Structured Stream Transport
Preliminary Protocol Specification

November 23, 2007

Contents

1 Introduction **1**
- 1.1 Structured Streams . . . 2
- 1.2 Other SST Features . . . 3
- 1.3 Why Another Transport? . . . 3
- 1.4 Document Structure . . . 3

2 Design Overview **3**
- 2.1 Interface Abstractions . . . 4
  - 2.1.1 Sessions . . . 4
  - 2.1.2 Channels . . . 4
  - 2.1.3 Streams . . . 4

3 Channel Protocol **5**
- 3.1 Protocol Organization . . . 5
- 3.2 Packet Layout . . . 5
- 3.3 Channel Identification . . . 6
- 3.4 Sequencing Layer . . . 6
  - 3.4.1 Sequencing Barriers . . . 6
  - 3.4.2 Extrapolating Sequence Numbers 6
- 3.5 Encoding Layer . . . 7
- 3.6 Integrity Layer . . . 7
- 3.7 Privacy Layer . . . 8
- 3.8 Feedback Layer . . . 8
  - 3.8.1 Transmitting Acknowledgment Information . . . 8
  - 3.8.2 Delayed Acknowledgment . . . 8
  - 3.8.3 Congestion Control . . . 9
  - 3.8.4 Fast Retransmit . . . 9
- 3.9 Path MTU Detection . . . 9

4 Stream Protocol **10**
- 4.1 Stream Layer Header Information . . . 10
- 4.2 Unique Stream Identifiers (USIDs) . . . 10
- 4.3 Attaching Streams to Channels via LSIDs 11
- 4.4 Special Internal Streams . . . 11
  - 4.4.1 The Channel Root . . . 11
  - 4.4.2 Service Request Streams . . . 11
- 4.5 Initiating Streams . . . 12
- 4.6 Initial Stream Data . . . 13
- 4.7 Response to Stream Initiation . . . 13
- 4.8 Post-Initiation Data Transmission . . . 13
- 4.9 Acknowledging Data Segments . . . 14
- 4.10 Retransmitting Lost Segments . . . 14
  - 4.10.1 Flow Control . . . 15
- 4.11 Pushing Data to the Application . . . 16
- 4.12 Marking Messages or Records . . . 16
- 4.13 Closing Streams . . . 16
- 4.14 Forceful Reset . . . 17
- 4.15 Rules for Reusing LSIDs . . . 17
- 4.16 Best-Effort Datagrams . . . 18
- 4.17 Attaching Streams . . . 19
- 4.18 Detaching Streams . . . 19

5 The Negotiation Protocol **19**
- 5.1 Basic Design Properties . . . 19
- 5.2 Message Structure . . . 19
- 5.3 Chunk Structure . . . 20
- 5.4 Initiation and Method Selection . . . 20
- 5.5 Lightweight Checksum Negotiation . . . 20
  - 5.5.1 Initiating Checksum-Authenticated Channels . . . 21
  - 5.5.2 Responding to Checksum Negotiation Chunks . . . 21
  - 5.5.3 Retransmitting Negotiation Packets 22
  - 5.5.4 Computing Checksums . . . 22
- 5.6 Reusing Channel IDs . . . 23

6 Registration Protocol **23**

1 Introduction

Structured Stream Transport (SST) is a new transport protocol designed to give today's sophisticated Internet applications the communication tools they need to operate effectively on today's fragmented and hostile Internet. In brief, SST offers:

- Multiple independent streams that can run in parallel over a single SST session, representing different requests or logical activities for example.
- Relative prioritizaion between streams to enforce application-specific policies.
- Semantically unified support for both reliable and best-effort delivery—not just two modes shoehorned into one protocol.
- Best-effort delivery imposes no formal or practical datagram size limit.
- Efficient support for short (e.g., transactional) use of reliable streams: no handshake delay on setup, no state retention after close.
- Streams may be arbitrarily long-running and can preserve internal application-specific record marks.
- Built-in communication security (both authentication and encryption) based on accepted, widely-scrutinized algorithms.
- Ease of deployment: normally runs over UDP [21]; can be implemented in libraries linked into applications with no special privileges.
- Hole punching support [14] for transparent communication across most NATs and firewalls.
- Wire efficiency: SST's header overhead including UDP encapsulation is only 4 bytes larger than TCP's.

1.1 Structured Streams

The most unique aspect of SST is its ability to manage and multiplex a dynamic hierarchy of streams onto a single end-to-end session. As with TCP or UDP, an SST *session* is an association between two endpoints, each endpoint identified by a combination of IP address and port number. An SST application typically needs to create only one session for each remote host it wishes to communicate with. Within this session, however, the application can create any number of *streams* at any time, each stream semantically comparable to a separate TCP connection.

SST organizes streams into a hierarchy: each session automatically has a *root stream*, and the application creates other streams as *substreams* of existing streams. Through this hierarchy, SST enables applications to control and prioritize concurrent network activities relative to each other in application-specific ways. A browser-like application might for example use one stream to download the main content for a given page, and use substreams of that stream to download embedded items within that page such as images or multimedia streams. If the browser supports multiple tabs, it can prioritize the stream for the currently visible page above the streams for other pages, so that the page the user is currently viewing loads quickly.

SST delivers data in a single stream reliably and in-order with respect to other data in that stream, but maintains no ordering relationship with other streams, so one stream's data may "pass" that of another. The receiver may accept data from different streams at different rates without losing data: for example, the application might write one stream to fast a local disk as quickly as the sender makes it available while delivering another stream to an audio/video codec for playback at a constant rate.

Creating and destroying individual streams within an SST session is fast and inexpensive. Stream creation requires no round-trip delay like TCP's 3-way handshake does, and SST can piggyback the signaling required for stream creation onto the initial data to be sent on the stream without additional packets or header overhead. Similarly, each endpoint can immediately discard all state related to an SST stream once the stream is closed, in contrast with TCP's mandatory TIME-WAIT state.

An application can close a stream either gracefully or forcefully. When an endpoint closes a stream gracefully, it indicates that it has no more data to send, but SST waits before destroying the stream to ensure that all sent data reliably reaches the other endpoint and still allows the application to receive further data sent by the other endpoint. The stream disappears entirely only after both endpoints have closed the stream and all transmitted data is acknowledged. When an application forcefully closes or *resets* a stream, SST immediately discards the stream's state without waiting for acknowledgment of data already sent.

An SST application requiring low-overhead best-effort delivery can send "datagrams" via *ephemeral substreams*. An ephemeral substream is semantically equivalent to an ordinary substream: the receiving application cannot tell the difference between an ephemeral substream and a normal one. Since different streams have no ordering relationship, SST does not delay delivery of the "datagram" contained in an ephemeral substream as a result of a lost or delayed packet in another ephemeral substream. Under suitable conditions, SST can optimize delivery of an ephemeral substream with a stateless UDP-like delivery mechanism for maximum efficiency. Unlike conventional datagram-oriented transports, however, SST imposes no limit on the size of a best-effort datagram: ephemeral substreams too large to deliver in datagram-oriented fashion without unacceptable probability of loss simply fall back to SST's normal reliable stream delivery mechanism.

1.2 Other SST Features

SST is designed for deployment either at system level as a "native transport" alongside TCP and UDP, or at application level running atop UDP. The latter usage allows applications to ship with a library implementation of SST without requiring any special privileges or extensions to existing operating systems, and they can use it as they would SSL/TLS [11] or DTLS [22]. Deploying SST atop UDP also allows it to traverse existing NATs that only natively support TCP and UDP.

