ISRN SICS-R--91/14--SE

An Implementation of the Revised Internet Stream Protocol (ST-2)

by

Craig Partridge and Stephen Pink

SICS research report
R91:14
ISSN 0283-3638

An Implementation of the Revised Internet Stream Protocol (ST-2)

Craig Partridge and *Stephen Pink*
Swedish Institute of Computer Science (SICS)
Box 1263
S-164 28 Stockholm, Sweden

Research Report R91:14

Abstract

ST-2 is a revision of an experimental protocol designed to support applications which require guaranteed network services. ST-2 provides mechanisms for creating *streams*, tree-shaped delivery paths with performance guarantees, for applications which require such guarantees. As part of the MultiG project, the authors implemented ST-2 in the BSD UNIX† system. The twin goals of this implementation were to see how easily a novel protocol like ST-2 could be incorporated into the BSD networking model and, more importantly, to thoroughly evaluate the ST-2 protocol. Our conclusions are that the BSD model proved quite general, requiring only modest changes (changes which were largely invisible to the application), but that the ST-2 protocol itself needs some reworking.

1. Introduction

The Internet Stream Protocol [#8] was developed in the late 1970's as an experimental network-layer protocol to support the transmission of voice streams and management of voice conferences over wideband data networks. The Stream Protocol has been used for several years as the primary protocol in a voice and video conferencing system which has operated over the Wideband and Terrestrial Wideband networks [#12,#16]. Based on this experience, the CIP (Connection-oriented IP) Working group of the Internet Engineering Task Force was formed to revise and refine the protocol specification. The goals of this revision were to "fill in some of the areas left unaddressed, to make it easier to implement, and to support a wider range of applications" [#2]. The revised experimental specification was issued in late 1990, as Internet Request for Comment 1190. The revised protocol is commonly known as ST-2.

We became interested in ST-2 as part of our work on the MultiG Project at the Swedish Institute of Computer Science (SICS). One part of the MultiG project is devoted to the support of high-bandwidth, delay-constrained, multimedia applications such as conferencing, virtual environment support, and telepresence. A critical problem is to determine what types of network protocols best support these kinds of applications. ST-2 seemed an interesting candidate for experimentation because it is based on prior experience and contains many features, such as flow reservation and hop-by-hop negotiation, that have been suggested as desirable for gigabit networking protocols [#3,#4].

To experiment with ST-2, we decided to implement it in the 4.3BSD kernel [#11]. We believe that implementing is often the best way of evaluating a proposed protocol because it provides operational experience with the protocol. The implementation process also forces the

† UNIX is a trademark of Bell Laboratories.

implementor to closely evaluate the specification and thus discover subtle issues that a simple reading would not. Another benefit of implementing the protocol was the opportunity to evaluate how well the 4.3BSD networking interface could be adapted to support protocols with performance guarantees such as ST-2.

2. An Overview of ST-2

Abstracted to its basic functionality, ST-2 builds routing trees between a sender (called the *origin*) and one or more receiving applications (called *targets*) and then transmits packets down the routing tree. The origin is the root of the tree and the targets are its children. The internal nodes of the tree are ST-2 routers.[1] Paths between routers, or between routers and the origin or targets, are called *hops*. Systems running the ST-2 protocols are called *agents*.

The data flow over the routing tree is referred to as a *stream*. *Streams* are uni-directional for data. Data can only be sent downstream, from the origin to its targets. When routes to different targets diverge at an agent, the agent duplicates the packet and sends one copy to each hop. Each stream has a flow specification associated with it. The flow specification specifies fourteen properties (such as precedence, reliability, cost constraints, and maximum and minimum packet sizes) that the stream requires of the networks and agents it flows through. Streams can be reconfigured to add or delete targets, change flow properties, or change routes, while data is flowing.

To provide this service, ST-2 is actually divided into two parts, which the ST-2 specification treats as semi-distinct protocols: the STream Protocol (ST) and the ST Control Message Protocol (SCMP). ST simply forwards data through the routing tree. SCMP is the control protocol which actually builds and maintains the routing trees used by ST.

ST-2 is closely tied to the Internet Protocol (IP) [#13,#14]. Beyond a special use of the IP version field to deliver ST datagrams (see the next section), ST-2 uses IP addresses to identify agents. Furthermore, ST-2 provides no mechanism for locating potential targets or learning their addresses: the specification suggests using datagrams sent over IP for target discovery.

2.1. The STream Protocol (ST)

ST provides the most basic of services: the forwarding of data down a pre-established routing tree. The designers of ST want to make data delivery extremely quick and believe that by separating out the data forwarding protocol from the control protocol, they can achieve a very efficient forwarding protocol.

Accordingly, the ST header contains only 8 fields, comprising a total of 8 bytes. The format is shown in Figure 1.

<table>
<tr><td>IV=5</td><td>Ver=2</td><td>Pri</td><td>TS</td><td>0</td><td>Len</td></tr>
<tr><td colspan="5">HID</td><td>Checksum</td></tr>
</table>

Figure 1

The first field is a 4-bit IP version field. The IP version field is an (understandable) hack. Because it is difficult to get protocol identifiers for different MAC layers, ST packets are

[1] Note that if hosts also act as routers, then the internal nodes of the tree may also contain targets.

encapsulated as if they were IP packets but the first 4-bits of the packet contain a special IP version number (version 5), and the receiving IP host is expected to recognize that packets of IP version 5 are to be passed to ST instead of IP.

The next 4-bit field is the ST version number, currently 2. The ST priority class (0 being lowest priority and 7 being highest priority) is to be used to determine which packets within a stream can be dropped if congestion occurs and the stream is using more than its allocated share of resources. The timestamp bit indicates if an optional 8-byte timestamp in Network Time Protocol [#5] format has been appended to the ST header. The length field is the length of the ST packet.

The HID field is the hop identifier. For each stream, each hop has a unique identifier. When an agent receives an ST packet with a particular HID, that HID uniquely identifies a particular stream and how the ST packet should be forwarded. When forwarding, an agent must update the HID field, from the HID of the last hop to the HID of the next hop. The HIDs for each hop are negotiated by SCMP. HIDs below 4 are reserved, and an HID of 0 means the contents of the ST packet is an SCMP message.

The checksum field is the Internet checksum [#6,#7] over the ST header.

2.2. The ST Control Message Protocol (SCMP)

SCMP is one of the more complex protocols the authors have encountered. (The authors' primary experience is with stateful transport protocols such as TCP and RDP, and with the stateless transport protocols, UDP and HMP). SCMP's function is to build, maintain, and, when finished, tear down the routing trees used by ST.

The format of the SCMP header is shown in Figure 2. The header has 11 fields and is 24 bytes long.

<table>
<tr><td>OpCode</td><td>Opt</td><td>Length</td></tr>
<tr><td colspan="2">Recv VLI</td><td>Send VLI</td></tr>
<tr><td colspan="2">Reference</td><td>LinkReference</td></tr>
<tr><td colspan="3">Sender IP Addr</td></tr>
<tr><td colspan="2">Checksum</td><td>Reason/HID</td></tr>
<tr><td colspan="3">Detector IP</td></tr>
</table>

Figure 2

The first field is the 8-bit operation code, which indicates which type of SCMP message is being sent. There are seventeen different SCMP message types. Most messages can only flow one way, either downstream (in the direction from origin to target) or upstream (in the direction from target to origin). There are a few messages, most notably acknowledgement messages, which may flow in either direction. In general, SCMP communications take the form of a request message replied to by a response message, with both request and response messages being reliably transmitted.

The meaning of the bits in the options field are message-type specific. The length field is the total number of bytes in the SCMP message.

The Virtual Link Identifiers (VLIs) can be thought of as SCMP ports. Each HID is typically managed by two SCMP peers (one at each end of the hop), each of which has a unique

VLI. If link-layer multicast is used, an HID may be managed among many peers, one upstream peer and multiple downstream peers on the same sub-network.

