Inter-Office Memorandum

To	Distribution	Date	11 July 77
From	Jim White	Location	Palo Alto
Subject	**Data Structure Transmission Protocol (Version 2)**	Organization	SDD/SD/CS

XEROX

Filed on: <White>DSTP2.Ears

Preface

This is the fourth in a series of memos [1, 2, 3] aimed at identifying new Mesa facilities for simplifying the construction of distributed systems. This memo obsoletes its predecessors by defining Version 2 of a *Data Structure Transmission Protocol (DSTP)* that facilitates the transmission of Mesa data structures between hosts. This second iteration provides support for strings, array descriptors, and pointers, and incorporates background and motivational material from previous memos.

Thanks are due Will Crowther and Hal Murray, who made themselves available for numerous discussions, and Jay Israel and Chuck Geschke, who contributed valuable information on the use of similar techniques within Juniper and Sil.

Outline

The present memo is organized along the following lines:

General Goals

A variety of network communication facilities are currently available within the Mesa programming environment. Depending upon his requirements for generality and efficiency, the programmer can select from among a number of subroutine packages, each providing a different grade of service (e.g. raw packet mode, packet stream mode, byte stream mode, and file transfer mode).

While providing an important spectrum of communication services, all of the existing facilities require the programmer to manipulate remote objects in ways that are different and typically more cumbersome than those used to manipulate local objects of the same type. Local program modules, for example, are accessible by procedure call; to communicate with a remote module, however, the programmer must construct (say) a packet stream, encode and transmit a service request to the remote module, and await and decode the remote module's reply. To read a local file, the programmer need only invoke the appropriate file system primitives; to read a remote file, on the other hand, he must first transport the file to his local machine using the File Transfer Package.

Some of the problems associated with providing uniform access to objects in a distributed system, especially problems in the area of distributed file systems, are already being tackled in the Mesa environment. A Page Level Access Protocol, for example, is being developed as a vehicle for more direct manipulation of remote files. The Mesa compiler is also being augmented to automatically retrieve from a remote file system any XM files it needs which aren't available locally.

The present memo addresses the uniform access problem in a more general way and describes a new Mesa facility that greatly simplifies the construction of distributed systems generally, rather than just aiding in the construction of one particular distributed service (e.g. a distributed file system). The author's ultimate aim is to extend the application domain of the Mesa programming system itself to include distributed programs, and so to reduce the cost of building and maintaining distributed systems.

Specific Goals

Underlying every distributed service is a protocol that specifies, among other things, the content and format of messages exchanged between the local program that requires the service and the remote program that provides it. Although the interface between consumer and supplier is thus ultimately message oriented, it is usually advantageous to interpose an additional piece of software local to the consumer that provides him with a *procedural interface* to the remote service. Doing so both localizes knowledge of the protocol (which may change from time to time) and makes the service easier and more natural to use.

One important task of the module that implements the procedural interface is to prepare messages for transmission to the remote system and interpret the messages it receives in response. To accomplish this task, the module must convert outgoing data structures (e.g. those that originate as arguments to its public procedures) from their standard *internal* representation, dictated by the Mesa compiler, to their standard *external* representation, dictated by the protocol. Similarly, it must convert incoming data structures (e.g. those destined to be returned as results of its public procedures) from their external representation to their internal one. Because the required formats for even the most common data types (such as integers and character strings) vary from one protocol to another, specialized conversion code must be written for each distributed service.

As an initial step toward simplifying the construction of distributed systems in the Mesa environment, this memo defines a standard, service-independent, *Data Structure Transmission Protocol (DSTP)* that facilitates the transmission of Mesa data structures between hosts by specifying a standard external representation for each Mesa data type

(including, of course, integers and character strings). DSTP is thus a first step toward extending the domain of the Mesa type system to include network protocols. DSTP's two immediate results are, first, *to reduce the size of programs that use it by enabling them to share a single internal-external conversion package*, and second, *to elevate higher-level* client *protocols that use DSTP above the level of detailed message formats to the more abstract level of Mesa data structures.* While both results are important, the second has profound implications for the protocol design process. Since DSTP specifies (once and for all) the precise encoding of each data type (e.g. INTEGER, STRING, ARRAY, RECORD), the designers of higher-level protocols are relieved of such chores and need only specify the abstract structure of the messages required by their applications. The syntactic descriptions of client protocols are, in fact, Mesa definitions modules (rather than prose or BNF) which can be compiled and used by software that implements the protocols.