On today's hostile Internet, communication security has become essential for almost every application. Since IP-level security (IPsec) [17] is still not widely deployed other than for corporate VPNs, however, applications must usually supply their own security, either by incorporating it into their protocols or by inserting a security layer such as SSL/TLS [11] above an insecure legacy transport. SST in contrast includes communication security as a basic feature, integrated into its design to minimize cost and complexity and to reuse common protocol elements such as sequencing and feature negotiation. SST's "baseline" ciphersuite built on AES128-CTR and HMAC-SHA256, for example, requires no additional headers or other per-packet overhead other than a 128-bit MAC.

Since a large percentage of hosts on the Internet today are connected behind NATs or firewalls, safe and effective traversal of these barriers has become a major challenge and requirement for many applications, especially applications with peer-to-peer communication patterns such as Voice-over-IP. Legacy transports generally do not address NAT traversal because they were designed for an idealized "flat" Internet of the past that no longer exists except in the minds of theorists and wishful thinkers. SST incorporates as a basic feature support for hole-punching [14] across BEHAVE-compliant NATs [3], which include the majority of NATs already deployed and almost all NATs that will be on the market in the near future.

1.3 Why Another Transport?

A number of alternatives to TCP [25] and UDP [21] exist today, but none of them offer the combination of functionality needed by today's highly asynchronous, media-rich applications, which often need to juggle many different types of network communication activities at once and must traverse NATs and firewalls. To summarize the limitations of some existing alternatives:

- SSL/TLS [11] and DTLS [22] implement connection security atop TCP and UDP, respectively, but inherit all of the other limitations of the base protocols.
- DCCP [18] implements congestion control for UDP-style best-effort communication, in the process incurring much of the same protocol complexity as TCP without providing reliable delivery, security, or other high-level features when they are desired.
- SCTP [24] can multiplex multiple logical streams over a single session, supports both reliable and best-effort delivery modes, and provides fail-over across a group of redundant endpoints. SCTP has limitations reflecting its telecommunications focus, however:
  - SCTP streams cannot be created mid-session, only negotiated "en masse" at session initialization, limiting their utility for ephemeral or transaction-oriented activities.
  - SCTP implements only one receive window per session rather than one per stream, so the receiver cannot accept data on one stream while applying back-pressure to others, further limiting their independence and usefulness to all but fixed-rate (e.g., telecom) applications.

Also, since DCCP and SCTP operate atop IP as new protocols alongside TCP and UDP, DCCP or SCTP sessions can only traverse those (currently very rare) NATs that explicitly support these new transports, and applications cannot implement these transports on existing end hosts without kernel support or special privileges. SST in contrast normally operates atop UDP, so it is immediately compatible with the vast majority of deployed NATs and firewalls that only understand TCP and UDP, and it can be implemented at user level by unprivileged applications or libraries linked into them. (SST could of course be adapted into a "native" transport operating directly atop IP, if that proves desirable.)

1.4 Document Structure

This document is organized as follows. Section 2 first presents a high-level overview of the design of the SST transports and the various sub-protocols it consists of. Following sections then describe these specific sub-protocols in detail: Section 3 describes the Channel Protocol, Section 4 describes the Stream Protocol, Section 5 describes the Negotiation Protocol, and finally Section 6 describes the Registration Protocol.

2 Design Overview

SST is organized into four separate but closely-related protocols, organized as shown in Figure 1:

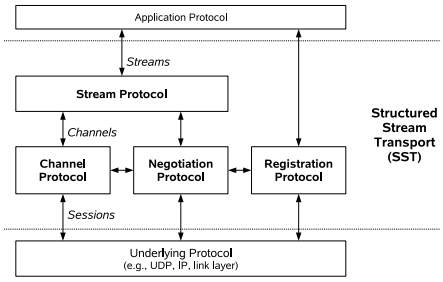

Figure 1: SST Protocol Architecture

- The *Channel Protocol* implements a basic packet-oriented *channel* abstraction that provides sequencing, connection security, and congestion control.
- The *Negotiation Protocol* provides the mechanics of setting up channels between hosts, including symmetric key agreement for channel security and negotiating optional protocol extensions.
- The *Registration Protocol* provides a simple, optional host registration and lookup service supporting secure host identities and NAT traversal.
- The *Stream Protocol* builds on the three protocols above to implement the convenient, high-level stream abstraction that SST presents to applications.

2.1 Interface Abstractions

Providing the glue within SST and between SST and its surrounding environment are three crucial abstractions: *sessions*, *channels*, and *streams*.

2.1.1 Sessions

A *session* represents a context in which SST runs over some underlying network protocol such as UDP or IP. Each session represents an association between two network *endpoints*. A session is always uniquely defined by a pair of endpoints, but the definition of an endpoint depends on the underlying protocol on which SST runs:

- When SST is run atop UDP, an endpoint consists of an IP address paired with a 16-bit UDP port number. From the perspective of any given host a session is thus uniquely defined by the 4-tuple (local IP, local port, remote IP, remote port). The session tuple for the opposite host is obtained by swapping the local and remote parts of the tuple.
- If SST is run directly atop IP as a "native" transport alongside TCP and UDP, then an endpoint consists only of an IP address, and thus an SST session is uniquely defined by the pair of IP addresses of the hosts involved: (local IP, remote IP). By definition there can be only one such "native" SST session at a time between any pair of hosts.
- If SST is run atop some other network- or link-layer protocol, then SST uses as its "endpoints" whatever the underlying protocols uses as an "address" or "host identifier." If SST were to be run directly atop Ethernet, for example, then SST's endpoints would be IEEE MAC addresses, and a session would be uniquely defined by a pair of MAC addresses.

2.1.2 Channels

The *channel* abstraction provides the interface between the channel protocol and the stream protocol. The channel protocol can multiplex up to 255 distinct *channels* onto a session. One 8-bit *channel number* for each direction of communication distinguishes different channels for one session: thus, a channel is uniquely identified by the 4-tuple of (local endpoint, local channel, remote endpoint, remote channel). Each channel represents a separate instance of the SST channel protocol resulting from a successful key exchange and feature negotiation using the negotiation protocol; SST's channels are therefore analogous in function to security associations in IPsec [17]. Different channels always use independent symmetric keys for encryption and authentication and may use entirely different encryption and authentication schemes or other optional negotiated protocol features. A given channel always uses one set of symmetric keys and negotiated parameters, however: when SST needs to re-key a communication session (e.g., to ensure freshness of symmetric keys), it does so by creating a new channel through a fresh run of the negotiation protocol and terminating use of the old channel. SST may keep multiple channels active at once to allow applications to select different security parameters for different streams, fully encrypting and authenticating sensitive streams for security while leaving less-sensitive streams in cleartext and only weakly checksummed for maximum efficiency, for example.

2.1.3 Streams

Finally, a *stream* is the high-level logical communication abstraction that SST presents to applications. Each stream supports two-way TCP-like communication, reliably preserving data content and ordering within the stream, while allowing communication on each stream to proceed inde-

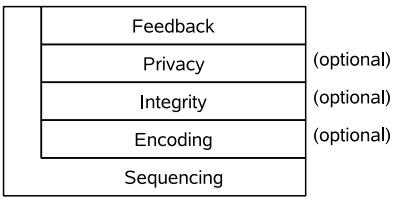

Figure 2: Channel Protocol Sublayers

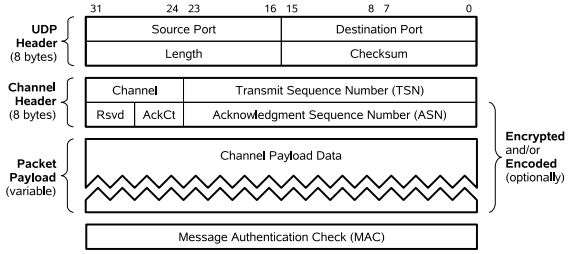

Figure 3: Channel Protocol Packet Layout

pendently of other streams. SST multiplexes all of the application's streams onto one or at most a few channels, one channel for each different set of security parameters required; hence the limit of 255 active channels at once is not likely to impact applications. SST places no arbitrary limits on the number of streams an application may have or on the duration a given stream may be used. SST enhances TCP's stream abstraction with zero-delay stream creation and optional message/record marking, and enables the application to organize streams hierarchically according to the logical structure of its communication activities.

