M.I.T. Laboratory for Computer Science
Computer Systems Research

Request for Comments No. 202
November 21, 1980

# SMP Protocol Specification

by Leandro Lopez

## 1. Overview

The SWALLOW Message Protocol (SMP) is a communications protocol intended for the efficient transfer of arbitrary size messages between nodes in a distributed data storage environment. More particularly, it implements the communications substrate of the SWALLOW data storage system [1].

The protocol allows several client processes to simultaneously send and receive multiple messages. Clients interface to SMP through *ports*; ports themselves are subdivided into *paths*. Each message is sent through a uniquely named path.

Transmission of short messages involves minimum overhead since no connection setup nor acknowledgment is required. SMP will guarantee the integrity of a received message but the higher level protocols are responsible for detecting duplicated and lost messages. Long messages, on the other hand, are broken up into variable length segments for transmission. A segment number associated with each segment, together with the unique path name, makes detection of duplicated and unordered segments possible. Lost packets are detected with the help of timeouts; retransmission is requested via negative acknowledgments (selective retransmission). Both ends are able to abort an ongoing transfer. The nature of the problem encountered is reported, when possible, to the client.

Each path is individually flow controlled; the end to end flow control mechanism is based on a credit scheme.

SMP makes use of the underlying datagram service provided by the User Datagram protocol [2].

## 2. Interfaces

SMP interfaces on one side to the higher level client processes and on the other to an underlying datagram service.

Message transmission from the client point of view is carried out by a sequence of appropiate procedure calls. A minimum set of such procedures must include means for opening and closing ports, managing paths, sending and receiving segments of messages, and querying the status of a path. Appendix 2 presents a MESA definitions file of a typical client interface.

SMP utilizes the underlying datagram service offered by the User Datagram Protocol (UD) [2], which provides unreliable and unordered port-to-port transport of packets. On top of it SMP implements the path management functions as well as flow control and error control on a per path basis.

## 3. Addressing and Multiplexing

An SMP implementation allows several client processes to concurrently send and receive multiple messages. Clients interface to SMP via one or more *ports*. Each port is further subdivided into *paths* through which messages are transferred. Hence, each client may simultaneously be sending and receiving several messages, each through a different path.

Paths are bidirectional, but at most one message may be transferred in each direction. This feature allows clients to send a higher level request and receive the corresponding response through the same path. A path is specified by the concatenation of an origin-destination host address pair, an o.d. port number pair and a unique path identifier (UID). The least significant bit of the UID distinguishes between messages sent through an unused path (requests) and messages transmitted through a previously active path (responses). The UID is unique among all messages originated by a given port; therefore, every message is transmitted through a different, network-wide unique path.

## 4. Packet Format and Purpose

Five packet types are defined: two of them are data carrying packets, the remaining three serve different control functions. Every packet includes its path identifier, but its format is type dependent.

SMP protocol packets are transmitted as datagrams. A fixed-size SMP header follows the UD header supplying additional SMP specific control information. Finally, packet type dependent data and control fields complete the packet. The data field is variable in length; each data carrying packet may contain from 0 to 536 data bytes[1].

(1) The Internet Protocol [3] —underlying the User Datagram protocol—recommends a packet-length upper limit of 576 bytes. The typical IP header is 20 bytes long, UD uses an additional 8 bytes and SMP requires 12 bytes for its header, leaving a maximum of 536 bytes for data.

The header comprises the following fields:

| | |
|---|---|
| UID | (32 bits) |
| Opcode | (16 bits) |

The UID should be distinct for every message originated by a given source. The opcode defines the packet type, one of the following:

| | | |
|---|---|---|
| (1) | FSM | First Segment of Message |
| (2) | SSM | Subsequent Segment of Message |
| (3) | SRA | Segments Request and Acknowledgment |
| (4) | NAK | Negative Acknowledgment |
| (5) | ABORT | Abort Request |

An FSM packet carries the first, or only, segment of a message. It also informs the recipient the total length of the message (in bytes) and the maximum number of outstanding segments the source is able to handle (i.e. the number of slots in its retransmission queue). Knowledge of the total length of the message allows the recipient to allocate the appropiate amount of resources, or to abort the transfer in the first place if there is not enough space for storing the incoming message. Also, the recipient can determine the end of the message without need of a special signal. For flow and error control purposes it is important that the receiver know the size of the sender's retransmission queue, since it imposes an upper limit on the window size.

