M.I.T. Laboratory for Computer Science

Request for Comments No. 235
January 25, 1983

Report on an Implementation of SMP in CLU

by Brian A. Coan

1. Introduction

This is the report on an implementation of SWALLOW Message Protocol (SMP) in the CLU programming language. This implementation currently runs on the DEC System-20 under the TOPS-20 operating system, and plans call for it to be ported to the VAX under the UNIX operating system. The only previously existing implementation of SMP is in MESA and runs on the ALTO computer. The compatibility of the two implementations has been demonstrated.

SMP is a protocol that will be incorporated as an internal mechanism of the SWALLOW system; it is described in the SMP protocol specification which is the primary reference on the features common to all SMP implementations including packet formats, reliability guarantees, robustness guarantees, and general attributes. This report will not attempt to duplicate the information from the SMP protocol specification; rather, it will document those features that differ in the CLU implementation -- mainly user interface and performance.

2. User Interface

2.1. Definition of Terms

The communications interface implemented by SMP is referred to as a *port*; the data transmission facility provided by a port is a *connection* over which a pair of users can exchange *messages* of arbitrary size (up to 2^{32}-1 bytes). Only two sequences of message exchanges are permitted through a connection: either the *initiator* of the connection sends a *request* to the *responder* or the initiator of the connection sends a request to the responder and the responder sends back a *reply*. If further conversation is desired, a new connection must be established.

A single port can manage multiple concurrent connections. To facilitate this, a port is divided into *paths* each of which can support one concurrent connection. When a connection is finished, its associated path can be reused.

A message (either a request or a reply) is a sequence of (up to 2^{32}-1) *segments* each of which can hold up to 536 bytes of data. To send a message, a user breaks it up into segments and gives them one at a time to his *local* port which forwards them to a *foreign* port for reception by the foreign SMP user.

The mechanism for naming local SMP objects differs from that for naming foreign ones. There are three naming mechanisms used to name local SMP objects. A local port is named by a CLU variable of type smp_port, a local path is named by a CLU variable of type smp_path, and the local end of a connection has the same name as the local path that it goes through. There are two naming mechanisms used to name foreign SMP objects. The first is used directly by the user to name the foreign port to which a connection is to be established, and the second is used internally by SMP to associate message segments with a connection. The mechanisms are:

- The name of a port is the ordered pair (HOST, PORT) where HOST is the internet address of the computer on which the port is located and PORT is a 16-bit port number that is unique only on a particular computer. This port number can be determined by the local user and must be made known to any foreign user who wants to establish a connection.

- The name of a connection is the ordered triple (SOURCE, DEST, UID) where SOURCE is the name of the initiating port, DEST is the name of the responding port, and UID is a unique sequence number that will not be reused for another connection between these two ports. SMP uses this name in all message segments to associate them with a particular connection and therefore with a particular pair of paths.

A user is never given an explicit name for a path in a foreign port; the reason is that this feature would not be useful. When establishing a connection, a user can make use of any free path in the correct foreign port; they are all the same. When using an established connection, a user implicitly refers to a the foreign port and path just by using the connection.

2.2. SMP Operations

SMP is implemented by a CLU cluster whose name is SMP[1]. This section contains a list of all of the operations defined by the SMP cluster together with a description of their use.

2.2.1. OPEN_ANY and OPEN_THIS

Interface:

```
open_any=proc(max_buf:int) returns(port:smp_port)
      signals(busy,_internet_error(string))

open_this=proc(max_buf,port_number:int) returns(port:smp_port)
      signals(busy,_internet_error(string))
```

Usage:

Either OPEN_ANY or OPEN_THIS is used to create a port. OPEN_ANY is used when the user does not care what port number is used; SMP assigns a number at random. OPEN_THIS is used when the user wants to use a particular port number. Generally, OPEN_ANY is used when a port will only be used to initiate messages and OPEN_THIS is used when a port may be used to respond to (or just receive) messages initiated elsewhere.