3 Channel Protocol

SST's channel protocol provides a sequenced, secure, congestion-controlled, packet-oriented communication abstraction for the stream protocol to build on. This section describes the channel protocol in detail, starting with the protocol's organization, header and packet structure, and encapsulation of SST packets for transmission across the Internet, followed by a description of each of the channel protocol's components.

All of the details in this section describe SST's baseline "version 1" channel protocol. Almost any aspect of this protocol may be changed in the future through the use of the negotiation protocol: two communicating hosts may adopt an extended or even a completely different channel protocol if they both support it.

3.1 Protocol Organization

The channel protocol is organized into five functional sublayers, shown in Figure 2 and summarized below from bottom to top:

- The *sequencing* layer assigns sequence numbers to each packet transmitted on a channel, and protects against replay attacks on packet reception.
- The optional *encoding* layer can implement forward error correction (FEC) or other special encodings that may be desired to enhance the performance of the underlying network path.
- The optional *integrity* layer attaches a message authentication check (MAC) to each packet transmitted, and verifies the MAC on packet reception.
- The optional *privacy* layer encrypts the payload of each packet transmitted and decrypts it on reception.
- The *feedback* layer sends acknowledgments for packets received on a session, and uses these acknowledgments to monitor the network's performance and implement congestion control.

The sequencing component extends up the left side in the figure because the sequence numbers it attaches to packets are used by all the higher-level layers and made available to higher-level protocols (e.g., the stream protocol) as part of the channel abstraction. Thus, unlike the optional intermediate layers, the sequencing layer adds "semantic value" rather than simply representing a stackable transparent payload transformation.

3.2 Packet Layout

Figure 3 shows the layout of a packet in SST's basic channel protocol. Since SST packets are normally embedded in UDP datagrams for compatibility with deployed NATs that only support TCP and UDP, the diagram includes the UDP header to clarify the relationship between the UDP and SST headers. UDP encapsulation is not essential to SST's design, however: SST could just as easily run directly atop IP or even a link-layer protocol, if desired.

The SST part of the packet consists of an 8-byte channel header, followed by a variable-length payload containing upper-level (e.g., stream protocol) headers and application data, and finally a Message Authentication Check (MAC) field that SST's integrity sublayer uses to verify the integrity of each packet. If privacy protection is enabled, as it usually is by default under SST, then all but

the first 32 bits of the SST header and all of the variable-length payload is encrypted. SST determines the cryptographic algorithms to use for integrity and privacy protection, if any, at session negotiation time as described later in Section 5.

3.3 Channel Identification

Each channel protocol packet begins with a 32-bit word, always transmitted in cleartext, containing an 8-bit channel number and a 24-bit transmit sequence number.

SST uses the channel number field to distinguish among channels multiplexed onto one session. During a run of the negotiation protocol, each participating host chooses a channel number and communicates it to the other host; each host then places the *other* host's chosen channel number in all subsequent channel protocol packets on that channel. The channel number field in the header could therefore be thought of as a "destination channel number," with the corresponding "source channel number" being implicit and omitted from the packet header. The interpretation of all other data in the packet depends on the channel number, so a channel might use a different channel protocol from the one described here as dictated by the results of negotiation for that channel.

The channel protocol never uses channel number zero: instead, a channel field of zero in a packet indicates a *control packet* for one of SST's control protocols, such as the negotiation protocol 5 or the registration protocol 6.

3.4 Sequencing Layer

Within each channel using this baseline channel protocol, each host assigns a 64-bit sequence number to each packet it transmits, and includes the low 24 bits of that sequence number in the *transmit sequence number* field of the packet header. The 64-bit sequence number counters for a channel always start at 1 when the channel is first negotiated and increase by 1 for each packet transmitted. Every packet transmitted over a channel consumes sequence numbers space, including packets that contain only an acknowledgment or a retransmission of previously sent data. A host never sends a packet with 64-bit sequence number zero: instead, when it sets up a new channel it initializes its state as if it had already sent a fictional "packet zero" and received the corresponding fictional "packet zero" from the other host. Senders *must not* allow the full 64-bit sequence number counter to wrap: if a host has sent $2^{64} - 1$ packets and exhausted its sequence number space, it must stop sending new packets on that channel and instead set up a new channel for further communication by re-running the negotiation protocol.

On the receive side, SST's sequencing layer uses sequence numbers to protect against accidental packet duplication or malicious packet replay attacks. Each host maintains in its internal state a 64-bit *receive sequence number* representing the highest-numbered packet it has successfully received and authenticated on the channel. In addition to the receive sequence number, the host may also retain some limited amount of information about which packets with lower sequence numbers have or have not been received, in order to allow reception of packets delivered out of order while still protecting against packet replay. A host typically maintains a bit mask representing the sequence numbers within a "sliding window" up to the last received packet: if the host's current receive sequence number is N and it uses a 32-bit mask, for example, then the mask represents which packets have been received in the sequence number range $N - 31$ through N. If the host receives a packet it has already received, or an old packet outside of the window for which it retains information, it must silently discard the packet. Although limited information is presently available to guide the choice of the size of this bit mask, some recent packet reordering measurements suggest that 32 bits is likely to be sufficient for most purposes [27].

3.4.1 Sequencing Barriers

Upper layers may wish at times to prevent packets from being delivered out-of-order around certain crucial synchronization points. To this end, an upper layer may explicitly request SST's sequencing layer to set a *barrier* at the host's current receive sequence number. To set a barrier, the sequencing layer simply marks every packet up to this point as having already been received for replay protection purposes, so that any packet that subsequently arrives with a lower sequence number is unconditionally dropped. SST's stream layer uses this barrier mechanism to prevent old packets from "passing" critical points where stream identifiers may be reassigned, as described later in Section 4.15.

3.4.2 Extrapolating Sequence Numbers

To determine the full 64-bit sequence number of a packet, the receiving host extrapolates the packet's 24-bit transmit sequence number field as follows. First the host subtracts the low 24 bits of its internal receive sequence number counter from the packet's transmit sequence number value, using two's complement arithmetic, to yield a 24-bit delta. The host then sign-extends this delta to 64 bits, and adds the result to its 64-bit receive sequence number counter to yield the full 64-bit sequence number for

the packet. This computation yields the sequence number intended by the sender as long as the intended value is within a window of plus or minus about 2^{23} packets from the receiver's current receive sequence number. To avoid getting desynchronized with the receiver, therefore, the sender must not allow its packet transmission to get too far ahead of the last packet it knows (via acknowledgments) that its partner has received. In high-bandwidth, high-delay scenarios in which a window of 2^{23} packets is insufficient to make full use of the available bandwidth, the hosts may negotiate a variant of the SST channel protocol with a wider transmit sequence number field.