SSM packets are utilized to send the rest of the segments composing a message. A segment number identifies the position of a given segment within the complete message. The segment sent along with the FSM packet is by definition segment number zero, thus the range of SSM segment numbers is [*1,(tns-1)*], where *tns* is the total number of segments of a message. The segment number is represented by a 32-bit long integer; hence, messages may have up to $2^{32}$ segments.

The purpose of SRA packets is to acknowledge the correct reception of all segments, up to and including the one specified by the contents of its *acknowledged segment number (asn)* field, and to report a *window* (i.e. number of segments the source is allowed to transmit starting with segment number *asn+1*).

When packets get lost or arrive damaged, the recipient requests their retransmission via NAK packets. The segment that needs retransmission is specified by the contents of the *requested segment number* field.

An ABORT packet is used, when an unrecoverable error is detected or another insurmountable problem is encountered, to inform the other end of the source-sink pair to abort the ongoing transfer. This is only an optimization since the inability to complete the transfer will always be detected by either sender or receiver timing out. The nature of the problem encountered is encoded in the error type field. A variable length error message may be appended.

## 5. Flow Control

The purpose of an end to end flow control mechanism is to adjust the source rate to the sink rate so as to avoid sink buffer congestion. Moreover, this is to be acomplished in a way that maximizes path throughput.

### *5.1 Mechanism*

SMP's flow control mechanism is based on a credit scheme. The recipient informs the sender how many segments it is permitted to send without need of an acknowledgment (i.e. specifies a window). The window size is limited by the number of free buffers allocated for message reassembly at the receiving end, and also by the size of the source's retransmission queue. This precludes any possibility of reassembly deadlocks[1] and assures that retransmission requests will be honored.

When a message transfer is initiated, the first packet sent informs the recipient the total length of the message, as well as the number of outstanding segments that the source can handle. The source will restrain from transmitting any segments until it is told to do so by means of a *segments request and acknowledgment* (SRA) packet, which acknowledges the correct reception of the initial segment and requests the transmission of a certain number of segments. The recipient makes further requests each time that the client —having processed the correctly received segments— frees up some buffer space. This process is repeated until the whole message is transferred.

### *5.2 Policy*

The flow control policy is concerned with (1) deciding how many segments to ask for, and (2) when to make the request, in order to optimize the use of the available resources (e.g. network bandwidth, disk transfer rate, etc.) and maximize the throughput.

If the recipient waits to send an acknowledgment till it receives all packets of a previously requested window, a silence period of duration at least equal to the round trip delay will be inevitable. Note also that since acknowledgments free buffer space at the source node, a delayed acknowledgment may, in some cases, unnecessarily slow down the source client (e.g. if the client runs out of buffer space and all buffers are used up by the retransmission queue, a new segment will not be immediately available for transmission at the moment an acknowledgment arrives). Careful allocation of buffers to both client and retransmission queue will solve this problem. Still another disadvantage of the single acknowledgment is that, if it gets lost, the only way to detect it is by means of a costly timeout. On the other hand, sending an acknowledgment for each correctly received packet will generate unnecessary traffic and added burden to both hosts. The best compromise is to send an acknowledgment when the number of free buffers exceeds a certain threshold value. The optimum threshold value will, in general, depend on several factors: window size, network delay and client's source and sink rate.

(1) A reassembly deadlock occurs when the receiver is waiting for an out of sequence segment but no buffer space is available for it.

On a high bandwidth, low error-rate local network environment, the effect of the round trip delay is minimized and it is possible that sending a single acknowledgment after receiving the complete window will not significantly degrade the throughput. Some experimenation is needed in order to decide on a final policy.