Parameters,returned values, and signals:

max_buf	An upper bound on the number of buffers that may be allocated for either the transmission or the reception of a single message -- the equivalent of the number of message segments that can be in transit at any one time. This parameter should be chosen with some care because SMP throughput is not always an increasing function of it.

[1]Because SMP is implemented as one CLU cluster whose name is SMP, all of the SMP operations work on objects of type smp. Objects of type smp are of two kinds those that represent ports and those that represent paths. To simplify the explanation in the rest of this report these two kinds of SMP objects are presented as though they were implemented as the two different CLU data types smp_port and smp_path. Because they are, in fact, implemented as the one CLU type smp, an SMP user must resolve some loose ends left by the presentation in this report. The suggested method is either to use the CLU equate mechanism to make smp_port and smp_path be aliases for smp or to change all occurrences of smp_port and smp_path to smp. The only negative effect of all of this is that it postpones some type checking to run time.

port_number	The number that is to be assigned to the port created by this operation (OPEN_THIS only).
port	A returned value that is the port created by this call.
busy	A signal meaning that the create operation failed because either the requested port number is in use (OPEN_THIS only) or it is impossible to create any more ports because of insufficient system resources (OPEN_ANY and OPEN_THIS).
_internet_error	A signal meaning that some failure was detected by the network software that SMP uses. The nature of the error is given in the string returned with the signal.

2.2.2. CLAIM_PATH

Interface:

```
claim_path=proc(port:smp_port,to_net,to_rest,to_port:int)
        returns(path:smp_path) signals(usage(string))
```

Usage:

CLAIM_PATH is used to allocate a path that can later be used to establish a connection to a foreign port. It is only used by the initiator of a connection. At the responding port a path is allocated automatically and identified to the user along with the first message segment.

Parameters,returned values, and signals:

port	The port from which a connection is to be established.
to_net	The net part of the internet address of the computer to which a connection is to be established.
to_rest	The rest part of the internet address of the computer to which a connection is to be established.
to_port	The number of the port to which a connection is to be established.
path	A returned value that is the path allocated by this operation.
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.3. GET_BUFFER

Interface:

```
    get_buffer=proc() returns(buffer:_byteptr8)
```

Usage:

GET_BUFFER is used to allocate a reusable buffer in which a user can build a message segment prior to transmission. Buffers are of type _byteptr8 and should be manipulated with the operations of this type.

Parameters,returned values, and signals:

buffer	A returned value that is the buffer that was allocated.

2.2.4. SEND_FIRST

Interface:

```
    send_first=proc(path:smp_path,buffer:_byteptr8,total_bytes:int)
         signals(abort,usage(string))
```

Usage:

SEND_FIRST is used to send the first segment of either a request or a reply. A request is sent on a path that was returned by the claim operation; a reply is sent on the same path on which the request being answered was received. It is necessary to specify the total number of bytes that will be in the entire message. An attempt to send a different number will fail.

Parameters,returned values, and signals:

path	The path from which the message will be sent.
buffer	A buffer that holds the first segment of the message.
total_bytes	The total number of bytes that will be in the message (all segments).
abort	A signal meaning that the connection has been abnormally terminated; no further communication is possible
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.5. SEND_NEXT

Interface:

```
send_next=proc(path:smp_path,buffer:_byteptr8)
     signals(buffers_full,eot,abort,usage(string))
```

Usage:

SEND_NEXT is used to send subsequent segments of a message. It may signal that it is at present not possible to send any more segments of the message because all of the buffers allocated to SMP are full. In this case, the user must retry the operation later. The recommended code to wait and retry is presented in section 2.4 of this report.

