M.I.T. Laboratory for Computer Science

Request for Comments No. 272
February 5, 1985

Unified Stream Protocol

by David D. Clark

1. Introduction

1.1. Goals

This documents describes the Unified Stream Protocol (USP). USP is an end-to-end transport layer protocol. In common with other transport protocols, it has the goal of providing a defined semantics upon which applications may construct communication services. However, USP differs rather markedly from most transport protocols. Most importantly, USP is designed to facilitate interoperation between machines implementing different protocol families. That is, USP is intended to facilitate protocol conversion.

To achieve this goal, USP is envisioned as a "veneer" which is implemented on top of whatever "native mode" transport protocol is available to it. USP hides the specific details of the native transport protocol, providing instead a single unified interface, which applications may use independent of which protocol family is actually implemented on the machine.

If USP is to achieve its goal of effective operation across a number of protocol families, it must realize a number of subgoals. First, the functionality supplied by USP to its clients must be useful to those clients. Second, the functionality supplied by USP must not require an inefficient use of the underlying native transport protocol. Third, USP must be easy to implement. The details of USP have been carefully chosen so as to meet these goals as much as possible.

The functionality supplied by USP to its clients is usually called a "virtual circuit" semantics. This is simply defined as reliable bi-directional byte stream. In other words, if one application asks USP to transport a number of bytes to the other end of the point-to-point connection, those bytes are delivered, in the order in which they were sent, without loss or duplication. If the bytes cannot be delivered, because of disruption within the communication system, both ends of the connection are notified that the connection is now broken.

Almost all protocol families contain a layer which provides approximately this function. However, a detailed examination of various protocols reveal that there can be great variation in the functions actually available to the client. It is this variation which the USP is intended to correct.

1.2. Functionality

USP is a block structure protocol. A block is a ordered sequence of bytes, of unrestricted length. Along with each block is transported a sixteen bit type field, the purpose of which is discussed below. The blocks are delivered in the order in which they are sent and the bytes within each block are delivered in the order in which they are sent.

Many protocol families implement a virtual circuit protocol with this sort of block structure. It appears that this semantics is helpful in the construction of application packages. However, the block structure was chosen for USP for a much more important reason: the block structure is critical for the operation of USP itself. As part of connection establishment and error recovery, USP itself must exchange information between the two ends of the connection. Since this information must be sent down the same data stream that is used for the transport of client information, some block structure is required in order that USP distinguish these two sorts of data. Also, there are certain data items which the client wishes to transport, in particular addresses of USP end points, which must be transformed by USP as they are transported through the system. (This problem is discussed in detail in the section on addressing below.) USP must be able to distinguish these addresses in the data stream, in order to translate them. To distinguish between blocks which contain USP information, blocks which contain client information, and client information which USP must manipulate, the type field which is associated with each block is used. There are a small number of type fields reserved for USP, a small number of type fields reserved for special address information, and the remainder of the type fields are reserved for arbitrary use by the client of USP.

For those blocks which USP transports on behalf of the client, there is no restriction on the content of the bytes which composed the block. However, for those blocks which USP and the client share, there must be standard representation of the data types, so that USP can manipulate them reliably. Therefore, USP also includes a definition of the representation of standard data types. And it is recommended that the client use these representations as well, in those cases where this implies no undue hardship. The motivation for this is that USP can supply, as part of its implementation, a standard set of library routines which translates the standard representation into the native representation for the particular machine. Therefore, for common data types, such as character strings, integers, and booleans, the effort required to implement a client module could be substantially reduced if the standard USP data translation library was used. However, this is not a requirement, except for those data records which USP interprets. Except for those records, USP can be used to transport arbitrary byte strings, with no restrictions of any sort on the content of the bytes.

1.3. Ease of Implementation

It is intended that USP can be implemented on top of any protocol family which provides a transport protocol with the functionality generally described as virtual circuit. The various features of USP are carefully selected to make it easy to implement on a variety of existing or proposed virtual circuit protocols. In particular, certain features been have omitted from USP, such as out-of-band signaling, because they substantially complicate the realization of USP.

In addition, certain decisions about USP have been made, taking into account the manner in which most protocols are implemented in today's operating systems. In particular, most protocol families provide for a logical sharing of the physical network resource among a number of virtual circuits. This demultiplexing of the incoming data into a number of virtual circuits is normally done inside the supervisor of the operating system, or at least in a protected process which cannot be easily modified by the applications programmer. USP has been carefully designed to avoid any need for further demultiplexing of a single virtual circuit among a number of clients. This permits USP to be implemented as a set of subroutines which run in the process of the application. Thus, USP can be implemented without modifications to existing network code, and without inserting additional functionality into protected portions of the existing operating system. For many existing protocol implementations, it would not be possible to implement USP at all if system modifications were required to support it.

Some of these points are discussed in more detail in the sections that follow.