Scope of the Protocol

DSTP provides a means for transmitting Mesa data structures between hosts. The term *data structure* here denotes the value of any data object whose definition is permitted by the Mesa type system, encompassing objects of all types (both predefined and constructed) and uses (constants, variables, procedure arguments, procedure results).

DSTP is concerned with the *format* of transmitted data, rather than with the mechanism of transmission. DSTP therefore specifies an encoding for transmitted data structures but does not legislate a transmission mode (or a rendezvous technique or, of course, the higher-level semantics of the data structures transmitted). For applications that use the communication system's packet-level transmission facilities, DSTP requires that the *text* field of each packet contain the encoding of an integral number of data structures and that no extraneous data be present. For applications that use the communication system's byte-stream transmission facilities, DSTP requires that the byte stream contain the encoding of an integral number of data structures and, again, that no extraneous data be present.

Most client protocols will enforce a one-to-one correspondence between data structures and commands to or requests of the remote host. In the general case, a service request will be implemented as a record whose components are the parameters of the request.

Specification of the Protocol

The current specification of the protocol assumes no modifications to the Mesa compiler. Where compiler modifications would significantly enhance the protocol and its use, that fact is indicated in small type (like this).

DSTP defines a network encoding for Mesa data structures transmitted between hosts, consisting of a fixed-length *header* component followed by variable-length *value* and *relocation vector* components:

Header (3 words):
  Type (1 word)
  Value Size (1 word)
  Vector Size (1 word)
Value (*value size* words):
  Primary data structure (length unspecified)
  Secondary data structure #1 (length unspecified)
  Secondary data structure #2 (length unspecified)

... *etc.*
Relocation Vector (*vector size* **words**):
Value offset to link to secondary data structure #1 (1 word)
Value offset to link to secondary data structure #2 (1 word)
... *etc.*

As he reads the remainder of this protocol specification, the reader is invited to consult, as an example, the following Mesa data structure and its DSTP encoding. The size (in words) of each element of the encoding is given as a subscript--$\langle element \rangle_{size}$--following the value of the element, which is enclosed in angle brackets:

Mesa Declaration:

```
dataStructure: RECORD [
  integer: INTEGER=3,
  POINTER TO RECORD [
    boolean: BOOLEAN=TRUE,
    string: STRING="example"]];

dataStructureType: TYPE = {dataStructure};
```

DSTP Encoding:

Header (3 words): $\langle$dataStructure$\rangle_1\langle 10\rangle_1\langle 2\rangle_1$
Value (10 words):
Primary (RECORD[INTEGER,POINTER]): $\langle 3\rangle_1\langle 2\rangle_1$
Secondary (RECORD[BOOLEAN,STRING]): $\langle 1\rangle_1\langle 4\rangle_1$
Secondary (string structure): $\langle 7\rangle_1\langle 7\rangle_1$ $\langle$"example"$\rangle_4$
Relocation Vector (2 words): $\langle 1\rangle_1\langle 3\rangle_1$

Header

The header records the size (in words) of each of the two variable-length components that follow it. It also identifies the *type* of the data structure by means of a code that associates the transmitted value with either a predefined type or a constructed type located in a definitions module to which both sender and receiver have access. Future specifications of the protocol might (with the compiler's help) include in the header a date-time-place stamp associated with the definitions module, thereby permitting verification that sender and receiver were compiled with the same version of that module. In the current specification of the protocol, the type code found in the header is supplied and interpreted by the sending and receiving applications programs, respectively (an enumerated type is suggested but not required as the means by which the code is specified). Because the type code is only interpretable by the applications programs, the receiver must use a loophole to assign the transmitted value to a Mesa variable. A better approach, one that preserves the integrity of the Mesa type system but which also requires compiler support, would be that in which the type code is generated (at the sending end) and checked (at the receiving end) by the compiler.

Value

The value component of the data structure's encoding contains a copy of the compiler's own internal (untyped) encoding of the *primary* data structure and any *secondary* data structures (as defined below) which the sender elects to transmit. Since the data structure's internal and external representations are identical, no conversion code or processing overhead is required by either sender or receiver.

The address and size (in words) of a data structure named *structure* of type *Structure* are given in Mesa by *@structure* and *SIZE[Structure]*, respectively; by definition, the primary data structure is the data described by these parameters. Immediately following the primary data structure (in arbitrary order) are zero or more secondary data structures to which the

primary data structure or another secondary data structure makes reference. (The term *composite data structure* denotes the primary data structure and all of its transmitted secondary data structures.)