## 6. Error Control

Packets may get lost, damaged, become duplicated or may arrive in an order different to the one in which they were sent. The recipient should be able to detect such occurrences and recover from them.

### *6.1 Error Control at Receiver*

The recipient SMP module will discard any incoming packet whose path does not correspond to one of the active paths. The only exception to this rule is, of course, the case of an FSM packet. Hence FSM packets are "dangerous" in the sense that duplicated ones may not be detected by SMP (for example, when the duplicated packet arrives once the corresponding path is no longer active). As previously mentioned, the client should recover from such events.

Recall that each segment carries an identifying segment number that defines its position within the complete message. This enables SMP to detect duplicated and unordered segments. Duplicated segments are simply discarded; the same applies to out of range segments, i.e., segments not belonging to the current window. Out of sequence segments are accepted since enough buffer space for resequencing is guaranteed by the flow control mechanism. Lost packets are detected with the help of timeouts, i.e., if a requested packet is not received in some specified amount of time it is assumed it got lost.

An informal description of the error recovery procedure of the recipient is the following. After receiving an FSM packet, the recipient issues a request for a certain number of packets and starts a timer. Under normal circumstances packets will start arriving before the timer runs out, but should a timeout occur, the request is retransmitted (probably got lost). The timer is restarted after receiving each segment. If a timeout occurs, the recipient checks for all missing segments and requests the retransmission of each of them issuing a NAK (selective retransmission). NAKs may also get lost; a timeout again will signal the need for retransmission. Should a certain number, say three, of consecutive timeouts happen, then the message transfer is aborted for in this case the source probably crashed or the transmission channel failed.

In the case of a local network—where packets usually arrive in order—it is probably a good idea to immediately request retransmission of all lower numbered missing packets when an out of order packet arrives, instead of waiting for the timeout.

### *6.2 Error Control at Sender*

The same timeout and retransmission policy applies to the transmission of FSM packets at the source end (only in the case of long messages, see below). Once an FSM packet has been acknowledged, however, it is the responsibility of the recipient to detect lost packets and request their retransmission. The source node will only timeout and abort the transfer if it does not hear from the recipient for a reasonably long period of time.

## 7. End of Transfer

After sending the last segment of a message the source considers its job done and does not expect a final acknowledgement. Correspondingly, the recipient does not acknowledge the last segment received. Note that, as a result, single fragment messages are not acknowledged.

Another consequence of this is that in the case of long messages, the sender will not wait for an acknowledgement after sending the last segment, but this last segment will most certainly belong to a window. Thus, all segments composing the last window will not be retransmitted in case they are damaged or lost. Although the probability of this happening is very small, it is nonetheless positive. The receiving SMP module will detect the error and report it to the client, who is responsible for recovering from such occurences.

## References

[1] Reed, D. P., Svobodova, L., "SWALLOW: A Distributed Data Storage System for a Local Network," M.I.T. Laboratory for Computer Science, RFC No. 192, June 1980.

[2] Postel, J., "User Datagram Protocol," IEN-88, USC-Information Sciences Institute, May 1979.

[3] Postel, J., "Internet Datagram Protocol," IEN-80, USC-Information Sciences Institute, February 1979.

## Appendix 1

*SMP Packet Format*

Five packet types are defined:

| | |
|---|---|
| (1) FSM | First Segment of Message |
| (2) SSM | Subsequent Segment of Message |
| (3) SRA | Segment Request and Acknowledgment |
| (4) NAK | Negative Acknowledgment |
| (5) ABORT | Abort Request |