1.4. Structure of This Document

The next section of this document defines in detail the functionality of the USP protocol. The sections that follow discuss specific aspects of USP, including protocol conversion, the naming of connection end points, reliability, etc. In addition to the actual specifics of USP, this document attempts to capture some of the motivation for design decisions in USP. Some of this information is discussed in later sections of the document, some is included within the body of the document at the point at which design decisions are introduced. To distinguish this information from the actual specification, it is printed with indented margins. After the document itself will be found a number of appendices which give the specific realization of USP on top of a number of common transport layer protocols.

2. Block Transport

2.1. Introduction

USP is defined as two layers. The bottom layer, described in this section, defines the reliable transport of blocks from sender to recipient. The next layer, described in the section following, specifies the representation of data transported in these blocks.

Level one USP provides a block transport service between two clients. Conceptually, the functions available to these clients can be represented as four subroutine calls:

```
Open_connection [foreign_client_name]
Close_connection []
Send_block [block_type, data_bytes]
Receive_block [block_type, data_type]
```

However, since blocks can be of unbounded length, it is unreasonable to demand that the client transmit the entire block to USP in one subroutine call. Therefore, the send_block function is replaced with the following:

```
Send_subblock [block_type, end_of_block_flag, data_bytes]
```

The receive call is similarly replaced by

```
Receive_subblock [block_type, end_of_block_flag, data_byte]
```

Note that the length of the block is not explicitly transmitted, therefore the sender need not know the length of the block, and the receiver may not know the length of the block until the last subblock has been transmitted. While USP reliably maintains the boundary between blocks, the boundaries between subblocks have no meaning, and USP will combine subblocks as appropriate to make efficient use of the underlying protocol. Any internal structure of the block must be represented either using level two USP, or using client specific structures.

> An alternative implementation of USP would have transported the length of the block explicitly as part of the block information, perhaps along with the block type. Obviously, this would be helpful to certain applications. For example if a large file is being transferred, it would be nice to know if there was room to receive the file before it has been transmitted. However, requiring that the length of a file be known in advance is, in many cases, an extreme inconvenience to the implementation, because the data being transported is not a literal copy of the precise bytes stored in the memory of the computer, but is instead a translation of these bytes into some canonical representation, such as EBCDIC into ASCII or internal data representations into level two USP representations. This transformation may cause the number of bytes to grow and shrink, in a manner that cannot be predicted except by translating the information. Therefore, if it is necessary to know the length of a file before transmitting any of it, it is necessary to transform all of the file before transmitting any of it. But this eliminates a very important parallelism which some implementations may wish to achieve, and in the worst case may require translating the information twice, if there is not room to store the translated version of the information along with the internal representation. In this case, it would be necessary to translate it twice, once throwing away the result but counting the bytes, and the second time actually taking the bytes as they are translated and sending them over the net. For these reasons, it appears unreasonable to require that the length of a block be known in advance. For those particular applications which wish to know this fact, the length can be transmitted as one of the data items in a client block.

3. Establishing a Connection

USP uses the parameter foreign_client_name provided as part of the open call to identify the other end of the connection to be established. The manner in which this name is translated is discussed in the section on naming, below. USP then uses whichever native transport protocol is available to establish a virtual circuit with that foreign machine and client. Once the connection is established, USP transmits one block from the initiator to the recipient of the connection. This block, discussed below, is the connection_open block, which permits each end of the connection to know the USP level name for the other end. The field is primarily important in those cases where protocol translation is being provided by USP between two native mode transport protocols. Once this block has been transmitted by USP, the client is permitted to exchange blocks as appropriate.

3.1. Data Type

Every block which is transferred from sender to receiver is tagged with a sixteen bit data type. Data types zero through three hundred are reserved for USP; their specific purposes are summarized in a table at the end of the document. Block types from 300 to 399 are reserved for client data types which USP modifies in transit. In particular, these data types are used for a variety of blocks which transport USP level client names. See the section on naming for details. Block types 400 and above are available for arbitrary use by the clients; USP places no interpretation upon them, and does not examine and modify the content of the blocks.

3.2. Closing a Connection

Once a client has called the USP entry point close_connection, no further data may be sent or received by that client. For this reason, the clients at each end of the connection must ensure that each has sent all the data it intends to the other, before calling close. The clients may perform this function in any way they choose; however two block types have been reserved to facilitate this function, in those cases where the clients have no special requirements. These block types are called "end" and "end_reply". The normal use of these block types is as follows. When one client determines that it has no further information to send, it transmits a block of type "end." The other end of the connection, on receiving this "end" block, sends any data that it requires in the other direction, and then responds with a type of "end_reply". The client sending this first "end_reply" must then dally for some period of time, perhaps 10 seconds, which is long enough for the other end of the connection to respond and terminate normally. The client first sending the "end" block will receive eventually a matching "end_reply". At this point it should promptly send a second "end_reply" in response to the first. After this, the sender of the original "end" can unilaterly terminate its connection. The receipt of this second "end_reply" by the dallying client similarly permits the other client to unilaterly close its connection.

It may occur that both clients simultaneously determine that they have nothing further to send, so that each send an "end" to the other. In this case, each should send an "end_reply" to the other and begin dallying.