Three classes of secondary data structures are currently defined: (1) If the composite data structure contains a STRING, the string structure it addresses (containing the string's LENGTH, MAXLENGTH, and TEXT) may be transmitted as a secondary data structure. (All *MAXLENGTH* characters are transmitted, regardless of the value of LENGTH.) The address and size (in words) of the string structure for a string named *string* are given in Mesa by *LOOPHOLE[string, POINTER]* and *2+(string.maxlength+1)/2*, respectively. (2) If the composite data structure contains an ARRAY DESCRIPTOR (which specifies the BASE and LENGTH of the array), the array itself may be transmitted as a secondary data structure. The address and size (in words) of an array of elements of type *type* denoted by an array descriptor named *descriptor* are given in Mesa by *BASE[descriptor]* and *LENGTH[descriptor]*SIZE[type]*, respectively. (3) Finally, if the composite data structure contains a POINTER, its referent may be transmitted as a secondary data structure. The address and size (in words) of the referent of type *Referent* of a pointer named *pointer* are given in Mesa by *pointer* and *SIZE[Referent]*, respectively.

Relocation vector

Each secondary data structure is addressed from somewhere within the composite data structure. Strings and pointers are nothing more then *links* to their respective string structures and referents, while the link to an array is embedded in an array descriptor as its BASE attribute.

In its passage from sender to receiver, each link between the composite data structure and a secondary data structure undergoes two successive transformations. Beginning as an address within the sending host, the link is converted to a *word offset within the value component* before the data structure is transmitted via the network. Once the data structure has been delivered to the receiver and positioned in its memory, the link is converted to an address within the receiving host.

The third and final component of the data structure encoding, the relocation vector, enumerates the links the receiver must convert from offsets to addresses to obtain a legitimate Mesa data structure. Each word of the relocation vector contains the word offset within the value component of a string, array descriptor, or pointer whose string structure, array, or referent was transmitted as a secondary data structure. To instantiate all of a data structure's secondary data structures, the receiver need only execute the code:

```
FOR i IN iVector DO
  siteOfLink ← vector[i] + baseOfValueComponent;
  siteOfLink↑ ← siteOfLink↑ + baseOfValueComponent;
  ENDLOOP;
```

Implications of the Protocol

The data structure encoding described above always achieves the intuitively correct results for types BOOLEAN, CARDINAL, ENUMERATION, INTEGER, SUBRANGE, CHARACTER, ARRAY (with element type chosen from this same set), and RECORD (with component types chosen from this same set). Such data structures are always transmitted by value and no secondary data structures are involved.

The sender may transmit a data structure of type STRING, ARRAY DESCRIPTOR, or POINTER with (i.e. *attached* to) or without (i.e. *detached* from) its associated string structure, array, or referent. While detached strings and array descriptors are rarely useful, the protocol

permits their use. Client protocol designers, however, should use them only with an understanding of their limitations and of the possible pitfalls involved in their use. Detached pointers, on the other hand, are often useful, as in the case where the pointer serves as a handle to an object maintained by the sender and the handle is always returned to its source for evaluation. The compiler could help here by providing primitives (for encoding and decoding data structures for transmission) that interpret strings, array descriptors, and pointers as attached or detached data structures according to directives embedded in the data structure's declaration.

Data structures of type PROCEDURE are admitted by the protocol, but must always be returned to their source for evaluation. In subsequent versions of the protocol, the body of a procedure could *conceivably* (in some cases) accompany the PROCEDURE as a secondary data structure.

Data structures of type SIGNAL and PORT are unsupported by the protocol and should not be transmitted. (The transmission of SIGNALS is pointless since signals are implemented as addresses within the sending host). The compiler could help here by providing primitives (for encoding and decoding data structures for transmission) that prohibit (or at least warn of) the use of these prohibited data types.

Specification of the Package

Described below is a Data Structure Transmission Package (DSTP) that implements the protocol. The Mesa source code for the definitions and program modules can be found in [4] and [5], respectively. The implementation described here is 434 bytes (less than one page) of code and requires a 12-word global frame.