Every packet carries a fixed size header followed by packet-type dependent fields. The fixed header consists of a unique path identifier (UID) and a packet type identifier (Opcode). The least significant bit of the UID should be set to "1" when the message is being transmitted through a path that was previously used to receive a message; to "0" otherwise. The following table specifies the format for each packet type:

| *Packet Type* | *Field* | *Length* |
|---|---|---|
| FSM | UID | 4 bytes |
| | Opcode | 2 bytes |
| | Total message length (in bytes) | 4 bytes |
| | Max. number of outstanding packets | 2 bytes |
| | Data | 0 to 536 bytes |
| SSM | UID | 4 bytes |
| | Opcode | 2 bytes |
| | Segment number | 4 bytes |
| | Data | 0 to 536 bytes |
| SRA | UID | 4 bytes |
| | Opcode | 2 bytes |
| | Acknowledged segment number | 4 bytes |
| | Window size | 2 bytes |
| NAK | UID | 4 bytes |
| | Opcode | 2 bytes |
| | Segment number | 4 bytes |
| ABORT | UID | 4 bytes |
| | Opcode | 2 bytes |
| | Error code | 2 bytes |
| | Error message | 0 to 536 bytes |

The correspondence between the opcode value and the packet type is:

| *Packet Type* | *Opcode Value* |
|---|---|
| FSM | $(1)_8$ |
| SSM | $(2)_8$ |
| SRA | $(3)_8$ |
| NAK | $(4)_8$ |
| ABORT | $(177777)_8$ |

## Appendix 2

```
--SmpClient.mesa (by Gail Arens)

--This module defines the client's interface to the Swallow Message Protocol package.

--Last change: November 6, 1980 by: LL

                          DIRECTORY

ProcessDefs:      FROM "ProcessDefs" USING [Ticks],
SmpPath:          FROM "SmpPath" USING [Path, PathType],
SmpPrimitives:    FROM "SmpPrimitives" USING [User, InternetAddress, ErrorType,
                  StatusType, Port];


SmpClient: DEFINITIONS =

BEGIN OPEN SmpPrimitives;


-- TYPES --

AbstractObject: TYPE = {hidden};

Path: TYPE = SmpPath.Path;

Port: TYPE = SmpPrimitives.Port;

User: TYPE = SmpPrimitives.User;

Siteid: TYPE = SmpPrimitives.InternetAddress;


-- ERRORS --

CommunicationError: ERROR [e: ErrorType];

TooManyPaths: ERROR [excess: CARDINAL];
```

**-- PROCEDURES --**

**-- Starting up and shutting down the Smp package --**

StartUp: PROCEDURE [maxbuffers, bufferpoolsize, pathpoolsize, rqSize: CARDINAL];

-- *This procedure does all initializations and then starts up the SMP module.*
-- *maxbuffers = max. number of buffers each client can claim*
-- *bufferpoolsize = total number of buffers assigned to SMP*
-- *pathpoolsize = defines max. number of concurrently active paths*
-- *rqSize = size of the receive process queue.*

ShutDown: PROCEDURE;

-- *Gracefully turns off the Smp package. Finishes up whatever buffer it is sending or receiving at the moment and then stops all communications with the network.*

**-- Registering with Smp --**

CreateClient: PROCEDURE RETURNS [port: Port];

-- *Gives the client an unused port through which it can send and receive messages. All packets of messages that are sent to a SMP client must include its port in the address.*

OpenClient: PROCEDURE [port: Port];

-- *Opens a specific port. Signals PortError if that port number is already being used.*

DestroyClient: PROCEDURE [port: Port];

-- *Cancels the given client. Smp won't accept any messages addressed to this client once this procedure is executed.*

**-- Management of the paths --**

-- There are multiple paths for each port through which messages can be sent or received. This allows the client to send or receive more than one message concurrently. Only one message can be transmitted through a path at once and the complete message must be transmitted through the same path. In order for a client to send a message it must first *Claim* a path. When receiving, all first segments of a message sent to the client are directed to the client's port and then are received through any available path so it doesn't have to *Claim* one. When the client receives the first buffer it also receives the path so that it can receive any subsequent segments that exist for that message. In either case, the client must *GiveUp* the path once it has sent or received the message.