> The design of a procedure for closing a USP connection involves a number of trade-offs. Closing a connection is one area in which different protocol families have substantial differences in the functionality they provide their client. Some transport protocols guarantee that before the connection is closed, each client has received all of the data which the other wishes to send. Other protocols have functionality more

resembling that chosen for USP: when one closes a connection it ceases to receive any data even if the other end continues to send.

The most efficient and reliable implementation of USP was, therefore, to assume minimum function in the underlying transport protocol, and implement the reliable close as part of the USP architecture. The choice then was whether USP should ensure that the connection was closed reliably, or whether the client should ensure that the connection was closed reliably. The design decision here was based on a desire to model most closely those protocols which USP resembles, in particular the Xerox NS Courier protocol. It is felt that by closely matching this protocol it will be easier to take advantage of it. It also appears that many clients can be naturally built in such a way that no special handshaking is required before they close the connection. That is, the natural structure of interchange between clients will enable each to know when the other has no further data to send. However, to make it easier for those clients who do not wish to think about the problems of closing a connection, USP supplies a default procedure, which could be implemented by a set of default subroutines which the USP client could use. Use of these subroutines would mean that USP has effectively provided a reliable close for the client, without any particular attention on the part of the client. This compromise seems to achieve the best of both possible solutions.

The function of the dally operation is to increase the reliability of the reliable close in the face of low level failures in the transport function. It can be shown that there is no way to guarantee in the face of low level failures that each end of the connection is correctly informed of the state of the other. Dallying in this fashion is a generally accepted strategy for recovering from delayed data while permitting the connection to clean up if the path is permanently broken.

4. USP Data Types

4.1. Introduction

The lower level of USP defines a block stream, a series of typed blocks of arbitrary length. This section describes the USP representation for data objects which may be sent in a block. There is one object transmitted per block, with a variety of constructor objects defined to permit sending aggregates of data items.

The data representations described in this section are explicitly modelled on the data types defined by the Xerox NS Courier protocol, with minor exceptions. The intention of this definition is to facilitate operation between these protocols. The reader is referred to the Xerox Courier specification for motivation for the detailed representation decisions.

USP defines seven base data types:

1. boolean
2. cardinal
3. long cardinal
4. integer
5. long integer
6. string
7. unspecified

4.2. Boolean

Boolean represents the values true or false. The standard representation of a boolean is a single bit preceded by fifteen 0 bits. The value true is encoded as 1, the value false as 0.

4.3. Cardinal

A cardinal represents an integer, N, with the range 0 <= N <= 65,535. The standard representation of a cardinal is a single sixteen bit field (2 bytes) that encodes the integer as an unsigned binary number. The most significant bits of the integer are transmitted in the first byte, most significant bit first.

4.4. Long Cardinal

The long cardinal represents an integer N in the range 0 <= N <= 4294967295. The representation of a long cardinal is a thirty two bit field (4 bytes) that encodes the integer as an unsigned binary number. The most significant bits of the integer are transmitted in the first byte, most significant bit first.

4.5. Integer

The integer represents a signed value N in the range -32768 <= N <= 32767. The integer is represented as a single sixteen bit field (2 bytes) that encodes its value as a two's complement binary number. The most significant bits of the integer are transmitted in the first byte.

4.6. Long Integer

The long integer represents the value N in the range -2147483648 <= N <= 2147483647. The representation of a long integer is a 32 bit field (4 bytes) that encodes its value as a two's complement binary number. The most significant bits of the integer are transmitted in the first byte.

4.7. String

A string represents an ordered collection of ASCII characters, whose number need not be specified until run-time. The standard representation of a string is a sixteen bit field that encodes the length of the string as a cardinal followed by the characters themselves, one character per byte. If the number of bytes is odd, the block is padded with 8 bits of 0 to make the length of the string an even number of 8 bit elements.

> The use of a cardinal to transport the length of a string limits the string to a maximum of 65,535 characters. This makes the string unsuitable for transporting very long sequences of characters. However, there is an alternative within USP, which is to transport the sequence of characters as a separate USP block. Since blocks are unbounded in length any string can be transported in this manner. Notice also that this representation of the string object requires that the length be known before any of the characters are transmitted, while, as discussed above, the block explicitly does not require this knowledge. Therefore, in many cases, the transmission of a character string as a separate block may be the more desirable representation for implementation purposes.

The ASCII sequence used for the newline function shall be the two ACSII characters CR and LF in that order.

4.8. Unspecified

An unspecified data object represents an arbitrary sixteen bit quantity whose interpretation is left to the client.

4.9. Constructed Types

USP defines four constructed types. Constructed data types are used to aggregate objects to make one larger object which can then be transported in a block. The different aggregation techniques correspond to normal programming language strategies for data aggregation. The defined constructor types are:

1. Array

2. Sequence

3. Record

4. Choice

4.10. Array

An array represents an ordered one-dimensional homogeneous collection of objects, whose type and number are statically known by the clients. The elements of an array may be of any type, including another array. The standard representation of an array is simply the standard representation of its elements.