Parameters,returned values, and signals:

path	The path from which the segment will be sent.
buffer	A buffer that holds the message segment to be sent.
buffers_full	A signal meaning that this operation cannot be completed at present because there are no available buffers.
eot	A signal meaning that the total number of bytes specified at the start of the transmission have already been sent. No more data will be accepted.
abort	A signal meaning that the connection has been abnormally terminated; no further communication is possible
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.6. GET_FIRST

Interface:

```
get_first=proc(port:smp_port,auto_reuse:bool)
     returns(path:smp_path,buffer:_byteptr8)
     signals(none_pending)
```

Usage:

GET_FIRST is used to initiate the reception of a request. It either returns both the first segment of a request and the path on which it is being received or it signals that there are at present no pending requests. The recommended code to wait and retry is presented in section 2.4 of this report.

The use of GET_FIRST is not analogous to the use of SEND_FIRST in that the former is only used to receive the first segment of a request and the latter is used both to send the first segment of a request and to send the first segment of a reply.

Parameters,returned values, and signals:

port	The port on which a request is to be received.
auto_reuse	A boolean parameter that indicates whether SMP may try to perform certain optimizations. A value of FALSE means that no optimizations will be attempted. A value of TRUE means that the optimizations may be attempted and that the user agrees to completely process the returned buffer before requesting another buffer in order to facilitate these optimizations.
path	A returned value that is the path through which the request is being received.
buffer	A returned value that is the buffer that contains the first segment of the request.
none_pending	A signal that means that there are currently no requests pending.

2.2.7. OK_TO_RECEIVE

Interface:

```
    ok_to_receive=proc(path:smp_path) signals(usage(string))
```

Usage:

OK_TO_RECEIVE is used to indicate a willingness to receive a request. It should be invoked right after the first packet of a request has been received. Its use is optional in that a request for a subsequent packet of a message will include an implicit OK_TO_RECEIVE.

Parameters,returned values, and signals:

path	The path on which the operation is to be performed.
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.8. GET_NEXT

Interface:

```
get_next=proc(path:smp_path,auto_reuse:bool)
     returns(buffer:_byteptr8)
     signals(none_pending,eot,abort,usage(string))
```

Usage:

GET_NEXT is used to receive any segment of a message except for the first segment of a request. It may signal that there are at present no message segments pending. In this case the user must retry the operation later. The recommended code to wait and retry is presented in section 2.4 of this report.

Parameters,returned values, and signals:

path	The path on which a message is being received.
auto_reuse	A boolean parameter that indicates whether SMP may try to perform certain optimizations. A value of FALSE means that no optimizations will be attempted. A value of TRUE means that the optimizations may be attempted and that the user agrees to completely process the returned buffer before requesting another buffer in order to facilitate these optimizations.
buffer	A returned value that is the buffer that contains the next segment of the message.
none_pending	A signal that means that there are currently no message segments available.
eot	A signal meaning that the entire message has been received.
abort	A signal meaning that the connection has been abnormally terminated; no further communication is possible
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.9. WAIT

Interface:

```
wait=proc(port:smp_port,timeout:int)
```

Usage:

WAIT is used to suspend operations until there is a significant event in the SMP interface. It should be used when the user needs to wait for SMP to either send or receive some message segments. Its use is demonstrated in section 2.4 of this report.

Parameters,returned values, and signals:

port	The port on which the wait operation is to be performed.
timeout	A timeout value in milliseconds; a negative value means forever.

2.2.10. ABORT

Interface:

```
    abort=proc(path:smp_path) signals(usage(string))
```

Usage:

ABORT is used to abnormally terminate a connection.

Parameters,returned values, and signals:

path	The path on which the operation is to be performed.
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.11. GIVE_UP

Interface:

```
    give_up=proc(path:smp_path) signals(usage(string))
```

Usage:

GIVE_UP is used to relinquish a path after its use for a particular connection is finished. All paths that are obtained with CLAIM or GET_FIRST must be explicitly given up using GIVE_UP before SMP will permit their reuse.

Generally, GIVE_UP is used when a conversation through a port has terminated (i.e. either one or two complete messages have been exchanged or the connection has been aborted). If, however, it is used on a path that is still active it aborts the connection before freeing the path.