The reference field is essentially a sequence number, used to uniquely identify an SCMP message. Acknowledgements of SCMP messages place the reference number of the message being acknowledged into either the link reference field or the reference field depending on the message type. There are two fields for acknowledging reference numbers because some acknowledging messages are also control messages, and must be acknowledged in their turn. For example, an ACCEPT message (described in more detail below) is a message which reliably accepts a connection request. Thus the ACCEPT message must contain both the reference number of the connection request (in the link reference field) *and* its own reference number, so it may be acknowledged.

The Sender field is the IP address of the previous hop in the path this message has followed. The checksum is the Internet checksum over the SCMP message.

The Reason/HID field contains either the HID being negotiated (which appears only in the CONNECT and HID negotiation messages), or a diagnostic reason code, used both for returning error indications and as advice about why changes are being made to the stream. The specification currently specifies sixty-five different reason codes.

The Detector IP field contains the IP address of the system from which the message originated. It is only used for some message types and is null (zero) for the others.

2.2.1. Creating and Using a Stream

To illustrate the general workings of SCMP, we consider a scenario in which an origin creates and uses a stream to one or more targets. The example has purposely been kept simple but contains the essential features of SCMP.

To create a stream, an origin sends a CONNECT message. The message contains the origin's SCMP VLI and a reference ID, plus several parameters. The critical parameters are:

- the Origin parameter, which contains a unique address of the originating application
- the FlowSpec parameter, which specifies fourteen flow parameters. The ST-2 specification does not specify precisely how the agents are to manage their data queues or negotiate with the lower layers to achieve the required flow properties. However, the basic scheme is that each agent establishes a flow that meets the flow specification's requirements as best it can, updates the FlowSpec and passes the FlowSpec to the downstream agent. The final targets each echo the FlowSpec back to the origin, after possibly updating the FlowSpec themselves.
- the Name parameter, which contains a globally and permanently unique name for the stream and is generated by concatenating the origin's IP address with a timestamp. The reason for keeping the Name permanently unique is for accounting purposes.
- the TargetList parameter, which contains the one or more target addresses of the CONNECT message traveling on this hop and target specific source routes. Because SCMP parameters cannot be longer than 256 bytes, and SCMP target addresses can be up to 252 bytes long, multiple TargetLists may be required to store all the target addresses.

In addition, the CONNECT message may propose an HID or invite the next hop to chose an HID. (The possibility of experimenting with some external HID assignment mechanism is also permitted).

One copy of the CONNECT message is sent to each next hop. Thus if there are two targets that are reached via a single next hop, one copy of the CONNECT message containing both target addresses will be sent. The CONNECT message is periodically retransmitted until it is acknowledged by an HID-REJECT, HID-APPROVE, ERROR-IN-REQUEST, ACK, REFUSE, or ACCEPT message, or it has been retransmitted too many times, in which case the origin can send a DISCONNECT message and give up, or try another route.

Upon receiving a CONNECT message, the next hop checks to see if the CONNECT message is for a known stream. If the CONNECT is for a known stream, then the message must be adding new targets to the stream, or be a duplicate, or be an error. If the stream is unknown, then the next hop starts to create a new stream.

To create a new stream, the next hop first allocates an SCMP VLI for the connection, then checks that the HID proposed in the CONNECT message is not already in use. If the HID is in use, the next hop sends back an HID-REJECT message. The HID-REJECT message may optionally contain information about which HIDs are free. Upon receipt of the HID-REJECT message, the previous hop (e.g. the origin) can choose a new HID and send an HID-CHANGE message to continue the negotiation, or decide to give up the connection attempt. If, on further examination of the CONNECT message, the next hop determines that even if an unused HID could be found it would not accept the connection, the next hop can send a REFUSE message to indicate that it will not accept the connection. (SCMP is designed such that it is possible to process the CONNECT message in parallel with the HID negotiation, to speed up connection establishment, so the HID-REJECT may have been sent in parallel with other processing).

If the HID is not in use, the next hop sends an HID-APPROVE message back to the prior hop and continues processing the CONNECT message.[2] Alternately, if the hop is to choose the HID, it picks an HID and returns that HID in an HID-APPROVE message. The next hop examines the FlowSpec parameter and the targets of the CONNECT message to see if it is willing and able to find routes with the given flow requirements to the targets. If the next hop cannot or will not provide a connection to any of the targets, it sends a REFUSE message, containing a list of the targets it cannot or will not connect to. Otherwise, the next hop updates the FlowSpec to reflect the type of service it can provide and sends CONNECT messages on towards hops closer to the targets.

Eventually a copy of the CONNECT message reaches a system on which a target resides. Information about the attempt to connect is then passed to the target application, which decides whether to accept or refuse the connection. To refuse a connection, the target sends a REFUSE message to the previous hop. The REFUSE message includes the address of the target that refused. To accept a connection, the target sends an ACCEPT message to the previous hop. The ACCEPT message contains both the address of the target that is accepting and the FlowSpec parameter after the target's final updates to it (so the origin can confirm that the flow parameters agreed upon with its target are sufficient). The ACCEPT or REFUSE message is retransmitted until the previous hop sends an ACK message. The ACCEPT or REFUSE messages are relayed up the routing tree to the origin, so the origin can know how targets responded to its attempt to connect.

Once the origin has received an ACCEPT message, the stream has been established (at least to the target which accepted) and the origin may begin sending data.

When the origin has finished sending data, it sends a DISCONNECT message. The DISCONNECT message can either contain a TargetList parameter, containing a list of selected targets to disconnect, or a flag can be set in the message to indicate that all targets should be disconnected. The DISCONNECT message is retransmitted until acknowledged by an ACK message from the next hop. The DISCONNECT message is propagated down the routing tree much like a CONNECT message.

A target may drop out of a conversation by issuing a REFUSE message. The REFUSE message contains the target which is dropping out of the conversation. A REFUSE message is retransmitted until acknowledged.

2.2.2. Changing an Existing Stream

A number of SCMP messages are used to change an existing stream.

The origin can issue new CONNECT messages at any time to add new targets to a stream. They are handled much like CONNECT messages at the initial set up time.

Any hop can issue a CHANGE message to change the flow parameters on the part of the tree which is below it. CHANGE messages are handled much like CONNECT requests, except that if the change fails, the affected portion of the stream must be torn down and rebuilt. Hops farther down the tree may ask parent nodes to issue a CHANGE message by sending a CHANGE-REQUEST message.

The HID of a hop may be changed using the HID-CHANGE.

2.2.3. Management Messages

SCMP has four messages used to detect and recover from failures.

HELLO messages are exchanged by ST agents to keep each other informed of their status. HELLO messages are sent on a per-neighbor agent basis, not on a per-stream basis. The HELLO message serves two functions: first, it is used to notify neighbors when an agent is restarted, so they know that streams through that agent have been severed; second, it is supposed to be sent often enough that an agent's crash or a network failure will be detected by the agent's failure to send a HELLO message before any stream's recovery timeout period (a stream parameter) is exceeded.

The STATUS message is used to determine if an adjacent agent knows of a stream. The STATUS message contains either the HID or the stream name of the stream of interest. The queried agent sends a STATUS-RESPONSE message which contains the stream parameters (name, flow parameters, current targets, etc) or an indication that the stream is not known to the agent. STATUS messages are sent when an agent has reason to believe that its upstream or downstream neighbors may no longer know about a stream or may contain information on an old stream which is no longer in use.

NOTIFY messages are used to inform upstream agents of changes that affect a stream, such as routing changes or a change in the resources that are available to a stream (such as the advent of a higher precedence stream that has preempted the existing stream's resources).

2.2.4. Protocol Error Messages

SCMP supports two types of protocol error messages: ERROR-IN-REQUEST messages and ERROR-IN-RESPONSE messages. ERROR-IN-REQUEST messages are used to indicate an error in handling a request message. ERROR-IN-RESPONSE messages are used to indicate an error discovered in handling a response message. The reference field of the SCMP header is used to indicate the particular request or response found in error. The reason field of the SCMP header indicates the particular error found. The message may also contain the contents of the SCMP request or response which was found in error, with a pointer to the location where the error was discovered. Error messages are not sent in response to error messages.