4.11. Sequence

A sequence represents an ordered one-dimensional homogeneous collection of data objects whose type is known to the application, but whose extent is specified at run-time. The elements of a sequence may be of any type. The standard representation of a sequence is a 16 bit field, coded as a cardinal, which represents the actual number of elements in the sequence, immediately followed by the standard representation of the elements in order.

4.12. Record

A record represents an ordered, possibly heterogeneous collection of data objects, whose type and size is known statically to the client. A record is composed of components, which may of any type. The representation of a record is simply the representation of its components in order.

4.13. Choice

A choice represents a data object whose type is chosen at run-time from a set of candidate types selected by the client. The representation of a choice is a sixteen bit field coded as a cardinal, which carries a client specified type identifier for the field that follows. This selector is followed by the expected object. Note that a choice may carry an object of any type, including another choice.

> With the exception of choice, all of the USP data types are untagged. That is, it is assumed that the client knows the sequence of data types that will be transmitted as part of any object. An alternative representation of USP would have associated with each data type an explicitly transmitted tag which would describe the form of the data. This would permit a receiving program to reconstruct the content of the data stream without

any prior knowledge of the information to be transmitted. It was felt that in almost all cases the clients did know the structure of the data being represented so the inclusion of tag data would needlessly increase the bulk and complexity of the data representation. However, USP provides two escapes from this in those cases where variant data is to be transmitted. The first is the choice data type, which permits the client to include arbitrary variants within a given data object. The second option for tagged data is the lower level type field associated with each block.

5. Reserved Block Types

5.1. Introduction

USP uses a number of reserved block types for its own operation. This section describes the object which is transmitted in each of those types, as well as the function which the block type performs for USP. The reserved block types are as follows:

BLOCK TYPE	TYPE IDENTIFIER
connection open	10
connection error	20
end	254
end_reply	255
path_convert	300-399

5.2. Connection Open

Connection open block is the first block sent during connection establishment from the active end to the passive end of a USP connection. Its purpose is to carry sufficient information about the desired connection so that intermediate translation points as well as the final end point can correctly establish the desired service. The connection open object is a record with three components. The first component names the foreign host to which the connection is being established. The second component names the host from which the request originates. The third component names the particular service which is to be invoked on the foreign host. The first component, the foreign host, is represented as a choice. If the choice selector has value 0 the representation of the foreign host is a string which encodes the foreign name as a "global name". If the choice selector value is 1, the representation of the foreign name is a string which represents the foreign host as a "path_name". The second component, the local name, is also represented as a choice with the same two options as the first component. The third component, the service name, is represented as a string. This representation is summarized in Figure 1. The use of the global name and path name options are discussed in the section below on naming.

Summary of connection_open object

block type: 10
representation:
record
choice (foreign host)
0: string (global name) 1: string (path name)
choice (local host)
0: string (global name) 1: string (path name)
string (service_name)

Figure 1: Connection_Open Block

5.3. Connection Error

The connection error block is delivered as appropriate to the end points of a USP connection to indicate that the connection has failed. Connection failure is the only possible error which USP can report, and a simple client implementation can ignore the detailed information which the connection error block reports. However, the information reported in the connection error block will prove helpful in eliminating the cause of the error, in many cases.

There are two parts to the information delivered by the connection error block: the nature of the error, and what module is recording the error. The contents of the connection error block is a record with four components. The nature of the error is represented in the first two components of the record. The first component is a cardinal, which will have one of the defined error types specified in Figure 3. The second component of the record is a string, which may contain further information on the nature of the error, for human consumption.

The third and fourth components of the connection error record identify the location at which the error was detected. The third component is a cardinal which contain a code defined below which specify the type of node which discovered the error. The fourth component of the error object describes the name of the module which reported the error. It is coded as a choice with the same representation as a host name in connection open record. The format is summarized in Figure 2.

Summary of connection_error object

block type: 20
representation:
record
cardinal (what error)
string (more info)
cardinal (what reported it)
choice (name of module reporting it)
0: string (global name) 1: string (path name)

Type of nodes detecting error

foreign USP module	1
transport layer implementation	2
intermediate USP module (protocol translation point)	3

Figure 2: Connection_Error Block

Errors associated with connection establishment

foreign name unknown	10
service identifier unknown	11
host down	12
service not supported on host	13
service not currently available	14
required protocol conversion not available	15

Errors associated with established connection

transport failure (explicit)	20
transport failure (timeout)	21
foreign client failure	22
path transformation impossible	23

undefined errors	1

Figure 3: Defined Error Codes

5.4. Connection Termination

Two types are reserved to assist clients in reliably closing an USP connection. The representation of both of these is a block of 0 length. In other words, there is no object transported as part of this block, merely the block type. The two defined block types are as follows:

```
end             254
end_reply       255
```

5.5. Conversion

As discussed in the following section on naming, the representation of the host name as a path requires that the name be converted as it is transported from the sender to the receiver so that the name has meaning in the context of the receiver. This conversion is performed as necessary in the connection_open object and the connection_error object. Additionally, it is presumed that the client will, from time to time, wish to transport path names in the implementation of client functions. These names must be translated, as well as the ones used by USP. For this purpose, USP reserves a range of block types from 300 to 399. These blocks may be used by the client, but USP will examine them, and perform the appropriate conversion on path names contained within them. In order for the conversion to be done it is necessary that enough of these objects be of a known representation so that the USP module can find the path names to convert. It is assumed that the representation of any path conversion object is a record of which the first component object will be a sequence. The sequence contains strings, each of which is assumed to be a path name. After this sequence, the first component of the record, any number of additional components of any type may occur. USP will not examine or modify them.