Parameters,returned values, and signals:

path	The path that is to be given up.
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.12. CLOSE

Interface:

```
    close=proc(port:smp_port)
```

Usage:

CLOSE is used to close a port. All transmission and reception through the port is halted immediately, and the resources used by the port (including its number) are freed. No attempt is made to notify any foreign ports to which there are connections that they are being abandoned.

Parameters,returned values, and signals:

port	The port to be closed.

2.2.13. GET_TOTAL

Interface:

```
    get_total=proc(path:smp_path) returns(total:int)
          signals(usage(string))
```

Usage:

GET_TOTAL is used to find out the size (in bytes) of the message that is currently being transmitted or received over a path.

Parameters,returned values, and signals:

path	The path on which the inquiry is to be performed.
total	A returned value that is the size of the message.
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.14. GET_DONE

Interface:

```
get_done=proc(path:smp_path) returns(done:int)
     signals(usage(string))
```

Usage:

GET_DONE is used to find out how many bytes of the current message have already been sent or received.

Parameters,returned values, and signals:

path	The path on which the inquiry is to be performed.
done	A returned value that is the number of bytes already sent.
usage	A signal meaning that this operation failed because SMP has detected what appears to be a user error.

2.2.15. GET_NUMBER

Interface:

```
get_number=proc(port:smp_port) returns(number:int)
```

Usage:

GET_NUMBER is used to find out the number of a port. This number is used when establishing a connection to the port.

Parameters,returned values, and signals:

port	The port whose number we want to know.
number	A returned value that is the number of the port.

2.2.16. GET_RESENDS

Interface:

```
get_resends=proc(port:smp_port) returns(resends:int)
```

Usage:

GET_RESENDS is used to find out how many packet retransmissions were done. If this number is substantially different from zero, the cause may be that the value of pause is too small. This is a problem that should be attended to because a substantial performance degradation may result.

Parameters,returned values, and signals:

port	The port on which the inquiry is to be performed.
resends	A returned value that is the number of negative acknowledgment packets that have been received at the specified port.

2.2.17. SET_PAUSE_MSEC

Interface:

```
set_pause_msec=proc(pause:int)
```

Usage:

SET_PAUSE_MSEC is used to set the minimum time between packets for all packets sent by any port implemented by this instance of SMP.

Parameters,returned values, and signals:

pause	The minimum time (in milliseconds) that SMP will wait between sending packets. SMP uses zero until another value is specified.

2.3. SMP States and State Transitions

A path is in one of 8 internal states. Transitions between the states are caused by invocation of SMP operations, by the reception of packets from other ports, and by timeouts. This section contains a definition of the path states, a definition of the SMP packet types, and a presentation of the state transition diagram. The states are defined as follows:

Undefined	The path is currently completely idle.
Claimed	The path has been reserved by the user for sending a request. No segments of the request have yet been sent.
Sending	The path is being used to send a request. At least one segment of the request has been sent.
Receiving	The path is being used to receive a request. At least one segment of the request has been seen by the user.

Sending reply	The path is being used to send a reply. At least one segment of the reply has been sent.
Receiving reply	The path is being used to receive a reply. At least one segment of the reply has been received by the SMP port.
Requesting	The path contains the first segment of an incoming request. The request has not yet been seen by the local user.
Aborting	The connection through this path has been aborted for one of the following three reasons: the local user aborted the connection, SMP timed out, or an abort packet was received.

The SMP packet types are defined as follows:

FSM	First segment of message. FSM-request is the first segment of a request and FSM-reply is the first segment of a reply.
SSM	Subsequent segment of message.
SRA	Segment request and acknowledge.
NAK	Negative acknowledgment.
ABORT	Abort.