Error messages have local (per-hop) significance. They are used to indicate cases where a protocol failure (a bad message format or failure to properly follow the protocol) has been detected.

The sender of the erroneous message is encouraged to retransmit the (presumably corrected) message when it receives an error message. The mechanism by which the sender is to determine how the message is to be corrected is not specified.

2.2.5. Special Functions

ST-2 supports features to make use of multicasting, if the network supports it. Multicasting is used to optimize delivery to multiple next hop agents by sending a single multicast packet. ST-2 also allows systems to request that both a forward and return ST-2 connection be established, to permit duplex communication.

ST-2 further allows ST to be encapsulated within IP, so that ST can be experimented with over IP networks.

3. The Structure of the BSD Networking Code

The BSD networking code has a well-tested structure intended to support multiple families of networking protocols and to make the addition of new protocols easy. The code has been extensively tested and refined by the implementation of several protocol suites, including the TCP-IP, OSI, and XEROX-PUP protocol suites. This generality of the BSD code made it the implementation system of choice for the ST-2 implementation and we were interested in seeing how well a multimedia protocol was served by the BSD code.

This section provides a brief overview of the BSD code, as used by connection-oriented protocols. (ST-2 was implemented as a connection-oriented protocol). Connection-oriented protocols are those protocols which require a connection to be established between peers before any data is sent. The BSD code also has extensive support for connection-less or datagram protocols. Readers interested in a more thorough discussion of the BSD code may consult [#11].

3.1. Application Interface

The BSD system provides a generic set of networking system calls for use by all applications, using any protocol suite. One interesting issue in implementing ST-2 was to see how well this generic interface could support ST-2 (and by implication, protocols with performance guarantees in general).

The key concept underlying the BSD networking code is that of a *socket.* A socket is a UNIX file descriptor which links an application to a particular networking protocol. Once a socket has been created, an application makes a series of system calls upon the socket to configure and establish the networking service the application desires.

To create a socket, an application calls the *socket* system call, specifying the protocol family (e.g. Internet, Xerox-Pup, OSI), type of service (e.g. byte-stream or datagram), and optionally, the particular protocol desired. If successful, the *socket* call returns a file descriptor (a positive integer) which provides a link to a largely uninitialized protocol structure.

Applications use a series of system calls to initialize the socket. The choice of system calls used typically depends on the type of application which is using the socket.

Most applications try to actively connect themselves to a remote server, which provides some service the application requires. To make the connection, the application calls the *connect* system call. Given an address structure containing the address of the server, the *connect* call attempts to create a connection to the remote server. If the *connect* call succeeds, the socket is then ready for transmitting and receiving data. Applications transmit data using the *write* or the *send* system calls (which are variants of each other). Applications receive data using the *read* or *recv* system calls (which are also variants). When an application is finished using a socket, the socket is closed using the *close* system call, and the connection will be severed.

Server applications, which are typically waiting at some well-known address for requests from other applications, usually call the *bind* system call. The *bind* system call is given an address structure and makes the provided address the local address of the socket. After calling *bind*, the application usually calls the *listen* system call to indicate to the kernel that the application is awaiting attempts to connect to it. Connection attempts are then accepted using the *accept* system call, which returns a new file descriptor linked to the newly created connection. The application can then communicate with the new peer via the new descriptor using the *read* and *write* system calls.

As commonly used and described above, the interface is quite simple. Applications call *socket* followed by *connect* and are then free to send and receive data. Servers call *socket*, followed by *bind* and *listen* and then use *accept* to service new connections. However, there are often some protocol specific options that an application or server may wish to request. The *setsockopt* and *getsockopt* system calls may be used to set or read options for a particular socket. Also, a server may wish to read from multiple connections at once. The *select* allows an application to check on the status of its file descriptors and learn which ones have data to be read or are ready for writing.

3.2. Kernel Protocol Tables

Inside the BSD kernel, most interaction with a protocol is handled through a *protocol switch table.* Each family of protocols (TCP/IP, OSI, etc.) has its own protocol switch table, which is a list of switch entry points, one set of entry points for each protocol in the protocol family. Implementing a protocol under the BSD kernel is largely a matter of writing the functions that implement the various entry points.

Each switch entry point contains pointers to nine functions (some of which do not need to be defined), plus some protocol flags.

Two routines are used to support the application interface. The *pr_ctloutput* routine is used to handle calls to *getsockopt* and *setsockopt*. All other system calls invoke the *pr_usrreq* (''user request'') routine, with an operation code indicating which service is required. These routines must be defined.

Inbound traffic from the network is handled by two other routines. For transport protocols, the *pr_input* routine receives all inbound traffic from the network layer. (Network layer protocols are called from the interface driver via a software interrupt without using the switch table). The *pr_ctlinput* routine is used to notify protocols of network errors; the *pr_ctlinput* routine is expected to take the necessary actions required by its protocol to respond to the error.

The *pr_output* routine is used to send datagrams. It is only used in special cases; most protocols (including our implementation of ST-2) send data by calling *pr_usrreq*.

The remaining four routines serve general protocol maintenance functions. The *pr_init* routine is called when the system first boots, to initialize the protocol. The *pr_drain* function (usually not implemented) is called by the system when it is short of space, asking the protocol to try to free any memory it can. The *pr_fasttimo* and *pr_slowtimo* routines are the protocol timeout routines and are called by the kernel at regular intervals. The fast timer is called five times per second and the slow timer twice a second.

The protocol flags in the switch entry are used to flag properties of the protocol to the socket layer (which sits between the system calls and the protocol routines). The important flags are:

flag	**meaning**
PR_ATOMIC	protocol only exchanges atomic messages
PR_ADDR	application gets the source's address with messages
PR_CONNREQUIRED	a connection must be made before sending data
PR_RIGHTS	a mechanism for passing additional significant information

In theory, any combination of these flags is meaningful. Our implementation uses PR_ATOMIC, PR_ADDR, and PR_CONNREQUIRED.

4. ST-2 Services Not Implemented

The implementation does not completely implement ST-2. For various reasons, a few things have been left out.

While the code was designed so that it would be possible to make the BSD kernel act as an ST router (i.e. an intermediate agent between two networks), the forwarding code is not complete and has never been tested.

We also chose not to implement network-level multicasting.[3] The standard BSD code does not support network-level multicasting and while modifications to the BSD code are available to implement multicasting, multicasting is a performance enhancement to ST-2 rather than

[3] Note that multicasting should not be confused with support for multiple targets, which the implementation does support.

a critical feature and we concluded it was not important to the experiment. (We did, however, structure the implementation such that adding multicast support should be easy). For similar reasons, we did not implement the point-to-point option nor the reverse flow option.

A more serious omission is support for HID-CHANGE messages. Recall that HID-CHANGE messages are used to negotiate a valid HID after the HID proposed in the CONNECT message has been rejected. We left HID-CHANGE out because we believe it is an unnecessarily complex mechanism (see sections 5.2.2 and 5.2.8 below), however its omission means our implementation will not be compatible with other implementations if the sender initiates negotiation of the HID (as our implementation does).

Another omission is support for HELLO, STATUS and (in part) NOTIFY messages. As described in section 5.2.5 we believe that supporting all three message may be unnecessary.

The implementation does not support timestamps because timestamp negotiation does not work.

4.1. Experience with the Implementation

Since implementing the protocol, SICS has experimented with running the Network Voice Protocol (NVP) and the Packet Video Protocol (PVP) over the ST-2 implementation [#20,#21]. These experiments have successfully demonstrated real-time delivery of data via ST-2 over a local Ethernet and via the Internet between London and Stockholm.[4]

5. The ST-2 Implementation

This section briefly describes our ST-2 implementation, with a focus on important implementation issues and places where the BSD code posed problems or challenges to the implementor.

5.1. ST-2 and the BSD Application Interface

The first problem in implementing ST-2 was to decide how to fit the protocol under the BSD socket interface. This problem was quite difficult for a variety of reasons.