In practice the sender can easily avoid overrunning the sequence number window by clamping its congestion window (described below in Section 3.8) to a suitable maximum. Suppose the sender's maximum congestion window is M packets. If the sender's current receive sequence number is S, then it may potentially send sequence numbers up to $S+M$ to fill the congestion window before receiving any further acknowledgments. Suppose further that the sender doubles its retransmission timer on each timeout, the minimum round-trip time that the sender can measure due to the resolution of its timers is T_{min}, and the maximum retransmission timeout before the sender gives up and closes the channel is T_{max}. Then, if the sender continues to receive no acknowledgments for new data, it may send up to $\lceil log_2(T_{max}/T_{min}) \rceil$ additional packets in retransmission attempts before receiving an acknowledgment. If the sender uses 64-bit counters for its timers, this latter value is bounded by 64, so the sender may reach sequence numbers up to $S + M + 64$ when the receiver is known to have seen sequence numbers up to S. At this point, the only way the sender can transmit new packets without its receive sequence number increasing is if it receives acknowledgments for some very old packets below S. If the sender uses a B-bit mask to prevent replay of old packets, then the sender may receive up to B such old acknowledgments, making one new transmission in response to each, before its bit mask is full and it can only transmit new packets as a result of acknowledgments that shift the window. The sender should thus limit its maximum congestion window M to $2^{23} - 64 - B$ packets; a limit of 2^{22} is suggested as a safe conservative maximum for all reasonable values of the parameters involved.

3.5 Encoding Layer

The optional encoding layer can provide forward error correction (FEC) or other wrapper encodings to compensate for extremely lossy or corruption-prone network paths. No actual encoding schemes are defined yet, however; this layer can thus currently be considered a placeholder for future encoding extensions.

3.6 Integrity Layer

In SST's integrity layer, the sender adds a Message Authentication Checksum (MAC) to each packet so that the receiver can verify that the packet originated from the correct source and was unmodified in transit. The details of the MAC scheme depend on the security parameters and algorithms determined for the channel by the negotiation protocol, but in general message authentication operates as follows.

Once the sending host has prepared a packet payload by processing it through the other higher-level sublayers (including optional encryption) and filling in the channel and sequence number fields in the header, the host computes a kayed MAC and appends it to the packet before transmission. Upon receiving a packet and extrapolating its full 64-bit sequence number as described above, the receiving host similarly computes a MAC over the received data and compares it to the MAC field received in the packet. If the MAC check succeeds, the receiving host trims the MAC field from the packet, adjusts its information about which packets have been received including updating its receive sequence number if appropriate, and passes the rest of the payload on to upper layers; otherwise, the receiver silently discards the packet. Note that the receiver *must not* update its receive sequence number or other related internal state until it has successfully verified the packet's MAC.

The sending and receiving hosts compute a packet's MAC over an 8-byte pseudo-header containing the packet's full 64-bit transmit sequence number, followed by the entire contents of the packet to be transmitted including the SST header and possibly encrypted payload, as shown in Figure 4. Including the full 64-bit sequence number in the MAC computation ensures that if the receiver incorrectly extrapolates the 24-bit sequence number field in the packet header (e.g., due to a long-delayed or maliciously replayed packet), the MAC check fails and the receiver safely drops the packet.

Regardless of what specific MAC algorithm SST uses for a particular channel, the MAC algorithm is keyed using symmetric keys agreed for the channel via the negotiation protocol, ensuring that packets intended for one channel instance cannot be mistaken for those of another if the same pair of endpoints reuse the same channel number for successive channel instances.

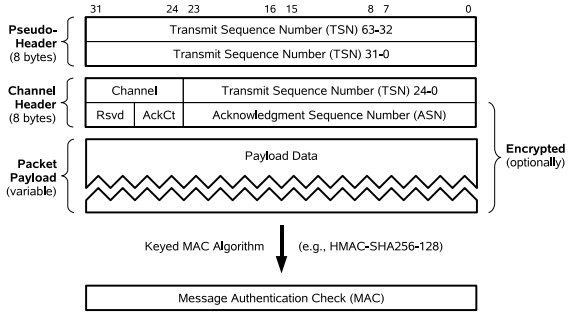

Figure 4: Packet Layout for MAC Computation

3.7 Privacy Layer

SST's optional privacy layer encrypts the contents of each packet transmitted over a channel to prevent intermediaries from snooping on communicated application data. On the sending side, the privacy layer takes a cleartext packet already prepared by higher-level layers, and encrypts the entire packet except for the first four bytes of the packet header before handing the packet down to the integrity layer. On the receiving side, the privacy layer accpets the cyphertext packet from the integrity layer, immediately after the integrity layer has verified and stripped the MAC field, and decrypts the packet header and payload.

All privacy layer parameters such as whether to encrypt at all, the specific encryption algorithm to use, and the symmetric keys for the encryption algorithm, are established at channel setup time by the negotiation protocol. In general, the only requirement is that the negotiated encryption algorithm represent a transformation that can successfully be reversed on the receiving end. Encryption algorithms must be able to handle packet payloads of arbitrary byte lengths (which may entail padding for block encryption methods), and must preserve the original payload length on decryption. The encryption transformation may expand the packet if necessary, to include an initialization vector (IV) for example, although the currently defined standard methods using AES in CTR (counter) mode preserve each packet's size exactly with no wire overhead.

3.8 Feedback Layer

The feedback layer handles the sending of acknowledgments, and using acknowledgment information to implement congestion control. The basic channel protocol defined here provides sufficient information to implement most sender-based forms of congestion control, including TCP's classic loss-based schemes and many recent delay-sensitive schemes. Other congestion control algorithms that require cooperation between sender and receiver, and perhaps extensions to the SST header, may be negotiated dynamically at channel setup time. Hosts normally treat each channel independently for congestion control, but they may if desired share congestion control state between channels with compatible congestion control schemes as certain TCP implementations do [26, 4], or even cooperate with a system-wide Congestion Manager [5] to share congestion control state across different transports.

3.8.1 Transmitting Acknowledgment Information

The feedback sublayer includes acknowledgment information in every packet sent, but leaves it to upper layers to decide when to send any packet—including when to send a packet for the sole purpose of acknowledgment. When transmitting a normal data packet whose primary purpose is something other than to acknowledge data, the sender includes in the packet's Acknowledgment Sequence Number (ASN) field the low 24 bits of its current receive sequence number, indicating the highest numbered packet it has received so far. When sending an explicit acknowledgment for a particular packet (not necessarily having the highest sequence number received so far), the sender places the transmit sequence number (TSN) of the packet to be acknowledged into the ASN field of the acknowledgment.

In either case, the sender may indicate in the 4-bit AckCt field the number of consecutive sequence numbers *immediately prior to* the specified ASN that it has also received. In this way the sender may acknowledge multiple consecutive packets at once and provide some redundancy against lost acknowledgments. Figure 5 for example illustrates the behavior of a host that receives packets with sequence numbers 1 through 7 in order, except for a missed packet having sequence number 4. When the host receives packet 2, its acknowledgment covers both packets 1 and 2; when the host receives packet 3, its acknowledgment covers packets 1 through 3. When the host receives packet 5, it sees that it missed a packet (which most likely has been dropped but may have merely been delayed), and "resets" the acknowledgment window to cover only the newly-received packet 5. If the acknowledgment for packets 1 or 2 is lost, then it is redundantly covered by the acknowledgment for packet 3, and the acknowledgments for packets 5 and 6 are similarly covered by the acknowledgment for packet 7.

3.8.2 Delayed Acknowledgment

XXX explain how delayed acks apply in SST

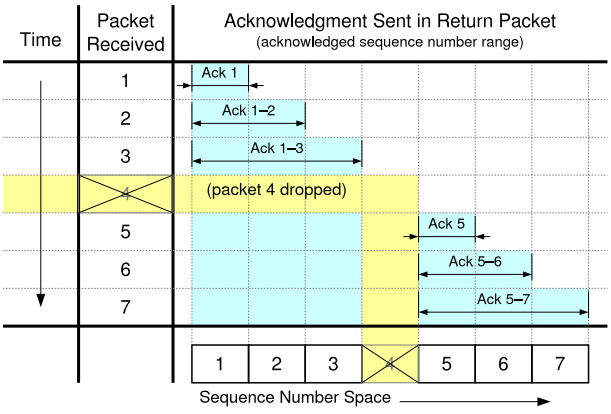

Figure 5: Acknowledgment Windows Example

3.8.3 Congestion Control

SST hosts must implement some suitable form of congestion control, such as classic TCP congestion control [2], or a more recent delay-sensitive scheme such as the one used in TCP Vegas [8]. Since the details of a particular congestion control scheme are largely independent of the details of the channel protocol itself, this section only outlines the considerations uniquely applicable to SST.