Claim: PROCEDURE [source, destination: User] RETURNS [p: Path];

-- *Claims a path in the port through which a client can send a message. A new path must be claimed for every message sent. The client never has to claim a* *path in order to receive a message. All first segments of messages along with the path through which any remaining segments may be received are handed to the client's main process when it executes GetFirst.*

GiveUp: PROCEDURE [p: Path];

-- *This procedure must be executed once a client has finished receiving or sending a message. Simply gives up the space occupied by the path for use by another message.*

Abort: PROCEDURE [p: Path];

-- *Aborts the transmission or reception of the message associated with this path. Sends ABORT packet to other end.*

**-- Receiving messages --**

-- Paths don't have to be claimed in order to receive a message since they are handed to the client process with the first buffer. They do have to be given up when the message is completely received.

GetFirst: PROCEDURE [port: Port] RETURNS [p: Path, buffer: DESCRIPTOR FOR ARRAY OF WORD];

-- *Every message is received through a path. All first segments of a message are obtained via this procedure. It returns the first segment of the message and the path through which it was received. All subsequent segments of this message (if any exist) must be received through this path via GetNext (below).*

OkToReceive: PROCEDURE [p: Path, maxbuffers: CARDINAL];

-- *The sending node won't send any subsequent segments until this procedure is executed by the client. SMP will tell the source how many segments it can send (i.e. specifies a window) before it must wait for an acknowledement. The client only needs to execute this procedure once for a single message. After that the SMP receive process will do the acknowledging and requesting for more as the client frees up the buffers that have already been received. SMP will never request from the sender more segments than the number (maxbuffers) specified in this procedure call.*

GetNext: PROCEDURE [p: Path, timeout: ProcessDefs.Ticks] RETURNS [buffer: DESCRIPTOR FOR ARRAY OF WORD];

-- *Returns the next segment buffer of the message being received through the given path. This procedure can't be executed before OkToReceive has been executed unless the client is getting a message that is a response to a message it originally sent, (see FirstDescriptor). This procedure returns a NIL descriptor if called once all buffers have been received. It will signal CommunicationsError if an error in the reception of the message occurs.*

**-- Sending messages --**

-- In order to send a message a client process must first *Claim* a path through which it will be sent. When it is finished sending the message it must *GiveUp* the path (see path management procedures).

<u>FirstDescriptor</u>: PROCEDURE [p: Path, maxbuffers: CARDINAL, awaitresponse: BOOLEAN] RETURNS [buffer: DESCRIPTOR FOR ARRAY OF WORD];

-- *Gets the first buffer in which the data of the message being sent may be placed. The system won't ever let a client have more than "maxbuffers" buffers outstanding at any one time. If the client expects a response to the message it is sending then it must indicate this through a TRUE value for "awaitresponse". SMP will then use the <u>same</u> path it used for sending the request to receive the response. All the client process needs to do is execute GetNext's on the same path until it has received all the segments of the response.*

<u>SendFirst</u>: PROCEDURE [p: Path, messagelength: LONG CARDINAL, bufferlength: CARDINAL];

-- *Sends the first segment of the message. "messagelength" specifies the length of the entire message, "bufferlength" is the length of the first segment.*

<u>NextDescriptor</u>: PROCEDURE [p: Path] RETURNS [buffer: DESCRIPTOR FOR ARRAY OF WORD];

-- *Gets the next buffer in which the data of the message being sent may be placed.*

<u>SendNext</u>: PROCEDURE [p: Path, bufferlength: CARDINAL];

-- *Sends the next segment of the message. This is used for messages that consist of more than one segment. If an unrecoverable error impedes further transmission it signals CommunicationError.*

**-- Status Information --**

<u>MessageLength</u>: PROCEDURE [p: Path] RETURNS [length: LONG CARDINAL];

-- *Returns the length (in bytes) of the entire message being transmitted or received through the given path.*

<u>PathID</u>: PROCEDURE [p: Path] RETURNS [source, destination: User, uid: LONG CARDINAL];

-- *Returns the path identifier of the message being transmitted or received through the given path.*

<u>StateOfMessage</u>: PROCEDURE [p: Path] RETURNS [t: SmpPath.PathType, s: StatusType, e: ErrorType, numSegments: LONG CARDINAL];

-- *Returns certain useful information about the state of the message being transmitted through the given path: whether or not a message is being sent or received, the status, any error messages and how many segments have been transferred.*