State Transition Diagram:

	CURRENT STATE:							
	1	2	3	4	5	6	7	8
CALLS:								
claim_path[1]	2	–	–	–	–	–	2	–
send_first	E	3	E	5	E	E	E	A
send_next	E	E	3	E	5	E	E	A
get_first[1]	–	–	–	–	–	–	4	–
ok_to_receive	E	E	N	4	E	E	E	N
get_next	E	E	3	4	E	6	E	A
abort	N	8	8	8	8	8	N	N
give_up	E	1	1	1	1	1	E	1
timeout (demon)	–	–	8	8	8	8	–	–
PACKETS:								
FSM-request[1]	7	–	–	–	–	–	–	–
FSM-reply	–	–	6	D	D	D	D	D
SSM	–	–	D	4	D	6	D	D
SRA	–	–	3	D	5	D	D	D
NAK	–	–	3	D	5	D	D	D
ABORT	–	–	8	8	8	8	1	D

<– next state
1 = undefined
2 = claimed
3 = sending
4 = receiving
5 = sending reply
6 = receiving reply
7 = requesting
8 = aborting
E = error signal
A = abort signal
N = call is a no-op
D = drop packet
– = case NEVER occurs
no state change for A, E, N, or D

[1]A call or packet usually refers to a specific path; however, claim, get_first, and FSM-request do not. Instead, they search all paths and select one from which there is a legal transition. The procedure claim will prefer a path in state 1 to a path in state 7. If there is no satisfactory path, claim signals all_busy, get_first signals none_pending, and FSM-request drops the packet.

1. Sample Code That Uses SMP

The following fragment of CLU code send a request and receives a reply:

```
smp_port = smp
smp_path = smp
port:smp_port := smp$open_this(<number of buffers>,<port number>)
path:smp_path := smp$claim_path(port,<net>,<rest>,<port number>)
buffer:_byteptr8 := smp$get_buffer()
<put data in buffer>
smp$send_first(path,buffer,<number of bytes to be sent>)
<put more data in buffer>
while true do
   smp$send_next(path,buffer) except
      when buffers_full: smp$wait(port,-1) continue
      when abort: <deal with the failure>
      when eot: break
   end
   <put more data in buffer>
end
while true do
   buffer := smp$get_next(path,false) except
      when none_pending: smp$wait(port,-1) continue
      when abort: <deal with the failure>
      when eot: break
   end
   <process buffer>
end
smp$give_up(path)
```

The following fragment of CLU code repeatedly receives a request and sends a reply:

```
smp_port=smp
smp_path=smp
port:smp_port := smp$open_this(<number of buffers>,<port number>)
while true do
   smp$wait(port,-1)
   path:smp_path, buffer:_byteptr8 := smp$get_first(port,false)
      except when none_pending: continue
   end
   <process buffer>
   while true do
      buffer := smp$get_next(path,false) except
         when none_pending: smp$wait(port,-1) continue
         when eot: break
         when abort: <deal with the failure>
      end
      <process buffer>
   end
   if <there is a reply> then
      <put first segment of reply in buffer>
      smp$send_first(path,buffer,<total bytes in reply>) except
         when abort: <deal with the failure>
      end
      <put second segment of reply in buffer>
      while true do
         smp$send_next(path,buffer) except
            when buffers_full: smp$wait(port,-1) continue
            when eot: break
            when abort: <deal with the failure>
         end
         <put next segment of reply in buffer>
      end
   end
   smp$give_up(path)
end
```

3. Status

SMP in CLU is now operational; it has been demonstrated both transmitting to itself and transmitting to the MESA implementation of SMP. The remainder of this section contains a full report on the status of the implementation including observations on the portability and efficiency of the code, statistics that measure the throughput rates that the code has achieved, a list of problems found in the SMP specification together with their solution, a list of bugs found in related code together with their current status, and some observations on the robustness of the SMP code.

3.1. Portability and Efficiency of the Code

The design of the SMP code is a lean one that should facilitate efficiency and portability at the expense of slightly reduced generality. The particular design choice with the biggest impact on this is that SMP is implemented as a cluster with no separate processes and therefore no interprocess communication and no independent scheduling of subprocesses.