SST acknowledges individual data packets by sequence number, when or shortly after they are received, regardless of whether packets transmitted earlier also arrived successfully. Classic TCP, in contrast, uses a cumulative acknowledgment for all bytes in a logical stream up to a certain point, and cannot promptly acknowledge segments received out-of-order except using the selective acknowledgment (SACK) extension [19]. SST's feedback layer in effect provides the information benefits of selective acknowledgment without the implementation complexity or wire overhead of TCP's variable-length SACK header option. SST hosts should naturally take advantage of this information in implementing congestion control [20].

Techniques already explored in the context of TCP for adapting dynamically to Internet packet reordering conditions [7, 28] are even easier to implement in SST. Since all packets in SST including retransmissions get fresh packet sequence numbers, upon receiving an acknowledgment the sender can tell exactly *which copy* (or copies) of a retransmitted packet arrived at the receiver. The sender can use this information to detect quickly when it has incorrectly retransmitted a packet that was in fact merely delayed rather than dropped, and dynamically adjust its "packet delay threshold" for fast retransmission according to observed line conditions instead of just using TCP's conventional "three duplicate ACK" rule [2].

Since SST's sequencing and feedback layers count packets instead of bytes, it is natural to implement SST congestion control using packet counts, instead of byte counts as TCP does, for crucial metrics such as congestion window (`cwnd`) and slow start threshold (`ssthresh`). This approach is not quite correct in theory, since a large packet consumes more network resources than a small packet, and a congestion window calculated over a run of small packets may be unsuitable for a run of large packets and vice versa. In practice, however, most application streams whose bandwidth use is congestion-limited, such as file transfers, tend to be steady streams of maximum-size packets anyway, and reasonable congestion control algorithms easily adapt to occasional changes of average packet size in the transmission stream. Furthermore, the typical maximum transmission unit (MTU) on today's Internet remains largely the same as it was decades ago—at or below the 1500 byte frame size of classic Ethernet—even as bandwidth has grown by several orders of magnitude. The significance of packet size in terms of overall network resources is therefore diminishing rapidly: the Internet has practically become a cell-switched network with a cell size of 1500 bytes. For these reasons, using packet counts instead of byte counts is probably acceptable for most purposes on today's Internet, although an implementation of SST designed for high-bandwidth networks with large MTUs may wish to use byte counts for more precise congestion control.

3.8.4 Fast Retransmit

XXX reinterpretation of standard "three duplicate ACKs" rule, since SST can tell exactly which packets arrived. But really preferred to make the rule adaptive [7, 28].

XXX what "acks for new data" means

XXX no need for congestion window inflation

3.9 Path MTU Detection

XXX dynamic path MTU versus maximum segment size (MSS)...

There's two MTU issues: detecting the dynamic MTU of the underlying IP-level path, and negotiating a suitable transport-level maximum segment size (MSS).

For the first, in theory there's really nothing new in SST in relation to TCP: the transport tries sending packets at some suitable initial size, with the "Don't Fragment" flag set, and responds to ICMP MTU exceeded messages by reducing its dynamic MTU setting for the channel appropriately. The only practical challenge is actually implementing this functionality above UDP in a "portable" user-space transport protocol implementation: different

operating systems probably have different ways of allowing UDP applications to request path MTU discovery, and some operating systems may not (yet)provide this functionality at all. So for SST implementations that run in language environments or on operating systems that don't expose path MTU discovery over UDP, SST will probably just have to "guess" at a suitable MTU and risk having its UDP datagrams fragmented at the IP level.

For the second issue, I haven't yet but have been meaning to add MSS negotiation to SST as part of the negotiation of a new channel, comparable to the MSS negotiation TCP performs during its 3-way handshake. This is obviously needed so that both endpoints can agree on a reasonable maximum they both can handle with or without dynamic path MTU discovery. The negotiated MSS also affects flow control: if the sender sees the receiver's flow control window drop below the MSS, it is allowed to wait for a flow control update increasing the window above the MSS before it sends more data. This rule ensures that the sender never unnecessarily fragments packets just to fit in a temporarily small receive window, thereby avoiding silly-window syndrome. On the other hand, this rule also implies that the receiver's overall receive buffer size MUST be at least the next power-of-two in size above the initially negotiated MSS: if it were smaller, then the sender might wait forever for the receiver to increase its window above the MSS, while the receiver waits forever for the sender to send some data. Thus, the negotiated MSS is crucial to establishing a "dividing line" determining who is responsible in what circumstances for ensuring that the protocol can make progress.

Besides MSS negotiation, there are other elements of stream-level negotiation that need to be added to the protocol, such as negotiating optional extensions like fat headers and chunk bundling. Someone (you?) also suggested making the datagram optimization a negotiated optional extension, which I completely agree with.

4 Stream Protocol

The stream layer is responsible for multiplexing the application's potentially many logical streams onto one or a few channels managed by the channel protocol. Although streams are always initiated in the context of some active channel, a stream can outlive the channel in which it was created. If an SST host runs out of transmit sequence number space or the channel's symmetric authentication/encryption keys expire, for example, SST initiates a new channel to the same communication partner and migrates the active streams to the new channel transparently to the application.

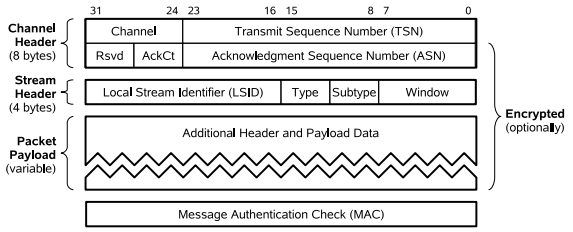

Figure 6: Stream Protocol Packet Layout

4.1 Stream Layer Header Information

As illustrated in Figure 6, each stream protocol packet consists of a 4-byte fixed stream header and a variable-length area for additional headers and payload data. The figure includes the channel protocol header as well for reference. The 32-bit fixed stream header contains a 16-bit Local Stream Identifier (LSID), a 4-bit Packet Type, a 4-bit Subtype field, and an 8-bit Window field. The size and content of the packet payload following the fixed SST header depends on the packet's type. The following type values are currently assigned:

Type	Description	Format
0	Invalid Packet Type	N/A
1	Init Packet	Figure 10
2	Reply Packet	Figure 11
3	Data Packet	Figure 12
4	Datagram Packet	Figure 16
5	Ack Packet	Figure 13
6	Reset Packet	Figure 14
7	Attach Packet	Figure **??**
8	Detach Packet	Figure **??**
9–15	Reserved	

4.2 Unique Stream Identifiers (USIDs)

SST assigns each logical stream a permanent *Unique Stream Identifier* (USID) when the stream is first created, and uses this identifier to refer to the stream if it becomes necessary to detach the stream from its original channel or migrate it to another channel. A USID consists of two components, as illustrated in Figure 7: a cryptographic *half-channel identifier* and a 64-bit *stream counter*. Each channel has two half-channel identifiers, one for each direction of information flow, both of which the negotiation protocol computes for the channel as part of its generation of symmetric key material. Which of a channel's half-channel identifiers is assigned to a given stream depends on which participant host originated the stream. The stream counter value, in turn, distinguishes among streams created by that host during the channel's lifetime.