5.1.1. Which Version of the Application Interface to Use?

One issue was that the BSD application interface is designed to serve as an interface over transport-layer protocols. The address structures are assumed to be transport-layer addresses and the system calls are designed on the assumption that applications are reading and writing data without protocol headers. To access network-layer or lower-layer protocols, the BSD application interface supports a special type of socket, called a *raw* socket, to allow applications to read and write datagrams, including protocol headers. Since ST-2 is a network-layer protocol, one might consider using the *raw* socket mechanism to support ST-2.

Unfortunately, however, the *raw* socket mechanism is largely stateless. It is intended as a pass-through to allow applications to implement special transport protocols over connection-

[4] Note that the ST-2 connections could not allocate a guaranteed flow over the Ethernet or the Internet and simply assumed the bandwidth would be sufficient. (It wasn't quite enough for London-Stockholm, so the sound quality was poor). We would like to thank Information Sciences Institute for making their PVP and NVP code available to us for experimentation.

less lower-layer protocols and does not normally support features such as retransmission and error recovery. But ST-2 requires retransmission and error recovery, and thus seems better suited for the full application interface. Perhaps we could have implemented the stateful part of ST-2 in user space over a raw socket interface but this seemed unwieldy and we decided to implement ST-2 as a connection-oriented protocol, directly under the regular application interface. (Interestingly enough, SUN Microsystems appears to have made the same choice when implementing X.25 [#15]).

5.1.2. Problems With the Application Interface

Implementing ST-2 as a connection-oriented protocol under the BSD application interface was still not easy. We discovered two major problems and one minor problem with the BSD application interface.

5.1.2.1. Maximum Address Size

The minor problem was that the BSD interface limits the maximum size of an address to be 112 bytes long, while ST-2 permits addresses of up to 252 bytes. After some thought, we decided to enforce the BSD limit, as the ST-2 address space is probably far larger than is necessary.[5]

5.1.2.2. Semantics of Accept

A more important problem was that ST-2 expects the application to be able to choose whether to accept or reject a connection attempt and to examine and possibly modify the flow specification before accepting the connection. The BSD interface forces the application to always accept a connection attempt, via the *accept* system call, before it knows who the connecting peer is or any properties of the flow.[6] However, once the accept system call has been made, the application can immediately terminate connections from unwanted peers, so while the BSD semantics are not quite the same as those of ST-2, the provided functionality was close enough that we decided not to modify the BSD interface.

5.1.2.3. Semantics of Connect

The major problem was that the BSD kernel enforces a rule that a socket may only be connected to one remote address at a time. This rule is clearly inappropriate for multi-target protocols, where a single socket may have multiple targets. Fixing the one-address assumption turned out to cause multiple minor problems, each of which had to be solved.

First, the *connect* system call had to be modified to permit multiple calls on the same socket. In the standard BSD code, sockets which must be connected before data can be sent (which is true for ST-2) are flagged as PR_CONNREQUIRED in the switch entry. While the

---

[5] This feature of BSD has been changed in 4.3 BSD, Networking release 2 (essentially the networking code for 4.4BSD). Addresses can be of unlimited size [#22]. However, if one is trying to write code portable to BSD and its earlier variants, the 4.4BSD changes are not yet widespread enough to base the implementation on.

[6] Again, this feature of BSD has been changed in 4.3 BSD, Networking release 2. To accomodate the OSI protocols, the code has been restructured to defer connection acceptance until a subsequent system call (e.g. *recv*, *send* or *close* is made), thus giving the application time to consider the request [#22] and examine and update the flow specification using *setsockopt* and *getsockopt*.

documentation suggests that PR_CONNREQUIRED simply states that a connection is required before sending data, the *connect* system call allows only one call to *connect* for sockets of this type. One option was to change *connect* to take multiple addresses as its arguments, but multiple addresses in one call makes error returns ambiguous. (*Connect* only returns a single error code; how is the application to know to which address the code applies?). Instead, we changed *connect* to permit multiple concurrent calls. An an additional flag, PR_MULTICONNECT, was added to the protocol switch table, which indicates that even if a connection is required, multiple calls to *connect* are acceptable.

Modifying the *connect* system call to allow multiple calls created a new problem. In UNIX, duplicate copies of a file descriptor may be shared. As a result, after a socket has been created, copies of it may be given to multiple applications, all of which may call *connect* on the socket. When *connect* only permitted one call, this scenario was harmless. The first call made the connection and the subsequent calls failed. But if multiple *connect* calls are permitted, then the *connect* call has to support multiple concurrent calls on the same socket; in other words, it must be multi-threaded. So we enhanced the code path in *connect* to support multi-threading.

Note that one potential problem with the BSD *connect* interface is that it forces the application to add new peers one connect call at a time. It has been suggested by one of the ST-2 designers that this may have undesirable interactions with routing. The question is whether a routing system can better choose the appropriate routes for a stream if it has complete knowledge of all targets when the stream is first being established. We do not know the answer to this question.

5.1.2.4. Need for a Disconnect System Call

Finally, given that an application could connect to multiple remote addresses, it had to be possible for applications to disconnect from selected addresses. In fact, the BSD kernel has support internally for disconnecting particular remote addresses, but lacked a system call to allow applications to access the internal code. So we added a *disconnect* system call. (Adding new system calls is simple in UNIX systems, and most binary distributions of UNIX permit easy installation of new system calls).

5.2. ST-2 in the BSD Kernel

Once its application interface is defined, implementing a protocol in the BSD kernel is largely a matter of defining the basic data structures and then writing the various entry-point routines for the protocol switch table, and their subroutines. This code can be quite complex. For example, the ST-2 implementation is over 6,500 lines of C code.[7]

5.2.1. Data Structures

The major data structures in the ST-2 implementation are the protocol control blocks (PCBs) and the timer control structures.

---

[7] In contrast, the 4.3BSD Reno distribution implements TCP in about 3,900 lines of C code and a BSD implementation of RDP (a protocol of complexity similar to TCP) required 2,700 lines of C code.

For every ST-2 stream, the kernel builds a two-level tree data structure. At the root of the tree is a *stream* PCB. Its children are *hop* PCBs. Data flows down the tree: when an application writes data to a stream or a packet arrives for a stream, the data is forwarded to all the hops for that stream.

The stream PCB contains all global stream information (such as the stream's name and origin) and all the information necessary to manage the upstream peer, which is either an application (if the stream originates locally) or the previous ST hop. This information includes the socket pointer if the upstream peer is an application, or a collection of ST-2 state information if the upstream peer is another system.

The hop PCB contains all the information necessary to send data downstream to the next hop. Note that the next "hop" is in some cases an application; the hop PCB can point to a socket instead of a next hop address.

One complicating feature of ST-2 is that a hop can be in multiple states at once. For example, a hop can be connected to its remote peer and ready to send data, while at the same time the hop is negotiating with its peer to add additional targets or to change the flow specification. Trying to track multiple states in a single hop PCB proved quite confusing, so the implementation uses *shadow* hop PCBs. For each hop, there is one primary PCB, through which all data is forwarded, and zero or more shadow PCBs. Each shadow PCB manages a particular SCMP control exchange such as the addition of a new target or the negotiation of a changed flow specification. Once the negotiation is completed, the results are folded back into the primary hop PCB and the shadow is deleted.

The *timer* structure is used to manage SCMP retransmissions. For each SCMP message, a separate timer structure is created which manages the retransmission of the message. If the message is acknowledged, the timer is turned off. If the message times out, the message is resent and the timer is backed off. If the message times out several times, the attempts to retransmit are viewed as having failed and the affected part of the ST-2 stream is disconnected. Round-trip time estimation is done using Karn's and Jacobson's algorithms [#9,#10].

Our implementation organizes the PCBs in doubly-linked lists. We originally tried to use arrays indexed by the HID as suggested by the specification (p. 58), but one of our test systems (a SUN 3/60 running MACH 2.5 with the BSD networking code) ran out of memory with tables much over 32,000 entries so we used linked-lists instead.[8]

5.2.2. The pr_usrreq Routine

The *pr_usrreq* routine is essentially a large switch, to handle 21 different types of requests. (Some BSD system calls cause multiple requests to be made of the *pr_usrreq* routine).