There are two principal advantages of a single process implementation. First, the mechanisms for interprocess communication vary widely from one operating system to another. Excluding this complexity from SMP has made it possible to write code that will likely be easy to port to another machine running CLU. Second, on some machines interprocess communication is expensive. These costs are avoided by SMP's using only one process. The disadvantage is that the SMP code does not have full control over its own scheduling. The user can fail to call SMP for a long period of time and cause the timeout of a connection because SMP code is unable to respond in a timely fashion. However, because of the design of SMP, this problem is also present (but to a lesser extent) in a multi-process version of SMP. It arises because a user may delay a long time before sending the next packet on a connection. SMP is then unable to send any packets (data or control) on the connection and the foreign SMP implementation may timeout. A natural example is a single packet request with a reply that takes a long time to compute. Until the first packet of the reply arrives, the initiator of a connection is unable to predict whether or not the responder will ever answer.

A solution to this problem might be an I-am-still-alive packet that an SMP implementation could send to its counterpart. It would mean: I don't have anything for you now, but please don't go away. If such a mechanism were introduced, it could also be used as a partial solution to the timeout problem in the single process implementation. If a user program were going to spend a long time between normal calls to SMP it could periodically request that SMP send out an I-am-still-alive packet on all active connections. This is only a partial solution because it still requires the user to be conscious of protocol timing considerations.

3.2. Throughput from TOPS-20 to TOPS-20

On a DEC System-20 computer running the TOPS-20 operating system (the MIT-XX computer), the communications throughput between two separate copies of SMP (both on the same computer) was measured. The measured throughput may be artificially low (compared to the throughput between two separate computers) because of the problem of self-interference, or it may be artificially high because no transmission through a network was required. A more reliable measure of throughput will be available after the CLU implementation of SMP has been ported to other machines.

The throughput has been measured in bits per second. Numerous measurements were taken over a wide range of values for system load where system load is measured as the average number of processes on the system run queue. The results are tabulated below:

System Load	Number of Observations	Throughput (10^3bits/sec) Mean+-Standard Deviation
1-2	30	216+-78
2-3	6	169+-71
3-4	18	94+-66
4-5	19	36+-16
5-6	18	29+-9
6-7	8	26+-14
7-8	5	19+-3
8-9	15	20+-3
>9	15	20+-4

3.3. Throughput Between TOPS-20 and MESA

The throughput observed between SMP in MESA and SMP in CLU has been disappointing. In both directions the observed throughput has been consistently in the range of 20 to 25 thousand bits per second. It is believed that this is in large part caused by a network interface that connects the two implementations; the interface (bridge) drops packets when its load exceeds 60 packets a second. One confirmation of this theory is the observation that the throughput from SMP in CLU to SMP in MESA rises to 60 thousand bits per second when the CLU code is modified to never send two packets separated by less than 17 milliseconds.

The maximum throughput that could be achieved by making full use of the 60 packets a second bandwidth of the bridge is about 190 thousand bits per second (based on 25% control packets in

SMP). The difference between the achieved rate of 60 thousand bits per second and this bound has not been fully explained although it could be due to operating system overhead, contention for resources, inefficiencies in one or both of the SMP implementations, or some other cause.

3.4. Proposed Amendments to the SMP Specification

During the implementation of SMP several problems not adequately addressed by the SMP specification came to light. This section contains a list of these problems together with proposed resolutions. The resolutions presented are consistent with the two existing implementations of SMP.

3.4.1. The Direction Bit of the UID Field

Problem: The SMP specification requires that the low order bit of the UID field indicate the direction of a packet, either request or reply. However, it is not specified which value corresponds to which direction; also, it is not explicitly stated how the direction bit is used in control packets (SRA, NAK, and ABORT).

Proposed resolution: Zero will be used to indicate a request data packet, and one will be used to indicate a reply data packet. Acknowledgment and negative acknowledgment packets will have a direction bit with the same value as the direction bit of the packet that is being acknowledged or negative acknowledged. An abort packet will have a one in the direction bit if and only if at least one reply data packet has been either sent or received on the corresponding connection. This proposed resolution is consistent with the two existing implementations of SMP.