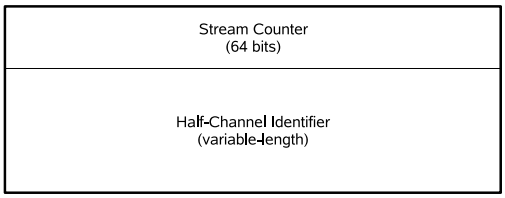

Figure 7: Unique Stream Identifier (USID) layout

Although in theory every stream has a USID, in practice for most short-lived streams that remain attached to their original channel throughout their lifetimes, the stream's USID is never actually transmitted or used by the wire protocol. Within the context of a particular channel, SST normally identifies streams using shorter 16-bit *Local Stream Identifiers* or LSIDs, described in the next section.

4.3 Attaching Streams to Channels via LSIDs

At a given point in time a stream may have between zero and four *attachments*, two for each direction of information flow. Each attachment binds the stream to a particular channel and associates a 16-bit *Local Stream Identifier* (LSID) to the stream for the purpose of transmitting stream data over that channel. The scope of an LSID is local to a particular channel and flow direction: each endpoint host on a channel has its own LSID space, within which it may assign LSIDs independently of the other endpoint and of other channels.

SST allows a stream to have up to two attachments in each direction so that a host can transmit data on a stream continuously using one attachment while setting up a second attachment to a different channel, in order to migrate streams from one channel to another smoothly and transparently to the application. Hosts may detach active streams not only to migrate them but also to free up LSID space; long-lived but inactive streams may remain unattached in one or both flow directions for arbitrary periods of time.

A future negotiable extension to the SST protocol may allow streams to have more than four attachments at once, allowing hosts to set up multiple channels over different network interfaces or paths and attach streams to several or all of them at once, using the redundant channels for fast fail-over as in SCTP [24], or for load-balanced simultaneous transmission (analogous to link layer port aggregation or "trunking").

```
struct ServicePair {
    string service;     // service name
    string protocol;    // protocol name
};

enum ServiceReqType {
    ReqPortNumber  = 0x0001,
    ReqServicePair = 0x0002,
};
union ServiceReq
switch (ServiceReqType type) {
    case ReqPortNumber:  unsigned int port;
    case ReqServicePair: ServicePair pair;
};
```

Figure 8: Service request message format

4.4 Special Internal Streams

Not all streams managed by the stream protocol are visible to applications. There are two special types of *internal streams* that the stream protocol creates and uses invisibly to the application: *channel root* streams and *service request* streams.

4.4.1 The Channel Root

Whenever a pair of SST hosts set up a new channel via the negotiation protocol, the hosts implicitly create a special stream for the channel called the *channel root*. A channel's root stream is always attached to the channel with an LSID of 0 in each direction, and never detaches or migrates to other channels. The channel itself terminates once its root stream is closed in both directions.

Applications are not generally aware of the existence of channel root streams at all: channel roots merely provide an outermost context in each channel that the stream layer uses to exchange control messages and initiate (or migrate in) other streams on behalf of applications.

4.4.2 Service Request Streams

When the application makes a `connect` request to open a new top-level stream to a given target host and service, the stream protocol on the initiating host creates a *service request stream* as a substream of a suitable channel's root stream. The initiating stream protocol then sends a *service request message* on this new stream, whose format is defined by the XDR [23] definition shown in Figure 8.

SST currently provides two types of service request messages, representing different ways of naming the service on the responding host to which the application wishes to connect. With `ReqPortNumber`, the initiator specifies a 32-bit port number, the low 2^{16} values of which

```
enum ServiceReplyType {
    ReplyOk         = 0x00,
    ReplyNoService  = 0x10,
    ReplyNoProtocol = 0x20,
    // XXX others...?
};
union ServiceReply
switch (ServiceReplyType type) {
    case ReplyOk:         void;
    case ReplyNoService:  string errormsg;
    case ReplyNoProtocol: string errormsg;
};
```

Figure 9: Service reply message format

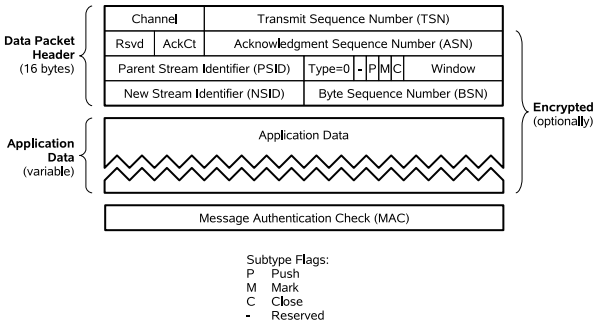

Figure 10: Init Packet Format

~~are intended to correspond to the IANA's traditional 16-bit port space used by TCP and UDP~~, while providing room for expansion to a full 32-bit port space in SST.

With `ReqServicePair`, in contrast, the initiator specifies a pair of UTF-8 strings: the first names a logical *service* such as '`mail`' or '`web`', while the second names a specific protocol to be used to communicate with the indicated service, such as '`POP3`' or '`HTTP/1.1`'.

Upon receipt of the service request, the responder looks up the specified port number or service pair in its table of available services, and responds on the service request stream with a message described in Figure 9 indicating whether the requested service is available. Error responses may contain an optional human-readable UTF-8 encoded string describing the error.

Upon receiving a `ReplyOk` message from the responder, the initiating stream protocol finally initiates a new child stream of the service request stream (a grandchild of the channel root), and hands off the use of this stream to the application to become the "top-level" stream the application requested. The initiating stream protocol may close the service request stream once the responder has acknowledged the initiation of the application stream, or it may cache the service request stream in case the application subsequently makes additional requests for new top-level streams connecting to the same service on the same target host. In the latter case, the initiating stream protocol simply initiates a new substream from the appropriate service request stream each time the application requests a new top-level stream connecting to that service.

(XXX how is the end of the service request and reply marked? Would be nice if could use FIN markers, while still holding stream "open" for creation of child streams - perhaps need multiple notions of "close...)

It is anticipated that additional service request and response message types will be added in the future permitting the initiator to, for example, query the names of available services, query the names of protocols available for interaction with a given logical service, and perhaps obtain human-readable descriptions of those services on behalf of the user.

4.5 Initiating Streams

Either host participating in an existing stream may initiate a new substream at any time. A host creates a new "top-level" application stream, or *application root*, by negotiating a channel with the desired endpoint if none exists yet and then initiating a new substream of the channel's root stream. The application can then initiate further substreams as children of streams it already has open.

To initiate a new stream, a host sends one or more Init packets, which have a packet ~~type of 0~~ and and the layout shown in Figure 10. In an Init packet, the primary LSID field in the SST header indicates the parent of the new stream to be created, and the *New Stream Identifer* or NSID field indicates the LSID that the sender has assigned to the new stream. Both of these LSIDs are interpreted in the sender's LSID space.

The host initiating a stream assigns the new stream's USID and LSID at the same time, using a 64-bit *stream counter* that the host maintains in association with each channel. In the simple case, the initiator simply increments the channel's stream counter by one, then appends the appropriate half-channel identifier to form the new stream's USID, as shown in Figure 7. The initiator then uses the bottom 16 bits of the resulting stream counter as the new stream's LSID.

Since the LSID is only 16 bits, however, the desired LSID may conflict with an existing attachment to the same channel for the same flow direction. In this case, the initiator skips the conflicting value and continues incrementing the counter until it finds an available LSID. If the desired stream counter is 0x12345 but LSID 0x2345 is already in use, for example, the initiator must skip that counter value and try LSIDs 0x2346, 0x2347, and so on. The new

stream's USID is derived from the final stream counter value corresponding to the first free LSID.