Principal Data Structures

DSTP maintains all of the state information it requires in a client-supplied data structure of type *Ds*. All of the components of the *state record* addressed by *Ds* except the first, *dsHeader*, must be initialized by the client before any DSTP procedures referencing *Ds* are called. (The client may also modify the state record between data structures if it so desires.) Once initialized, a single state record can be used to send and receive an arbitrary number of data structures in succession:

```
Ds: TYPE = POINTER TO DsObject;
DsObject: TYPE = RECORD [
  dsHeader: DsHeader,
  dsSegmentQueue: DsSegmentQueue,
  dsRelocationVector: DsRelocationVector,
  dsSendBlock: SendBlock,
  dsReceiveWholeBlock: ReceiveWholeBlock,
  dsSendReceiveParameter: UNSPECIFIED];
```

The state record contains a workarea *dsHeader* in which the data structure's header component is constructed, a descriptor *dsSegmentQueue* for a client-supplied array in which such information as the address and size of primary and secondary data structures is recorded for later transmission, and a descriptor *dsRelocationVector* for a client-supplied array in which the data structure's relocation vector component is constructed:

```
DsHeader: TYPE = RECORD [
  dsType: INTEGER,
  dsSegmentQueueSize: INTEGER,
  dsRelocationVectorSize: INTEGER];
DsSegmentQueue: TYPE = DESCRIPTOR FOR ARRAY OF DsSegment;
DsSegment: TYPE = RECORD [
  dsSegmentReferencePointer: POINTER,
  dsSegmentPointer: POINTER,
  dsSegmentSize: INTEGER,
  dsSegmentOffset: INTEGER];
DsRelocationVector: TYPE = DESCRIPTOR FOR ARRAY OF WORD;
```

The state record also identifies client-supplied procedures (*SendBlock* and *ReceiveWholeBlock*) that DSTP may call to send and receive blocks of outgoing and incoming data structures, and an arbitrary parameter *dsSendReceiveParameter* (e.g. a byte stream handle) to be passed to those procedures by DSTP. Requiring the client to supply such procedures allows a single DSTP package to be used in conjunction with the communication system's various (e.g. packet-level and byte stream) transmission facilities:

```
SendBlock: TYPE = PROCEDURE [parameter: UNSPECIFIED,
blockPointer: POINTER, blockSize: INTEGER, blockIdentity: BlockIdentity];
  ReceiveWholeBlock: TYPE = PROCEDURE [parameter: UNSPECIFIED, blockPointer: POINTER,
blockSize: INTEGER, blockIdentity: BlockIdentity] RETURNS [blockPointer: POINTER];

BlockIdentity: TYPE = {firstOfDs, interiorOfDs, lastOfDs};
```

The *blockPointer* passed to *ReceiveWholeBlock* may be NIL, in which case *ReceiveWholeBlock* is expected to provide the necessary storage.

Public Procedures

DSTP provides five procedures for sending data structures, only two of which need be called to send data structures that contain neither strings, array descriptors, nor pointers (i.e. that have no secondary data structures):

```
EnqueueDsForSend: PROCEDURE [ds: Ds,
dsType: UNSPECIFIED, dsPointer: POINTER, dsSize: INTEGER];
  SendDs: PROCEDURE [ds: Ds];
```

The first procedure, *EnqueueDsForSend*, records in the state record the type, address, and size (in words) of the (primary) data structure to be sent. The second procedure, *SendDs*, actually sends the data structure, calling the client-supplied *SendBlock* procedure indicated in the state record. The data structure is not copied by *EnqueueDsForSend* but rather is passed directly to *SendBlock* by *SendDs*.

To send data structures containing (attached) strings, array descriptors, or pointers, calls to one or more of the following additional procedures must be interposed between the calls to *EnqueueDsForSend* and *SendDs* (strings, arrays descriptors, or pointers which the client fails to identify in this fashion will be sent as detached data structures):

```
NoteStringInDs: PROCEDURE [ds: Ds, stringSite: POINTER TO STRING];
  NoteArrayDescriptorInDs: PROCEDURE [ds: Ds,
arrayDescriptorSite: POINTER--TO DESCRIPTOR FOR ARRAY--, arrayElementSize: INTEGER];
  NotePointerInDs: PROCEDURE [ds: Ds, pointerSite: POINTER TO POINTER, referentSize:
INTEGER];
```

A call to *NoteStringInDs, NoteArrayDescriptorInDs, or NotePointerInDs* makes known to DSTP the existence of a string, array descriptor, or pointer within the composite data structure and causes the corresponding string structure, array, or referent to be transmitted when *SendDs* is invoked. These procedures simply note (in the client-supplied segment queue) the address and size (in words) of the secondary data structure, which *SendDs* later passes directly to *SendBlock* (no copying is performed). (*SendDs* must modify the composite data structure to convert embedded links (to secondary data structures) from addresses to offsets before delivery to *SendBlock*, but restores the data structure to its original state before returning to the client.)