Summary of path conversion objects

block types: 300-399
representation:
record
sequence
string (path_name)
.
.
.

6. Protocol Conversion

As has been discussed before, one of the major goals of USP is to permit conversion between different protocol families. In general, protocol conversion is impossible. Protcol conversion at the application layer sometimes works, if the two application make sufficiently similar assumptions about the functions they provide. For example, it is sometimes possible to map the remote login application in one protocol family into the remote login application in another protocol family. However, protocol translation at the transport level almost never works. There are two reasons for this failure, functional differences and naming.

Although almost every protocol family implements some form of virtual circuit protocol as an option for its transport layer, there is a wide range of functionality which can be described by this term. There are many examples of features which may or may not be included within some particular virtual circuit design: out of band signalling, synchronizing of stream after an error, partitioning of the stream into blocks, associating type information with different bytes in the stream, and assigning priorities to certain bytes. Converting between protocol families which do and do not support these features can only be achieved if the application on whose behalf the conversion is being performed does not use the feature which is not supported in certain families. This is why conversion sometimes works at the application level where this sort of information is available, but almost never works at the transport layer in general where one is performing the conversion on behalf of an unknown set of applications.

The second problem with protocol conversion is naming. Almost every protocol has its own distinct naming structure. This naming structure is in general sufficient to name all of the machines which can be reached using this protocol family, but is usually not extensible to naming machines which are available outside the bounds where this protocol is usable. Thus, almost all protocol families provide insufficient naming structure to permit a protocol converter to establish the required connections. Consider an example, in which a connection is being established from protocol family A to protocol family B using a converter which implements both of those protocols. A host in the domain served by protocol family A establishes a connection to the protocol converter for the purpose of creating a follow-on connection to some host in environment B. The naming structure available to that host was sufficient to name the protocol converter, since that machine is accessible to protocol family A.

Therefore, a connection can be established from host A to the protocol converter. But now the protocol converter must identify the host within protocol family B which is to be the ultimate destination of this connection. There is no way, using the protocol family A, that the machine in protocol family B can be named. Therefore the protocol converter cannot proceed. Again, this problem can be solved at the application level, by designing an application protocol which is sufficiently general so that, within the structure of the application, the names for machines outside the protocol boundary can be transported. Thus, again it is true that while conversion at the application level can sometimes be made to work, conversion at the transport layer, where this escape hatch for naming is not available cannot be made to work. USP has been designed to provide those features necessary so that protocol conversion can be performed at the transport layer. Using USP, one can construct a single protocol converter at the USP level, instead of constructing one converter per application, as would otherwise be required. It achieves this goal by directly attacking the two problems discussed above. USP avoids the problem of detailed functional differences by supplying its own semantics, which it enforces uniformly by an additional set of programs superimposed on the transport protocol of each individual protocol family. The naming problem is somewhat complex, and is discussed below.

USP achieves this conversion capability at substantial cost, which is that a whole new set of application packages must be implemented to run on top of USP. The existing application packages from any one protocol family are not suitable for this function, because they are not designed to run on top of the functionality supplied by USP but rather by the functionality supplied by their native transport, whatever that might be. However, this problem is not as severe as it might seem, as there are a number of existing application protocols which could be easily redefined to work on top of USP. For example, the internet mail delivery protocol, SMTP, was specifically designed to work over a minimal virtual circuit. It would work with very little modification over USP.

7. Naming

As the previous section suggests, one of the major problems in protocol conversion is the establishment of a naming strategy which permits, from inside one protocol region, the naming of entities in another protocol region. Since most existing transport protocols do not solve this problem, a solution has been integrated into USP. In fact, much of the structure of USP is related to the management of names for hosts and services.

Names within USP are character strings which are assumed to describe hosts and services. The problems associated with naming hosts and naming services are rather different; they will be discussed in turn.

7.1. Host Names

USP can be used in a variety of contexts, each of which has different naming requirements. Perhaps the simplest use of USP is when there is only one underlying transport protocol, and that transport protocol provides a standard naming strategy for hosts. In this case, USP need provide no enhancement to the existing naming strategy. However, if USP is being used in a context where there is more than one transport protocol in use, with protocol converters connecting these different regions, then it is necessary in general for USP to define a new, higher level namespace within which the hosts in each of these protocol environments can be named.