The major question when writing the pr_usrreq routine is whether to try to be fast and avoid extra subroutine calls, or write modular code and use subroutine calls to handle each type of request. We chose clarity over speed, and implemented every request as a call to a subroutine. Implementing the subroutines was less of a chore than it sounds, as only 11 types of requests are actually relevant to ST-2 (the rest are no-ops or errors).

[8] The problem was that the kernel memory space began to overlap with frame buffer memory, which caused strange errors. The table was simply an array of pointers, so the size was about 128 Kbytes.

5.2.3. The Output Routines

Most calls to ST-2's *pr_usrreq* routine eventually cause an SCMP message to be sent. The various SCMP messages are very different in the information they carry, so we chose to write a separate routine to generate each different SCMP message type. In general, this approach worked very well. It was, for example, easy to set debugging break points to trace the sending of a particular message type. The one nuisance about this approach was that there was a lot of replicated code in the SCMP message routines and when a bug was found in the replicated code, one had to be sure to fix every instance of the bug.

The SCMP routines all call a single ST output routine, *st_output1*, which sends an ST packet to a single destination.

When data is sent on a socket, the routine *st_outputall* is called, which then repeatedly calls *st_output1* to send a copy of the packet to each of the downstream hops.

5.2.4. The Input Routine

To make input work for ST-2, we had to modify the IP input routine to recognize IP packets with an IP version of 5 as ST packets and call the ST input routine, *st_input.*

St_input confirms that the ST packet is valid and for a known HID. If there are any problems, the packet is discarded. If the HID is 0, the packet is passed to the SCMP input routine, *sc_input*, otherwise the packet is forwarded to the next hop or delivered to the local target (as appropriate).

The *sc_input* routine checks that the SCMP header is valid and then calls the input routine for the particular SCMP message type. As with SCMP output and for the same reasons, we chose to implement a separate input routine for each SCMP message type.

5.2.5. ST-2 over IP

In addition to being used as a native network-layer protocol, ST-2 can also be encapsulated in IP. The implementation supports ST-2 over IP mode as transparently as possible.

Sending ST-2 packets over IP is supported by use of a dummy network interface. Any data given to the dummy interface's output routine is simply encapsulated in an IP datagram and sent via IP. If a sending application requests ST-2 over IP mode using *setsockopt*, the *st_ctloutput* routine simply adjusts all PCBs to use the dummy interface. This indirection is transparent to all the SCMP and ST-2 transmission routines.

Receiving ST-2 packets over IP is a bit harder. A transport-level entry point had to be defined for ST-2, so that IP could call ST-2 as a transport layer. This routine strips off the IP header and calls *st_input*, with a flag that the ST-2 packet came in over IP. *St_input* must be informed which mode an ST-2 packet arrived in because the message may be an SCMP message that requires an acknowledgement. SCMP must know whether to send the acknowledgement or error message back via native ST-2 or IP. So when the ST-2 transport layer routine is called, it calls *st_input* with a flag to indicate that the ST-2 packet arrived via IP and this flag has to be passed up to all the ST-2 and SCMP input routines.

Note that while the ST-2 and SCMP code must keep track of whether packets arrive via native ST-2 or IP, this information is hidden from receiving applications. When an application listens on a socket, it will receive all incoming requests via either ST-2 or IP. If curious, an application can call *getsockopt* to learn whether the particular ST-2 connection is running over

IP.

5.2.6. Managing the Flow Specification

SCMP assumes that there is some entity, possibly external to the SCMP protocol itself, which manages the availability of resources. SCMP passes the flow specification to this entity to request stream resources. The entity is responsible for establishing a flow (if possible) consistent with the flow specification and updating the flow specification to reflect the stream properties achieved.

In the BSD code, we felt the most logical place for this entity to reside was in the per-network interface code. The interface code is the place where most network-specific information is kept and thus it seems logical to negotiate with the interface code for flow specifications.

Like interactions with the network layer, interactions with the interface drivers are managed through a fixed suite of five routines: *if_init* is used to initialize the device; *if_output* is used to send data; *if_ioctl* is used to serve I/O control requests; *if_reset* is used to reset the interface; and *if_watchdog* is a timer routine, usually used to catch lost interrupts. In addition, input from the interface is managed by calling the per-network-layer specific input routines. Various versions of the BSD code add some additional routines to these six, but this is the common set of routines.

Our implementation assumes that negotiation of flow properties is handled through the *if_ioctl* and *if_output* routines. When a stream is first being established at an upstream agent, the agent establishs a flow by calling the *if_ioctl* routine with the flow specification. The *if_ioctl* routine establishes the flow (if possible) and returns the update flow specification along with a flow identifier. This flow identifier is then passed to *if_output* on every packet transmission. If the flow fails, *if_output* can return an error code that indicates the flow has failed. (Note that while code for this flow negotiation exists in our implementation, it hasn't been tested because we did not have access to networks that supported guaranteed flows).

5.2.7. Problems with the BSD Routing Code

ST-2 expects that an agent will be able to consider and potentially try to use all the possible routes to a destination. This expectation implies that the system routing tables must be able to return a list of all possible routes to a destination (or provide some mechanism for learning of alternate routes). The BSD routing code only returns a single route, with no mechanism for finding alternate routes.

6. Evaluating ST-2

When evaluating ST-2 we found it useful to structure our thinking around three distinct questions:

- *Is the ST-2 philosophy appealing?* ST-2 is the reflection of a particular philosophy about data communication; summarized in brief, this philosophy can be characterized as arguing that certain types of applications such as multimedia applications require networks that can establish guaranteed data flows. The ST-2 protocol specification is, in part, an attempt by researchers to express a particular view about how this philosophy should be implemented. It is useful to consider some of the philosophical decisions that were made.

- *Can ST-2 be simplified and still provide the same services?* ST-2 was defined as a vehicle for research in networking support for multi-destination isochronous applications. Based on our experience, could ST-2 be improved to serve this function better?
- *Is the ST-2 specification well-written?* Is the protocol specification clear and largely free from errors and ambiguities? In short, can one be sure that if one understands the specification, one understands the protocol?

This section discusses each question in turn.

6.1. ST-2 Philosophy

ST-2 was designed by a group of researchers who believe that to support applications with strict performance requirements it is necessary to install state information on all nodes in the path from the data source to the data sink. Whether these applications require the installation of state information, and if so, how much state information must be maintained is currently a subject of hot debate in the networking community.

One of our goals in implementing ST-2 was to try to get a better insight into this debate. While the problems with the specification made a proper evaluation of the ST-2 philosophy difficult, we did encounter interesting philosophical issues during the implementation and continue to feel that the general ST-2 philosophy is intriguing. This section discusses our views about some of the more philosophical features of ST-2.

6.1.1. Nice Features of ST-2

There were two properties of ST-2 that we particularly liked.

6.1.1.1. Negotiation of Flow Properties

One nice idea was the principle that each agent modifies the flow specification to reflect the flow properties the agent could actually promise with the final flow specification being returned to the origin for a final decision about whether the properties achieved were sufficient. We see at least two advantages to this approach. First, in cases where the original flow specification could not be met, the origin gets a more useful reply than a yes-no answer. The origin can decide if the delivered flow seems sufficient, with some confidence that if the flow is not sufficient, it is probably still all the network can offer. Second, the ST-2 mechanism eliminates possibly extended negotiation over flow parameters at each hop. This feature would seem likely to reduce the time necessary to establish a flow.

6.1.1.2. Targets Specified by Origin

We also believe that allowing the origin to specify its targets is a useful feature. The origin can better control who its data reaches and how that data is delivered. One question about this scheme, however, is whether the list of targets could become too large for the origin to manage.

6.1.2. Less Pleasant Aspects

There were a couple of philosophical decisions about ST-2 that we found ourselves disagreeing with. Problems with the strict separation of data and control information appeared

during the implementation. And we have questions about the design of the flow specification.