DSTP provides a single procedure for receiving data structures:

```
ReceiveDs: PROCEDURE [ds: Ds]
```

```
    RETURNS [dsType: UNSPECIFIED, dsPointer: POINTER, dsSize: INTEGER];
```

ReceiveDs returns the type, address, and size (in words) of the received (composite) data structure. Space for the composite data structure is provided by *ReceiveWholeBlock*; proper disposal of this storage is therefore the responsibility of the client. When *ReceiveDs* returns, links to any string structures, arrays, or referents will have been converted from offsets to addresses within the client's host.

Conclusions

The Mesa type system provides powerful facilities for controlling the exchange of procedure arguments and results between programs on a single machine. DSTP brings these same facilities to bear upon the exchange of messages by distant programs linked by a network.

DSTP elevates higher-level client protocols from the level of detailed message formats to the more abstract level of Mesa data structures. The syntactic description of a DSTP-based protocol is itself Mesa source code (a definitions module) that can be compiled and used by programs that implement the protocol.

DSTP is basically a discovery (that Mesa objects can be transmitted via the network) rather than an invention; less than a page of code is required for its use. DSTP reduces the amount of communications software required within the system by enabling diverse applications to share a single, much simpler software package for converting data structures between their internal (Mesa) and external (protocol) representations.

Appendix A: Byte Stream Procedures

Given below are the definitions and program modules (available also as [6, 7]) for the byte stream implementation of the *SendBlock* and *ReceiveWholeBlock* procedures required by the Data Structure Transmission Package. The implementation given here is 80 bytes of code and requires an 11-word global frame.

Definitions Module

```
DIRECTORY
  PupDefs: FROM "PupDefs",
  DSTPDefs: FROM "DSTPDefs";
DEFINITIONS FROM PupDefs, DSTPDefs;

BsDSTPDefs: DEFINITIONS =
  BEGIN

  -- BsDSTP procedures
  BsSendBlock: PROCEDURE [bs: ByteStream, blockPointer: POINTER, blockSize: INTEGER, blockIdentity:
BlockIdentity];
  BsReceiveWholeBlock: PROCEDURE [bs: ByteStream, blockPointer: POINTER, blockSize: INTEGER,
blockIdentity: BlockIdentity] RETURNS [blockPointer: POINTER];

  END.
```

Program Module

```
DIRECTORY
  PupDefs: FROM "PupDefs",
  DSTPDefs: FROM "DSTPDefs",
  BsDSTPDefs: FROM "BsDSTPDefs";
DEFINITIONS FROM PupDefs, DSTPDefs, BsDSTPDefs;

BsDSTP: PROGRAM [PupFacilities: PupInterface] IMPLEMENTING BsDSTPDefs =
  BEGIN OPEN PupFacilities;

  -- BsDSTP public procedures
  BsSendBlock: PUBLIC PROCEDURE [bs: ByteStream, blockPointer: POINTER, blockSize: INTEGER,
blockIdentity: BlockIdentity] =
    BEGIN
    blockFinger: BlockFinger;
    blockFinger ← DESCRIPTOR[blockPointer, blockSize, WORD];
    ByteStreamPutBlock[bs, @blockFinger];
    IF blockIdentity = lastOfDs THEN ByteStreamSendNow[bs];
    END;

  BsReceiveWholeBlock: PUBLIC PROCEDURE [bs: ByteStream, blockPointer: POINTER, blockSize:
INTEGER, blockIdentity: BlockIdentity] RETURNS [blockPointer: POINTER] =
    BEGIN
    blockFinger: BlockFinger;
    IF blockPointer = NIL THEN blockPointer ← AllocateHeapNode[blockSize];
    blockFinger ← DESCRIPTOR[blockPointer, blockSize, WORD];
    ByteStreamGetWholeBlock[bs, @blockFinger];
    END;

END.
```

Appendix B: A Slice of a Client Protocol and its Implementation

Given below are the definitions and program modules for the implementation of a small subset of a hypothetical, DSTP-based, Page-Level Access Protocol (PLAP). Note particularly the procedure *OpenRemoteFile*, which demonstrates the use of the DSTP Package. Note also that the syntactic descriptions of PLAP messages are Mesa declarations.