As part of USP connection set-up, this higher level name must be made available to each of the protocol converters involved in the connection, so that the protocol converter can successfully make the next lower level connection which will be used to carry the USP information. The "connection_ open" block, which is the first block sent on a USP connection, is used to transport this name.

To enable this naming mechanism to work properly, USP circuit connection proceeds in a number of stages. Initially, the originating host obtains from its higher level application the name of the foreign host to which the connection is to be made. It then translates that name within the context of the protocol suite being used on that host into a lower level address. The name mapping tables for that protocol suite must be arranged so that the address associated with that particular host is either the name of the foreign machine itself, if it speaks the same protocol family, or the name of an appropriate protocol converter, if protocol conversion is necessary to reach that host.

If a protocol converter is required the host must then open a lower level transport connection to the converter and send the connection open block down the stream. Only then does the protocol converter have the name of the host to which the connection is ultimately to be opened. The protocol converter can then repeat the name look-up operation, this time using the name mapping tables of the new protocol suite. Eventually, proceeding through each protocol converter in turn, the connection will finally be established to the ultimate host. At this point, the connection open block can be delivered to that machine. This will permit it to know the name of the machine at the

other end of the connection, and also provide the name of the service to be invoked (the naming of services is discussed in the section below).

The foregoing example illustrates why it is necessary to have a name structure at the USP level, but it does not explain what sort of names USP should provide. In fact, USP tends to be very general, and to permit almost any sort of name structure which is appropriate for the particular set of protocols and converters in use.

In general, all naming schemes can divided into one of two sorts, global or relative. In a global naming scheme, any particular host is assumed to be identified by the same character string name at any point within the USP environment. A postal address is an example of a global name. The address on the outside of an envelope is the same, no matter where the letter is posted (actually, the statement is not quite correct. When the source and destination of a letter is the same country, the name of the country is customarilly omitted from the envelope.)

The alternative to global naming is relative naming. In relative naming, the destination is named by giving a series of instructions based on the particular source. For example, "Go two blocks up the street, turn left, and find the third house on the right." is an example of a relative name. It is unable only so long as the starting point remains unchanged.

Conceptually, global names are easier to understand. However, there are many cases in which relative names are the only ones that can be successfully employed. The drawback to global names is that in order to create a global naming structure, it is necessary to identify a centralized management strategy for the name space. Hierarchical structure can be used to permit decomposition of name space management into separately controllable subcomponents, but even with hierarchy it is necessary to agree on single manager for the root of the hierarchy. In many cases, achieving this degree of centralized control is impossible. Thus, for example, in many of the mail systems which are in use today, the names of recipients are expressed as relative names. While this sometimes leads to great confusion in mail systems, it is also proved to be the only viable strategy. For this reason, USP provides a mechanism, called pathnames, to permit relative naming.

A pathname is assumed to consist of a series of names, separated by a recognizable delimiter. (The details of the delimiter are discussed below.)

The assumption is that the first name will be meaningful within the first protocol domain in which the USP connection is being established, the second name will be usable by the first protocol converter in entering into the next region, and so on. At each stage in the creation of the connection, the relevant component of the name will be interpreted, and then removed from the string, thus shifting to the front of the string the name which the next protocol converter is to use in turn. Eventually, as the last name in the string is used up, the connection that is established should reach the actual destination machine.

Thus, in the connection open block, the string which identifies the destination machine is used, piece by piece, as the connection is created. At the same time, the string that identifies the source of the connection must be built up, a piece at a time, by each protocol converter through which the connection goes.

As the reader will recognize, pathnames are considerably more complex to deal with than global names. Ideally, any particular users of USP will attempt to arrange a naming environment in which global names can be used. However, practical experience with the tying together of different protocol domains has suggested global naming structures can often not be achieved, for both political and technical reasons. Thus, this strategy of path names will prove, as a practical matter, of great importance.

It may be desirable for the user to pass names across the connection, while it is open. For example, in mail, the body of the text not only contains the name of the sender and the name of the recipient, but the names of all other people to whom copies of this message were sent. One of the most common failures which arises in mail systems occurs when these additional names are relative names, rather than global names. Normally, these additional names are transported from source to destination without being modified; since they are names relative to the source, they are thus meaningless at the destination. This makes it very difficult to reply to a piece of mail, because the user is required to translate the names manually into relative names which are meaningful at the destination. To solve this problem, USP provides a mechanism by which it will translate pathnames for the user as they are transported from the source to the destination. This is the purpose of the path-convert block types (300-399). In general, the translation which is performed on these names is similar to the translation which is done on names during connection set-up. However, as a name traverses the system from one side to the other it may be subjected to either deletion or addition of

components, depending on whether the name is being transported toward or away from the host which it names. The details of the rules for name transformation, both during connection set-up and as part of the path convert block type, is discussed in the section titled Name Translation below.

7.2. Service Naming

As part of connection set-up the application is expected to supply a character string name of the service which is desired on the foreign host. This name presumably maps, in the context of the destination machine, into a lower-level representation of the particular service that is to be invoked. A typical service might be called "Remote Login" or "Mail Delivery."