6.1.2.1. Separation of Data and Control

ST-2 almost completely separates data and control information. Other than the HID, ST packets contain no information about where they came from, where they are headed, or the type of traffic they contain. All such information is negotiated via SCMP. The reason for keeping all state information out of ST was to keep the ST headers small, with the expectation that smaller headers would be faster to process in the ST agents.

The implementation experience suggests that ST-2 may separate data and control information a bit too much. In particular, it is more difficult to recover from failures. For example, if an ST agent crashes and reboots, it may start receiving ST packets for the streams that it previously supported. Unfortunately, because the ST packets contain only an HID, it is impossible for the ST agent to determine which upstream agents need to be notified of its failure. Eventually a HELLO message should timeout and turn off the flood of packets but if by some chance, the HELLO mechanism fails, the consequences can be painful. (While debugging our code without HELLO support, we hit a situation in which one of our rebooting systems was so flooded with unwanted ST packets it could not reboot!)

The key problem with complete separation of control and data is that either the control or the data path may fail while the other path is still working, leading to odd failures. For example, at one time, our implementation had a bug which caused it to fail to properly acknowledge ACCEPT messages. The result was that the origin would receive the ACCEPT message, establish the connection and begin sending. The target would start receiving and reading the data (since the target assumes the connection is open once it sends the ACCEPT). But after several retransmissions, the ACCEPT message would time out and SCMP would shutdown the connection, on the assumption that a protocol failure had occurred.[9]

To avoid these scenarios, it probably makes sense to carry a modest amount of control state information in every data packet. (Our view is that if an agent receives a ST packet in error, the packet should contain enough information that the agent can take some action to try to prevent further errors from occuring). In ST-2's case, we think that adding the IP address of the previous hop, and perhaps reference numbers of the most recently seen SCMP control messages is sufficient. (Note that the source link-layer address of the previous hop would also be sufficient).

6.1.2.2. Specifying Flow Properties

The ST-2 Flow Specification is extremely detailed, ennumerating fourteen different performance parameters. Furthermore, the specification permits applications to add additional fields to the flow specification if desired. The reason for all the detail was to permit researchers to experiment with a wide variety of parameters. This idea is consistent with some exploratory papers on flow management [#23,#24].

---

[9] Observe that the obvious fix to this problem (to turn off the ACCEPT retransmissions when the data starts arriving at the target) doesn't work. If there is an existing target at a system, and the origin adds a second target at that system, data may be flowing down the stream for the existing target even if the ACCEPT message for the new target never gets through. One needs some confirmation that an individual target's ACCEPT message has been seen.

For a few reasons, we find ourselves wondering if a large flow specification is still the right approach. In particular, we believe that it is more effective to try to identify the minimum number of parameters necessary to provide the required range of guarantees, and test that set, rather than provide an all-one-could-want experimental space. (ST-2 makes it possible to extend the Flow Specification, so if the minimum set proves too small, one can still add new variables).

First, while the large number of variables gives large opportunity to experiment, it also can hinder experimentation. A large number of variables means there is a large state space of factors to study. Performance analysts typically recommend working with a selected small set of factors [#17]. (To see how problems might arise, consider two ST-2 experiments in flow allocation techniques, each of which only used three variables in the flow specification to specify the flow and ignored the other eleven. If the two experiments use disjoint sets of variables, it is not clear how to compare the results of the experiments).

Second, a smaller set of parameters makes it more likely that a particular network has the mechanisms to support the particular type of flow desired. Since ST-2 is designed as an internetworking protocol, maximizing the types of networks it can operate over would appear desireable.

Third, there is now some paper and pencil analysis that suggests that guaranteed multimedia flows can be specified using only two or three parameters (delay, bandwidth, and possibly, interpacket delay variance) [#18,#19]. While much of this work has been done since the ST-2 specification was produced, it does suggest that an experiment using a very small set of variables is appropriate.

6.2. Can ST-2 Be Improved?

Based on our implementation experience, we believe that the ST-2 protocol and in particular, SCMP, could be made considerably simpler and still both provide the same functionality and be a useful vehicle for research. This section suggests some ways in which the protocol could be made simpler.

6.2.1. A Minor Nuisance in Handling Error PDUs

The specification provides a method for returning part of the SCMP message that is in error in ERROR-IN-REQUEST and ERROR-IN-RESPONSE messages. A portion of the SCMP message that was in error plus a pointer to where in the message the error was encountered are encapsulated in an SCMP parameter.

Returning the message found to be in error is useful, but requiring a pointer to where the error was encountered proved a nuisance. Keeping the pointer requires that all the SCMP input code, through all its subroutines, be able to track offsets into any SCMP message being handled, on the chance the message may turn out to have an error. Furthermore, there is often ambiguity about where an error occurred. For example, if a parameter is shorter than its length field indicates, is the pointer to point to the parameter's length field, or to the last byte of the too-short parameter? (Our implementation realizes the problem only at the last byte of the undersized parameter). Finally, what about errors where the SCMP message is inconsistent with local state information? For example, if the HID and the stream Name are inconsistent, is this an error in the HID field, the Name parameter, or should it be considered a state inconsistency in the agent's code (and if it is a state inconsistency, what does one put in the pointer

field?).

While returning the message in error is useful, the pointer to the place the error was discovered is of limited value and a modest nuisance to the implementor. Given that this parameter is optional, this nuisance is not severe (one can just not return the parameter). However, in the interests of encouraging the return of error information the pointer should probably be eliminated.

6.2.2. HID Negotiation at Connection Time

When an agent sends an SCMP CONNECT message with an HID and the next hop refuses the message with an HID-REJECT, the next hop must still process the CONNECT message and the connecting agent is expected to continue negotiation with an HID-CHANGE message.

This procedure seems unnecessarily cumbersome. It requires the next hop to try to process a CONNECT message and begin to build data structures, even though a key piece of information, namely the proper HID, is missing. (It may be difficult to install the ST forwarding information without the HID, because the HID is the primary lookup key for ST). It causes the next hop to support two different code paths for processing a CONNECT message: a path for handling CONNECT messages with good HIDs, and a path for handling CONNECT messages with bad HIDs followed by HID-CHANGE messages. It would seem far simpler to just have the connecting agent resend the CONNECT messages with a new HID.

6.2.3. Useless TargetList Parameter

SCMP messages contain a parameter called a TargetList, which contains the addresses of the targets that the message pertains to.

The goal of the TargetList is to consolidate all the target information in one place. Unfortunately, the TargetList parameter is defined as having a maximum length of 256 bytes, while the largest target can be 252 bytes. Thus if a message has multiple Targets, more than one TargetList parameter may be required to transmit them all. The ST-2 specification (p. 96) recognizes this problem and notes "an implementation should be prepared to accept multiple TargetLists in a single message."

Clearly this feature is unnecessary. If an implementation must be able to read multiple TargetLists in one message, it could just as well read multiple target parameters, unencumbered by the extra layer of wrapping. As currently specified, the TargetList parameter is extraneous.

6.2.4. Many and Unhelpful Reason Codes

The purpose of SCMP reason codes is to convey information about the reason for an error message, or for modifying a stream (e.g. with a REFUSE or DISCONNECT message). Unfortunately, the system of reason codes is inadequate to this task.

First, the list of reason codes attempts to be exhaustive. It currently includes sixty-five different codes, including codes noting that oversized UserData parameters have been encountered (code 64) and that a control PDU contains a duplicate target (code 23). Unfortunately, despite its length, the list is not comprehensive; our implementation includes several cases where no reason code matched the error encountered. (For example, there are no error codes for for the following situations: problems with IP make it impossible to set up an ST over IP

connection; one of the VLIs is in error; or the target in an ACCEPT message is not one of the targets requested by the origin). Furthermore, the vast majority of reason codes do not seem to apply: our implementation uses less than a third of the codes defined. We suspect that the current list of reason codes reflects particular implementations and that as new implementations are done, new reason codes will have to be added to the list. In short, the notion of a comprehensive set of reason codes would appear to be an "m×n" protocol feature.