3.4.2. The Transmission Order of Bytes

Problem: SMP packet headers contain some 2-byte integers and some 4-byte integers. The SMP specification does not say in what order the bytes of these integers will be transmitted.

Proposed resolution: The high order byte of a 2-byte integer will be sent first. That is, the integer $A*10^8+B$ will be sent in the order A then B. The low order 2-byte piece of a 4-byte integer will be sent first; it will be sent in the order high byte then low byte. That is, the integer $A*10^{24}+B*10^{16}+C*10^8+D$ will be sent in the order C then D then A then B. This proposed resolution, bizarre as it might seem, is consistent with the two existing implementations of SMP.

3.4.3. Numbering of SMP Packets

Problem: The SMP specification requires that SMP data packets be numbered starting at 0. This is inconsistent with both of the existing implementations.

Proposed resolution: SMP data packets will be numbered starting at 1 (the FSM packet, whose number is implicit). This will reduce the number of data packets that can be used for a message by 1 to 2^{32}-1 which is not a significant restriction. This proposed resolution is consistent with the two existing implementations of SMP.

3.4.4. Restrictions on Window Size

Problem: The SMP specification requires that each initial data packet and each acknowledgment packet specify a window size. It is not adequately specified how this window size is to be chosen. Some convention is required in order to make SMP implementable. The particular convention recommended below -- one that is somewhat stronger than absolutely required -- greatly simplifies the implementation at the expense of minimal loss of functionality.

Proposed resolution: Before SMP sends any packets on a connection, it will decide how many buffers it is willing to use for the connection. This decision may not later be changed. The window size that SMP specifies will be the minimum of the number of buffers it is willing to use and of the window size specified by the other end of the connection. This number will converge in one step to the size of the smaller of the two buffer pools allocated to the connection. This proposed resolution is consistent with the two existing implementations of SMP.

3.5. Bugs

During the implementation of SMP in CLU several bugs were found in related software. This section contains a list of the bugs found and their current status.

~ The MESA implementation of UDP computes checksums incorrectly in that it does not include protocol number and packet length. Status: For testing purposes this bug has been circumvented by disabling UDP checksums. The code to fix this problem has neither been written nor installed.

~ The MESA implementation of UDP stops working after it receives a packet with a bad checksum. Status: The code to fix this problem has been written but not installed.

~ It is not possible to send very large SMP packets (535 or 536 bytes of data) from an

ALTO to XX. Status: The cause of the problem has been traced to the apparent inability of XX to receive internet packets of the maximum size. This problem has been reported to the XX system administrator. Resolution is still pending.

~ An attempt to send more than about 60 packets a second between XX and an ALTO will result in the excess packets being dropped. Status: The problem is caused by a weakness in the bridge between the two networks. This problem is known to the maintainer of the bridge and he expects it to be resolved eventually.

3.6. Robustness

The SMP protocol has been demonstrated to be robust in the presence of very high packet loss rates (up to 50%) with one exception. That exception is that at the end of a message the sender fails to wait for the receiver to make requests for retransmits. This causes the entire message transmission to be wasted. There are various possible remedies that can be used to fix this problem; however, since robustness was not a design goal it is recommended that no action be taken.

4. Future Work

The most important item of future work is to port SMP to a VAX 11/750 running UNIX. Doing this should put some of the claims made in this report to the test. Specific questions that should be answered are whether SMP is as portable as is claimed in this report and what the throughput will be between two copies of SMP in CLU running on different machines. Also, it might be possible to gain further insights into the extent to which network limitations (like a slow bridge) degrade the performance of SMP. Porting SMP to the VAX is, therefore, an item of the highest priority.

Other future work includes following up on the bugs and other problems listed in sections 3.4 and 3.5 of this report.