The simplest strategy for using these names would be to have the translation from character string to lower-level representation performed by the destination host, after the connection to that machine has been opened. However, for implementation reasons this is undersirable. If the translation is performed by the destination machine, then it is necessary to have a special dispatcher process on the destination machine to which the lower level connection is originally opened. When the connection is opened and the name has been transferred down the connection, only then can the correct destination for this connection be identified. At that point, it is necessary to hand off the lower-level connection from the dispatcher process to the process which will actually perform the service. In many systems, this is a complex operation, which may not be supported by the underlying system.

Most transport protocols already provide a mechanism for demultiplexing of incoming connection to the proper service. Some protocols use well known socket numbers, other protocols use a character string-based rendezvous structure. Whatever the strategy, it is appropriate to use that service-dispatching strategy instead of a special one for USP, because it is the design goal of USP to avoid building any demultiplexing mechanisms into this layer. To take advantage of the low-level demultiplexing structure, it is necessary that the translation from service name to low-level representation be done at the source of the connection, rather than the destination. Once the translation is done at the source, the proper low-level service description can be used as part of connection initiation, so that the low-level demultiplexing mechanism will work properly. This means that it is necessary, as part of USP, to provide another form of name-mapping, which translates from service names to low-level service representations, for each of the relevant protocol suites. In this version of USP, the strategy for implementing this name-look-up is not architected.

In simple implementations, it can be done by a fixed table within each of the hosts and protocol converters. However, it is necessary that names for services be centrally managed, and it will eventually be necessary to define, as part of USP, a mechanism for managing and translating these names.

8. Name Translation

[This section to be supplied later. Topics to be discussed here include the details of how names are translated as they are passed through protocol converters, and the details of how names are parsed, and delimiters for different name components are identified within a pathname string.]

9. Remote Procedure Call

Remote procedure call is currently used within many protocol suites as a mechanism for invocation of services across a network. In remote procedure call, the client invokes the service by sending a message across the net which contains a procedure to be invoked, along with the parameters which are provided to that service. The client then waits until a reply to this message arrives from the server. Thus, the pattern of interaction between the client and the server is similar to that of sub-routine invocation, hence the name. USP, as defined, does not provide explicit mechanisms for remote procedure call. In certain cases, it can be used to achieve this purpose. Since there is a substantial overhead to the establishment of a USP connection, it would not be appropriate to use USP in the case where only one request and reply will be transferred before the connection is closed. However, in those cases where a sequence of requests and replies will be sent, it is very easy and efficient to create a remote procedure call interaction using USP. For example, the different services which might be invoked on the server machine can be associated with different block types, and the block itself can be used to transport the parameters of the invocation. Thus, for example, one might open a connection to a remote file server, and then make a number of requests, to delete certain files, rename certain files, or list various directories. In certain cases, a series of remote procedure calls are associated with each other, in that the client and the server build up state information, within the context of which subsequent procedures are invoked. Thus, for example, the user might set a working directory, and then make additional file system calls relative to this established setting. Different protocols have different strategies for tying together the various procedure calls which are part of the same sequence. In USP, the preferred strategy is to use a distinct USP connection for each such sequence of remote procedure calls. An alternate,

such as including a transaction identifier in each procedure call, is to be avoided in general because it implies the requirement of demultiplexing different procedure calls based on this transaction identifier. Since demultiplexing is not to be done in the USP level, this strategy is inappropriate. In special cases, where the application has been designed in such a way that several different request streams are handled by the same process, then some sort of additional identifier can be provided, as one of the parameters of the invocation. However, application designers using USP are encouraged to avoid strategies which imply that some implementations may have to do demultiplexing at the USP level.

This strategy for creating sequences of remote procedure calls, in which the lower-level circuit abstraction is used to associate the different procedure calls, is in contrast to the strategy used by the Xerox NS Courier Protocol. In Courier, there is an architected transaction ID which is a part of each remote procedure call. The Courier specification explicitly states that one Courier connection can be used to transport procedures calls from several different sequences, and that several different connections can be used to transport procedure calls that are part of the same sequence. Thus, the Courier approach requires a dispatching or demultiplexing function in the Courier level protocol. Because we want to make USP realizable as a user-level process, so that it can be implemented on top of existing protocol packages, we cannot use this strategy for making sequences of remote procedure calls.

Many of the details of USP have been based on the data structures and strategies proposed by the Courier specification. This deviation from the specification is a regrettable but necessary result of our desire to insure the implementability of USP. Note, however, that USP could be used to transport the precise records which are used to achieve Courier remote procedure call. In other words, it would be possible to build a process, which sat on top of USP, and provided direct inter-operability with Courier-style remote procedures calls. The requirement would be that all Courier-level demultiplexing be done inside that Courier package.

10. Acknowledgements

The ideas of USP have come from a number of other protocols. Most obviously, many components of USP have been derived directly from Xerox NS Courier Protocols. Where possible, specifications for such things as data types have been specified identically.