Second, while the specification states that reason codes are to be used to disambiguate fatal conditions from more benign errors, the specification does not indicate which codes are fatal. Nor are any bits in the error codes clearly reserved to indicate fatal versus non-fatal errors. This means that if different implementors interpret the same reason codes differently (e.g. one views a reason as benign, the other as fatal), surprising protocol interactions may occur. For example, the specification generally encourages the immediate retransmission of messages in response to an error message, when possible. If the agent sending the error message believes the reason code indicates no retransmission is possible, while the receiving agent believes the error code is benign, a loop of error messages and retransmissions will occur.

A better reason code scheme would define the reason code field to be a bitmap, with bits reserved to indicate fatal and benign error codes, and perhaps a few general error codes to indicate roughly how an error occurred.[10]

6.2.5. Timestamp Negotiation Doesn't Work

There is a mechanism in SCMP to negotiate whether timestamps should be included in ST packets. As defined, the negotiation scheme doesn't work.

An SCMP CONNECT message sends a two-bit timestamp proposal to the target. The meanings of the bit values are:

bits	meaning
00	no proposal.
01	cannot insert timestamps.
10	must always insert timestamps.
11	can insert timestamps if requested.

The ACCEPT message contains a two-bit reply to the timestamp proposal. The interpretation of the two bits are:

bits	meaning
00	not implemented.
01	no timestamps are permitted
10	timestamps must always be present
11	timestamps may optionally be present

Observe that the following non-working scenarios are possible[11]:

[10] We understand that the committee designing ST-2 considered using a fatal bit but decided not to. Given hindsight, we believe the bit is necessary.

[11] Conversations with the designers indicate that timestamp negotiation wasn't intended to work in all cases -- the idea was that systems could experiment with timestamps and that if a non-working scenario was encountered, the stream would be torn down. Unfortunately, the specification does not indicate that

- CONNECT message proposes "must always insert timestamp." ACCEPT message replies "no timestamps are permitted" or "not implemented."
- CONNECT message proposes "cannot insert timestamp." ACCEPT message replies "timestamps must always be present."

The specification offers no guidance on how to resolve these conflicts. Furthermore, it is not clear what is to be done if, in response to a CONNECT message offering "can insert timestamps if requested," some targets request timestamps and others decline timestamps. Perhaps intermediate agents could remove timestamps for those targets that do not wish them but this solution would make agent processing more complex.

There are a couple of possible solutions. A completely interoperable timestamp scheme could be used. A one bit scheme in which the origin offers to send timestamps and the targets indicate if they can accept timestamps would work. Alternately, a one bit scheme in which the origin insists on sending timestamps and targets which cannot accept timestamps must refuse the connection, would also work. The current scheme, however, is much less than optimal.

We also wonder if timestamps are a useful or necessary service for ST to provide. Since timestamps are required on a per-application basis, perhaps the applications should implement their own timestamping schemes?

6.2.6. HELLO Message Unnecessary?

SCMP has four different mechanisms for detecting various types of problems in a stream or in the communication between ST agents. There are STATUS messages (to check on the state of a stream), NOTIFY messages (to advise neighboring agents of problems such as routing loops) and HELLO messages (used to periodically confirm that neighboring agents are up and to notify neighboring agents when an agent restarts). Furthermore, there is information from the ST data stream itselfi; the arrival of an ST packet is a fairly strong indication that the upstream agent is still functioning.

We believe that it should be possible to reduce the number of failure recovery and notification mechanisms. The most tempting candidate for elimination is the HELLO message.

HELLO uses a ping mechanism. Agents regularly exchange HELLO messages to confirm that neighboring agents are still up. However, in most cases, all neighborying agents will be up, so the HELLO messages are simply consuming bandwidth. For this reason (and others), some protocol suites strongly discourage pinging between non-router systems (see, for example, page 52 of [#14]).

Another concern is that the HELLO mechanism requires agents to track peer agents separately from streams. While not a particularly onerous task, tracking peer agents can imply more data structures, more timers and more complexity.

Finally, it appears that the function of the HELLO message is redundant. Receipt of a valid ST packet from another agent is implicit confirmation that the agent that sent the ST packet is functioning. If an agent has not sent an ST packet in a reasonable period, a STATUS message for any of the streams through that agent will confirm the agent is still up. When an agent reboots, it can notify the upstream agents of the loss of state when it receives ST packets

---

timestamp negotiation was not intended to work in all cases, nor does it say what to do when the negotiation fails.

for unknown streams.[12]

In summary, there appear to be an excess of failure recovery and notification mechanisms in ST-2. We suggest that at least one of those mechanisms be eliminated and suggest the HELLO message is the logical candidate for deletion.

This reasoning leads us to believe the HELLO message could be eliminated.

6.2.7. Multiple Addresses for the Same Stream

In ST-2, a stream can be identified three different ways: by its unique name, by the HID of a particular hop, or by the SCMP sender and receiver VLIs. That is one too many addresses for the same stream. Because the stream name is designed to be globally unique, it is arguably too big to use as a primary address, so a second, simpler address such as an HID is useful. However, the HID and the VLIs uniquely identify the same hop in the same stream, so there's no reason for both of them.

We actually tested this view in our implementation by making the VLIs equal to the HIDs. In other words, the SCMP VLI at an agent was always the same as the HID of the stream. This approach caused only two minor problems.

The first problem appeared while running ST-2 over a software loopback, so that the hop both originated and terminated on the same machine. In this situation, there existed both an outbound stream and inbound stream with the same HID and thus the same SCMP VLI. Since most SCMP messages are directional, it was usually clear which type of stream the message was for and there was no problem looking up the right one using the HID. However, ACK messages (and some other message types) can travel in either direction, and so there was ambiguity about which stream to look up. Note that the same problem can occur if the various HID spaces[13] are distinct, so the ambiguity is probably indicative of a problem with ACK messages (they do not indicate direction) rather than the need for having both HIDs and VLIs.

The second problem occurred in HID negotiation at connect time. When the initial HID is rejected (as described in section 2.2.1. above), the rejecting agent is expected to return an SCMP VLI. The connecting agent then continues the HID negotiation by sending the HID-CHANGE message to the VLI provided by the rejecting agent. Because the implementation only choses a VLI after a valid HID was found, this negotiation does not work. But, as discussed in section 6.2.2. above, we believe the CONNECT followed by HID-CHANGE negotiation is a bad idea anyway.[14]

More generally, from the implementors' point of view, there is a strong interest in minimizing the number of different ways a stream can be identified. Each different method of

---

[12] Note that this mechanism requires that agents ensure that each of their downstream neighbors periodically receives an ST packet, even if all streams through those neighbors are idle. But this is a very different requirement than pinging regularly with a HELLO message.

[13] Our implementation originally used a single HID space but one could have many distinct HID spaces in one agent. For example, different spaces for upstream and downstream nodes, or HIDs on a per-interface basis.

[14] Note that if an agent decides to change an HID once the connection is established, the change works correctly with just HIDs as addresses. To ensure that data continues to flow, the change mechanism must take the form of first adding a new HID to the stream, then deleting the old HID. So when the new HID is added, there are briefly two HIDs pointed to the same state information. Then the old HID is deleted. There is no need for VLIs to make this negotiation work.

addressing requires a different look up scheme, all for looking up the *same* state information. Furthermore, a conservative implementation will properly confirm that all addresses provided in a message are consistent. For example, if an inbound SCMP message contains both a VLI and a name, the receiver should check that the pair of addresses are consistent with what the receiver has. More addresses means a potential requirement to do more consistency checking.

We believe SCMP would be simpler if one of the naming schemes were abolished (probably the VLIs).

6.3. Evaluating the ST-2 Specification

The ST-2 specification has a relatively large number of mechanical errors when compared with other experimental specifications that the authors have implemented. This section discusses some of the mechanical failings that caused problems for the implementors. The point of this discussion is to note that that the specification probably needs editing before it is more widely used.

6.3.1. Inconsistencies in the Specification

One problem in the specification is a modest number of inconsistencies. For example, page 118 states:

> Either agent may terminate the negotiation [to find an HID for a connection] by issuing either a DISCONNECT or a REROUTE. The agent that issued the HID-REJECT may issue a REFUSE, or REROUTE at any time after the HID-REJECT.