When a host receives the first Init packet for a new stream, it extrapolates the 16-bit LSID in the header's NSID field to determine the new stream's full USID. For this purpose, the receiver maintains a 64-bit *receive stream counter* associated with the channel, in which it records the highest stream counter value it has seen its peer assign so far. To extrapolate a new stream's 64-bit stream counter, the receiver forms a 16-bit two's complement delta from its current receive stream counter and the received NSID, and then adds the sign-extended delta to its receive stream counter, in the same way that the sequencing layer extrapolates the TSN and ASN fields.

As with packet sequence numbers, the host initiating a new stream must avoid getting too far ahead of its peer in stream counter space and causing the receiver to extrapolate a received NSID incorrectly. For this purpose, each host must keep track of its highest stream counter value that its peer definitely knows about—i.e., the highest-numbered stream for which the initiator has received an acknowledgment for one of the stream's Init packets. The initiator must in no event assign a new LSID that exceeds the current highest acknowledged stream number by 2^{15} or more. If the initiator runs out of free LSIDs in this range, it must delay the initiation of new streams until one or more LSIDs within this range become free.

The initiator does not have to assign stream numbers in strictly ascending order: it may go back and assign stream counter values that were previously skipped due to LSID conflicts, as long as it assigns each individual 64-bit stream counter value at most once to ensure that each stream's USID is unique. The initiator can explicitly detach old streams that are still active but not in frequent use in order to free up LSID space, as described later in Section 4.18. To avoid the possibility of overflowing the receiver's 16-bit counter arithmetic, the sender must not assign stream counter values that are less than the highest stream counter it has assigned so far by 2^{15} or more.

4.6 Initial Stream Data

The Init packets a host sends to initiate a new stream may also contain application data: unlike TCP, SST does not require a round-trip handshaking delay before the application can begin sending data on a new stream. The 16-bit Byte Sequence Number (BSN) field in each Init packet indicates the logical byte offset in the new stream at which the payload data is to be placed. Unlike TCP, whose initial sequence numbers the endpoint hosts negotiate via their SYN packets, the byte sequence numbers for a new SST stream always begin at zero.

When a host receives the first Init packet for an unknown LSID, it sets up its internal state for the new stream as described above and immediately begins collecting data segments for the new stream. If the host receives the new stream's Init packets in order starting with an Init packet having a BSN of zero, then the receiving host may immediately start delivering this data to the application. If the receiving host obtains some other Init packet first, it still sets up the new stream as usual but buffers any application data the packet contains until the lower-numbered data segments arrive or the sender retransmits them.

Since an Init packet's BSN field is 16 bits wide, the initiator may send up to $2^{16} - 1 + MTU$ bytes of data immediately during stream initiation before it must start using regular data packets. The initiator cannot start sending regular data packets, however, until it is certain that its peer knows about the new stream—i.e., until it has received an acknowledgment for at least one of its Init packets for the new stream. Thus, a host can send slightly more than 2^{16} bytes on a brand-new stream with no round-trip handshaking delay. (An future extension to the SST wire protocol for for high-bandwidth, high-latency networks may increase the size of the BSN field both for Init and non-Init packets to allow for larger in-flight windows.)

4.7 Response to Stream Initiation

When a host receives the first Init packet for a new stream, the receiver must send one or more Reply packets to order to assign an LSID to the stream for use in the return direction. Reply packets have a packet type of 1 and the format shown in Figure 11. The packet's Initiator Stream ID (ISID) field indicates the LSID the initiator assigned to the new stream via the NSID field in its Init packet. The packet's Reply Stream ID (RSID) field in turn indicates the corresponding LSID the responder has assigned the new stream from its own LSID namespace, on the same channel but for the opposite flow direction.

As with Init packets, Reply packets may contain application data, which is sequenced via the packet's 16-bit Byte Sequence Number (BSN) field. The responder may send up to $2^{16} - 1 + MTU$ response bytes on the new stream before it must switch to using ordinary data packets, which it may do only after it has received an acknowledgment for at least one of its Reply packets.

4.8 Post-Initiation Data Transmission

After setting up a new stream, the participating hosts must switch to using regular Data packets to transmit data beyond the 16-bit BSN limits of Init and Reply packets.

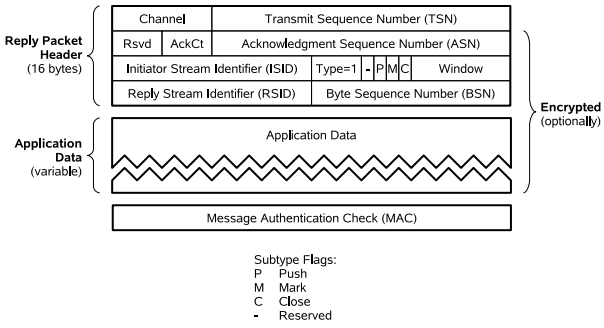

Figure 11: Reply Packet Format

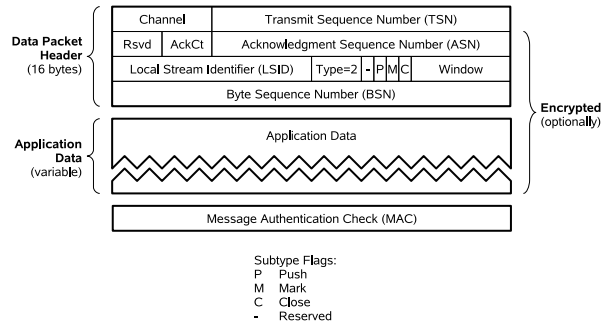

Figure 12: Data Packet Format

Regular Data packets have a type of 2 and the layout shown in Figure 12.

In contrast with Init and Reply packets, Data packets have a 32-bit Byte Sequence Number (BSN) field for the purpose of ordering data segments comprising the stream. SST uses 32-bit wraparound arithmetic to handle byte sequence numbers and reorder data at the receiver, exactly as in TCP, so there is no limit on the total amount of data the application may send on a given stream.

4.9 Acknowledging Data Segments

A host must acknowledge every Init, Reply, or Data packet it receives and successfully processes, including zero-length data segments. Hosts may acknowledge received data segments either by sending an explicit Ack packet or, preferrably, by piggybacking the acknowledgment onto the next packet it sends for some other purpose. Hosts should use standard delayed acknowledgment techniques to minimize the number of explicit Ack packets they need to send [9].

If a host receives a data packet but cannot successfully process it due to a temporary resource shortage—because it does not have enough buffer space to store it for the application, for example—then the host may drop the packet without acknowledgment. Receivers should not generally have to do this, however, if both sender and receiver implement flow control properly as described later in Section 4.10.1. Using flow control to avoid buffer exhaustion is much preferred over dropping packets, because lost packets trigger the sender's congestion control and throttle data transmission for *all* streams sharing the channel.

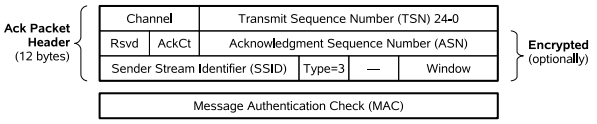

Figure 13: Acknowledgment Packet Format

A piggybacked acknowledgment for a data segment only needs to be on the same stream as the acknowledged data if the receiver wishes to adjust its receive window for the stream, as described below in Section 4.10.1. Otherwise, acknowledgments are stream-independent since they refer to the channel's packet sequence numbers and not the stream's byte sequence numbers. Through the acknowledgment window defined by the AckCt field, any packet flowing in one direction may acknowledge several packets sent on various streams in the other direction.

When a host needs to send an explicit, non-piggybacked acknowledgment, it sends an Ack packet with a Type field of 3 and the format shown in Figure 13. The host specifies in the Ack packet's Sender Stream ID (SSID) field the LSID of the appropriate stream in the *other* host's LSID space—i.e., in the LSID space of the host that sent the data being acknowledged. A host can thus send Ack packets containing flow control information even on streams that currently have no associated LSID in its own space.