The strategy for the management and conversion of pathnames is based on the ideas in the Network Independent Transport Service, otherwise described as the British Yellow Book Service. The idea of transporting blocks as sequence of sub-blocks is based on ideas in the Network Independent File Transport Service.

Appendix A

Implementation of USP Blocks on Defined Virtual Circuit Protocols

A. DOD TCP

The virtual circuit supplied by TCP is not block-structured, but is a simple stream of bytes. Therefore, the sub-block boundries must be defined by inserting data items into the stream itself. The representation of a block is two bytes which carry the block-type, followed by a number of sub-blocks. A sub-block begins with two bytes, the first four bits of which are flags, and the last twelve bits of which are the length of the sub-block, including the sub-block header. Thus, each sub-block can be no more than 4096 bytes long. Packet boundries are completely ignored in realizing the sub-block structure, since in TCP the packet boundries are not preserved, especially during re-transmission. The flags which are transported as the first four bits of each sub-block are coded as follows. The first bit is "on" if this sub-block is the last sub-block of the block. If it is "off," another sub-block follows. The next three flags are reserved for future use, and must be "0."

The block and sub-block headers are illustrated in Figure A1.

The DOD Protocol Suite provides a mechanism for naming hosts using character strings. If USP is being used entirely within the bounds of this protocol family, no extension to this name structure is needed to provide USP names. That is, internet-style domain names can be used directly as USP global names.

The strategy for naming services within TCP is based on "well-known ports." Well-known ports are particular reserved TCP end points, identified by small integers. To use USP over TCP, it will be necessary for the implementation to supply a table which translates between whatever USP based services are available, and the equivalent TCP port number. Note that different USP services should be associated with different TCP ports. That is, it is not appropriate to establish a single TCP port called "USP," for this would imply additional demultiplexing at the USP level. USP should simply be thought of as a particular strategy which a given service uses over a TCP connection.

11. X.25

X.25 is not, strictly speaking, a transport protocol. However, in certain contexts it is used as such, and therefore it is appropriate to offer a definition of USP implemented directly on X.25.

X.25 is a block-structured protocol. Thus, the block structure associated with USP can be directly provided by X.25. In particular, a USP block is transported as an X.25 complete packet sequence. Presumably, the sub-block structure can be use to correspond to packet boundries, but note that USP does not require that the sub-block boundries be preserved. Therefore, it is quite reasonable for X.25 to combine and break apart packets, as is required by the definition of a complete packet sequence.

X.25 provides only a limited strategy for identifying a type field with a block, in particular the Q bit. Thus, the 16 bits that give the type of each block must be transported as part of the data stream. They are the first two bytes of every block. The Q bit is reserved for future expansion, and must be "0."

X.25 uses a character string host-naming strategy. The most common definition of host names is the protocol X.121. These names are suitable for USP, within the single protocol environment.

X.25 provides a rather limited mechanism for identification of service. The call user data field (CUDF) which is transported as part of connection set-up, contains 4 bytes which can be used to select services. However, many implementations do not do an effective job of dispatching on these bytes. However, this field is the preferred mechanism for service dispatching, and it is assumed that when X.25 is used to support USP services, that a standard mapping will be supplied between those services implemented and acceptable values of the CUDF.

12. Xerox NS Sequenced Packet Protocol

Xerox NS Sequenced Packet Protocol is a block-structured virtual circuit. Thus, the block structured supplied by SPP can be directly used to transport USP blocks. Further, SPP provides a 16 bit typefield associated with each block (actually with each packet) which can be use to transport the typefield. Thus, no information associated with USP block structure need be transported in the data stream itself when implementing USP over SPP.

As part of the Xerox Protocol Structure there is a high level character string naming mechanism, Clearinghouse, which provides for a translation between names and host identifiers. This naming structure is suitable for USP in the single protocol context.

The Xerox architecture provides for service naming on the basis of well-known ports, reserved integer identifiers for the connection point for different services. If USP is to be used on top of SPP, it will be necessary to provide a table which translates between USP service names and these well-known ports.

13. CHAOS Protocols

The CHAOS Protocols provide a packet oriented reliable connection. That is, the packet boundries are significant in CHAOS transport, the packets are delivered reliably in order. A USP sub-block corresponds to a CHAOS net packet. The packets are assumed to be of 8 bit data. Packet type 200 is used for the last sub-block of a block; packet type 201 is used for all but the final sub-block of a block. Thus, a block consists of 0 or more data packets of type 201, followed by exactly one block of type 200. All other data types are reserved for future use, and must not be used.

The CHAOS protocols provide no mechanism for associating a typefield with a block, other than the data type which USP uses to define its sub-block structure. Thus, 16 bits that identify the typefield must be transported as the first two bytes of each block.

Some standard mechanism, either a name server or a distributed table, must be used to translate between character string names, as needed by USP, and internal CHAOS net addresses.

The CHAOS net protocols directly provide for service naming on character strings. Thus, USP service names can be directly mapped into the contact name transported in a RFC Packet.

14. DECNET

[This section to be supplied.]

15. ISO/NBS Transport

[This section to be supplied.]