Unfortunately, no REROUTE message is defined in the specification. Similarly diagrams of HID negotiation on pages 62 and 63 mention HID-CHANGE-DELETE and HID-CHANGE-ADD messages. No such message exists – the intended message types are the HID-CHANGE message with the D or A bits turned on. Finally, the specification often refers to the HID-APPROVE message as the HID-ACCEPT message.

Another inconsistency is that most of the SCMP message format specifications (pages 100-126) show that the link reference field is to be filled in but, unfortunately, the link reference field only makes sense in cases where a control message is being sent in reaction to another control message. But there are several messages that do not reply to other control messages, including the STATUS, HELLO, DISCONNECT, and CONNECT, for which a link reference value is specified. Furthermore, acknowledgement messages are supposed to use the reference field to acknowledge a message, but for some of these messages (HID-REJECT, HID-CHANGE, HID-APPROVE, ACK, ERROR-IN-RESPONSE, ERROR-IN-REQUEST, and CHANGE-REQUEST) the specification indicates the link reference field should also be filled in and does not explain what value to use.

6.3.2. Lack of Formal Definition

A more serious problem with the specification is a lack of any formal presentation of the protocol, in the form of state tables, pseudo-code, or expression in a specification language. A formal description is important for several reasons. If expressed in pseudo-code or a specification language, the formal description often helps the implementor verify that an implementation is conformant. By fully describing the state space, the formal description helps the implementor determine the proper data structures around which to structure his or her code. Finally, a formal description is a useful tool in helping designers of a protocol confirm that it is

complete. We were particularly hampered by the difficulty of determining the state space and designing the right data structures. We believe that the ST-2 state is best thought of as the combination of the states of each hop with respect to each target but wish we had received better guidance.

7. Conclusions

From the implementation experience, we can offer three major conclusions:

- The BSD UNIX networking code proved very flexible, able to support a protocol providing a novel communications paradigm with only minor changes.
- ST-2 proved to be a difficult protocol to implement. Some of this difficulty was due to problems with the protocol, but much of the difficulty was due to problems with the specification. The absence of formal descriptions, the various confusing mechanical errors and some redundant functionality made the implementation process more painful than necessary. The specification is in need of revision.
- It appears that a substantially simpler version of ST-2 could still provide the same service. Some ideas about how the protocol could simplified are presented in Section 5. We believe that such a simplified protocol would be a more interesting proponent of the ST-2 philosophy.

On a lesser note, the problems encountered with strict separation of control and data messages lead us to believe that such a separation is not consistent with robust internetworking. We also feel that more thought needs to be given to defining a minimum set of flow parameters.

Combining these conclusions into a general review, we offer the following high-level review. The idea that multipoint isochronous applications require connection-oriented multipoint network protocols is an interesting one that could benefit from further study. We are less sanguine about the current version of ST-2 as a vehicle for doing that further study. The basic concepts underlying the design of ST-2 have largely been validated by version 1 of ST. As a result, the experimental interest is not in simple proof of concept testing, but rather in widespread experimentation with the protocol. In its current form, ST-2 is not ready for widespread distribution.

Acknowledgements

Författarna vill tacka dem som har gett synpunkter på utkast till denna uppsats; Alex McKenzie, Bob Braden, Steve Blumenthal, Steve Casner och särskilt Charlie Lynn för speciellt insiktsfulla kommentarer. Vi vill också tacka Joachim Nilsson och Ricardo Civalero för arbetet att flytta PVP och NVP till vår implementering. Författarna är ansvariga för innehållet endast i denna uppsats.[15]

[15] English translation: The authors would like to thank a number of people who reviewed drafts of this paper: Alex McKenzie, Bob Braden, Steve Blumenthal, Steve Casner, and especially, Charlie Lynn, whose comments were extraordinary in their insightful detail. We would also like to credit the work of Joachim Nilsson and Ricardo Civalero on porting PVP and NVP to run over our implementation. The authors are solely responsible for the content of this paper.

References

[#1] J. Postel. *Internet Control Message Protocol*, RFC-792. Internet Request for Comments, no. 792. September 1981.

[#2] C. Topolcic, Ed. *Experimental Internet Stream Protocol, Version 2 (ST-II)*, RFC-1190. Internet Request for Comments, no. 1190. October 1990.

[#3] R.M. Sanders, and A.C. Weaver. "The Xpress Transfer Protocol (XTP) – A Tutorial," in ACM SIGCOMM *Computer Communication Review*, October 1990, Vol. 20, No. 5, pp. 67-80.

[#4] G.M. Parulkar, "The Next Generation of Internetworking," in ACM SIGCOMM *Computer Communication Review*, January 1990, Vol. 20, No. 1, pp. 44-53.

[#5] D. Mills. *Network Time Protocol (Version 2) Specification and Implementation*, RFC-1119. Internet Request for Comments, no. 1119. September 1989 (Revised February 1990).

[#6] B. Braden, D. Borman, and C. Partridge, *Computing the Internet Checksum*, RFC-1071. Internet Request for Comments, no. 1071. September 1988.

[#7] T. Mallory, and A. Kullberg, *Incremental Updating of the Internet Checksum*, RFC-1141. Internet Request for Comments, no. 1141. January 1990.

[#8] J. Forgie, *ST - A Proposed Internet Stream Protocol*, IEN-119. Internet Engineering Note, No. 119. September 1979.

[#9] V. Jacobson, "Congestion Avoidance and Control," in *Proc. ACM SIGCOMM '88*, pp. 314-329. Stanford, California. August 1988.

[#10] P. Karn and C. Partridge, "Improving Round-Trip Time Estimates in Reliable Transport Protocols," in *Proc. ACM SIGCOMM '87*, pp. 2-7. Stowe, Vermont. August 1987.

[#11] S.J. Leffler, M.K. McKusik, M.J. Karels and J.S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[#12] W. Edmond, K. Seo, M. Leib and C. Topolcic. "The DARPA Wideband Network Protocol," in *Proc. ACM SIGCOMM '90*, pp. 79-89. Philadelphia, Pennsylvania. September 1990.

[#13] J. Postel, Ed. *Internet Protocol*, RFC-791 Internet Request for Comments, no. 791. September 1981.

[#14] R. Braden, Ed. *Requirements for Internet Hosts — Communication Layers*, RFC-1122. Internet Request for Comments, no. 1122. October 1989.

[#15] Sun Microsystems Inc., *SunLink X.25 Packet-level and HDLC Programmer's Guide.* April 1989.

[#16] S. Casner, K. Seo, W. Edmond and C. Topolcic. "N-Way Conferencing with Packet Video," in *Proc. 3rd Intl. Workshop on Packet Video.* March 1990. Morristown, New Jersey.

[#17] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley, 1991.

[#18] Dinesh Verma and Hui Zhang and Domenico Ferrari, "Guaranteeing Delay Jitter Bounds in Packet Switching Networks." *Proceedings of TriComm'91*, Chapel Hill, North Carolina. April, 1991.

[#19] Domenico Ferrari and Dinesh Verma, "A Scheme for Real-Time Channel Establishment in Wide-Area Networks." *IEEE Journal on Selected Areas in Communications*. April, 1990.

[#20] D. Cohen. *A Network Voice Protocol NVP-II*. USC/Information Sciences Institute, April 1981.

[#21] E. Cole. *PVP - A Packet Video Protocol*. W-Note 28, USC/Information Sciences Institute, August 1981.

[#22] K. McKusick and M. Karels. Course Notes for *4.3BSD UNIX Internals: Data Structures and Algorithms*. Oxford-Berkeley Summer Engineering Programme, 1991 (8-12 July 1991).

[#23] D. Ferrari, "Client Requirements for Real-Time Communication," *IEEE Communications Magazine*, VOl. 28, no. 11, November 1990, pp. 65-72.

[#24] R. Droms, B. Haber, F. Gong and C. Maeda, "Report from the Joint SIGGRAPH/SIGCOMM Workshop on Graphics and Networking," *Computer Communication Review*, Vol. 21, No. 2, April 1991, pp. 17-25