Definitions Module

```
PLAPDefs: DEFINITIONS =
  BEGIN

  -- PLAP data types
  OpenMode: TYPE = {read, write, append, readWrite};
  Handle: TYPE = CARDINAL;

  -- PLAP signals
  NonExistentFile: ERROR [file: STRING];
  -- etc.

  -- PLAP procedures
  OpenRemoteFile: PROCEDURE [bs: ByteStream, file: STRING, openMode: OpenMode] RETURNS [handle:
CARDINAL];
  -- etc.

  -- PLAP command and response types
  CommandType: TYPE = {openFile}; -- etc.
  ResponseType: TYPE = {successful, noSuchFile}; -- etc.

  -- PLAP commands and responses
  OpenFileCommand: TYPE = RECORD [file: STRING, openMode: OpenMode];
  OpenFileResponse: TYPE = Handle;
  -- etc.

  END.
```

Program Module

```
DIRECTORY
  PupDefs: FROM "PupDefs",
  DSTPDefs: FROM "DSTPDefs",
  BsDSTPDefs: FROM "BsDSTPDefs",
  PLAPDefs: FROM "PLAPDefs";
DEFINITIONS FROM PupDefs, DSTPDefs, BsDSTPDefs, PLAPDefs;

PLAP: PROGRAM [PupFacilities: PupInterface, bs: ByteStream] =
  BEGIN OPEN PupFacilities;

  -- PLAP signals
  NonExistentFile: ERROR [file: STRING] = CODE;

  -- PLAP global variables
  sendBlock: PROCEDURE [UNSPECIFIED, POINTER, INTEGER, BlockIdentity] ← BsSendBlock;
  receiveWholeBlock: PROCEDURE [UNSPECIFIED, POINTER, INTEGER, BlockIdentity] RETURNS [POINTER]
← BsReceiveWholeBlock;
  ds: Ds;
  dsSegmentQueue: DsSegmentQueue = DESCRIPTOR[segmentQueue];
  segmentQueue: ARRAY [0..2) OF DsSegment;
  dsRelocationVector: DsRelocationVector = DESCRIPTOR[relocationVector];
  relocationVector: ARRAY [0..2) OF WORD;
  handle: Handle;

  -- PLAP procedures
  OpenRemoteFile: PROCEDURE [bs: ByteStream, file: STRING, openMode: OpenMode] RETURNS [handle:
CARDINAL] =
    BEGIN
    -- declarations
```

```
  commandType: CommandType ← openFile;
  responseType: ResponseType;
  openFileCommand: OpenFileCommand ← [file, openMode];
  openFileResponse: POINTER TO OpenFileResponse;
  -- send open file command
  EnqueueDsForSend[ds, commandType, @openFileCommand, SIZE[OpenFileCommand]];
  NoteStringInDs[ds, @openFileCommand.file];
  SendDs[ds];
  -- receive open file response
  [dsType: responseType, dsPointer: openFileResponse] ← ReceiveDs[ds];
  handle ← openFileResponse↑;
  FreeHeapNode[openFileResponse];
  -- interpret response
  SELECT responseType FROM
    = successful => RETURN;
    = noSuchFile => ERROR NonExistentFile[file];
    ENDCASE => ERROR;
  END;

--etc.

-- PLAP main program
ds.dsSegmentQueue ← dsSegmentQueue;
ds.dsRelocationVector ← dsRelocationVector;
ds.dsSendBlock ← sendBlock;
ds.dsReceiveWholeBlock ← receiveWholeBlock;
ds.dsSendReceiveParameter ← bs;

END.
```

References

[1] White, J.E., Mesa Data Type Transmission Protocol, 8 June 77, Filed on <White>DTTP.Ears.

[2] White, J.E., An Example of DTTP's Use By Higher-Level Protocols, 10 June 77, Filed on <White>DTTPExample.Ears.

[3] White, J.E., Data Structure Transmission Protocol, 24 June 77, Filed on <White>DSTP.Ears.

[4] DSTP Definitions Module, Filed on <White>DSTPDefs.Mesa.

[5] DSTP Program Module, Filed on <White>DSTP.Mesa.

[6] BsDSTP Definitions Module, Filed on <White>BsDSTPDefs.Mesa.

[7] BsDSTP Program Module, Filed on <White>BsDSTP.Mesa.

Distribution:

SDD/SD/CS
David Boggs
Chuck Geschke
Charles Irby
Jay Israel
Butler Lampson
Hugh Lauer
Bill Lynch
Paul McJones
Dave Redell
Ed Satterthwaite
John Shoch
Wendell Shultz
Dan Stottlemyre
Ed Taft
Smokey Wallace
John Weaver
Ben Wegbreit
John Wick