmake program in the **FUSION** might be an added selling point. Such a program can take good advantage of the **FUSION** environment to improve performance significantly.

    b. The current RPC mechanisms is synchronous in nature. TCF experience has shown that asynchronous write operations may perform better, particularly for devices, sockets, and pipes. Such an approach might prove desirable in **FUSION** though it may require parallel RPC mechanisms or using greater numbers of threads to build it.

    c. Support of POSIX style Asynchronous I/O is going to be important at some point for some vendors. **FUSION** will need to provide this in the distribute environment it provides.

    d. The ps interface maybe should be improved to avoid reads from kernel memory. Other programs that read from kernel memory could also be enhanced in this way.

    e. SVR4 provides the /proc interface. Support for remote /proc might be appropriate when porting **FUSION** to the SVR4 environment.

    f. At some point the **FUSION** protocols should be published in order to support implementations on other platforms, such as Windows 3.0. This needs additional study.