A host obviously must not send an Ack packet just to acknowledge another Ack packet, since doing so would result in an endless acknowledgment loop. Ack packets may however be acknowledged incidentally as part of the acknowledgment window of a packet sent in the opposite direction for some other purpose.

4.10 Retransmitting Lost Segments

The stream layer normally stores each application data segment it has transmitted until it receives an acknowledgment for the packet containing that segment, and periodically retransmits segments that appear to have been lost. Since SST acknowledgments refer to packet sequence numbers and not byte sequence numbers, the sender needs to keep track of the packet sequence number that the se-

quencing layer assigned to each data segment the last time it was transmitted, so that it can look up and free that segment once a matching acknowledgment arrives.

The receiver uses the BSN fields of incoming segments on the stream to determine the correct order of the received segments and to deliver them reliably to the application in that order. If the receiver obtains one or more segments in a stream out of order, it must hold those segments and delay delivering them to the application until any gaps in the sequence are filled. The receiver need not take any explicit action to request retransmission of specific data segments: the sender knows which data segments to retransmit based on which segments haven't been acknowledged. The SST receiver does not send any explicit form of "cumulative acknowledgment" operating in a stream's byte number space like TCP does; SST in effect relies exclusively on selective acknowledgment [19].

To calculate a conservative estimate of the receiver's *cumulative receive position*, or the first point in the stream at which the receiver is missing data, the sender simply uses the BSN of its oldest data segment that has not yet been acknowledged. The sender's idea of the cumulative receive position may be less than the receiver's actual value, due to network delay or lost acknowledgments. One or a few lost acknowledgments are not likely to make the sender see a ficticious "gap" in a contiguous run of sequence numbers that actually arrived successfully, however, because the receiver's acknowledgments during such a run will have large AckCt values and thus highly redundant overlapping acknowledgment windows.

4.10.1 Flow Control

While congestion control operates at a channel granularity, SST provides flow control in a model similar to TCP's at the granularity of individual streams. Each host (in theory at least) reserves some amount of buffer space for each stream, in which it stores data it has received but not yet delivered to the application. The receiver regularly indicates how much buffer space it has available, and the sender must not send data beyond the receiver's specified receive window.

Whenever the stream sublayer sends a packet related to a stream, it uses the 8-bit Window field in the packet header to indicate its current receive window. A value n less than 128 in the Window field indicates that the receiver has at least n bytes of buffer space available beyond the receiver's current cumulative receive position. A value of zero in particular indicates that all buffers are full and the sender must not send any new data on this stream.

A value of $n \geq 128$ on the other hand indicates that at least $2^{(n-128)} - 1$ bytes are available beyond the cumulative receive position. This exponential encoding allows the receiver to express large windows efficiently with only a few header bits: the sender does not need much precision when the window is large because it will take a while for it to fill the window anyway, but the receiver's window size indications become more precise as its buffers fill and the window shrinks. As in TCP, the window size must always be less than 2^{30} bytes to avoid overflowing the 32-bit sequence arithmetic [16], so a host must not send a value in the Window field greater than $128 + 30 = 158$.

The sender's idea of the cumulative receive position may occasionally be smaller than the receiver's due to lost acknowledgments, as described above in Section 4.10, causing the sender to underestimate the receive window horizon by the same amount. This situation is not likely to occur often or to last very long, however, because of the redundancy in the receiver's overlapping acknowledgments. If the sender does unncessarily throttle its transmission as a result of underestimating the receive window, it will at any rate obtain the correct receive window position automatically as soon as it has retransmitted and received fresh acknowledgments for any data segments whose original acknowledgments were lost.

To avoid sending an unnecessarily large number of small packets when transmission rate is limited by the receive window (a problem known as "silly window syndrome" [9]), the sender should avoid sending segments smaller than the current MTU just to make them fit into the remaining receive window. Instead the sender should merely send as many complete segments as will fit into the receive window and buffer any further segments until the window increases.

If a segment contains a special marker such as a push, message boundary, or end of stream, the sender should transmit the segment as soon as it fits into the current receive window. In addition, if such a segment does not fit into the receive window but is less than twice the size of the current receive window, then the sender should immediately send the part of the segment that does fit into the current window. Because of the exponential window encoding, the sender's notion of the receive window may be less than the receiver's actual window by just under a factor of two; the above rule allows the sender to make progress and obtain a further-refined window update on the next round-trip when it cannot be sure whether the pushed segment fits into the actual window.

For flow control purposes, hosts treat the data in any Init packet as belonging to the *parent* stream—the stream specified in the packet's main LSID field—even though the data itself is semantically associated with the new child stream being created. In effect, when a host im-

mediately sends data on a new stream without waiting for the responder to acknowledge the Init packet and provide a starting window size for the new stream, the sender effectively "borrows" from the parent stream's receive window to send this initial data. This borrowing behavior is essential to maintaining proper flow control and avoiding overrunning the receiver's buffers while allowing stream creation with no round-trip handshaking delay.

Issue: we probably need to negotiate a transport-level maximum segment size on channel setup as in TCP, to avoid deadlock in case the MTU one host observes is close to or larger than the other host's maximum receive window size.

Issue: we also need a flow control mechanism for new substream initiations, not just for new data transmissions on this stream.

Issue: hosts might wish to retain the "borrowing" behavior throughout the lifetimes of certain substreams, instead of always establishing separate receive windows after the first round-trip.

4.11 Pushing Data to the Application

Init, Reply, and Data packets all have a Push (P) flag in the packet's subtype field. This flag works the same way as in TCP, indicating that the receiver should "push" the data contained in this segment up to the receiving application as quickly as possible, without waiting for further segments to follow. The sender normally sets the Push flag automatically at the end of every atomic "write" operation the application performs, unless the application specifically requests otherwise.

If the sender does *not* set the Push flag or any of the other marker flags described below, then the receiver may hold the data segment arbitrarily long in order to collect additional segments and combine them for more efficient delivery to the application. Even if a segment's Push flag is set, the receiver may still combine the segment's data with subsequent segments for delivery to the application if those subsequent segments are already buffered due to out of order delivery, or if the application is not ready to accept data at the time the pushed segment first arrives.

4.12 Marking Messages or Records

Many applications logically divide their streams into messages or records, which traditionally requires introducing an explicit record marking facility above TCP's homogeneous byte streams. To simplify these applications, SST provides a simple facility by which the sendor can insert explicit record marks into the stream, which (unlike Push markers) the receiving host is guaranteed to preserve while delivering data to the receiving application.

When the sending application writes data to the stream and indicates the end of a record, the SST stream layer sets the Mark (M) flag in the packet containing the last segment of that data. Init, Reply, and Data packets all have Mark flags for this purpose. Upon receiving a packet with the Mark flag set, the receiver delivers stream data to the application only up to the end of the marked segment, and *never* combines the segment with subsequent segments for delivery, even if subsequent segments are immediately available. The receiving stream sublayer explicitly indicates to the application that the delivered data constitutes the end of a record, if the API in use supports such an indication.

4.13 Closing Streams

As with TCP, SST allows the two participants in a stream to close their respective "ends" of the stream independently. In a transaction-oriented application protocol such as HTTP for example [13], it is often convenient for the client to open a stream, issue a request, and close its end of the stream to indicate to the server that the request is complete; the client then awaits and receives the server's response on the now *half-open* stream.

When an application closes its end of an SST stream, the stream sublayer sets the Close (C) flag to mark the final data segment it sends on the stream. Init, Reply, and Data packets all have Close flags: a host may set the Close flag in an Init or Reply packet to close the stream even before the new stream's round-trip handshaking has completed. The sender must not send any further data packets on the closed stream other than retransmissions of old segments. On a full-duplex stream, the sender may still expect to receive data segments from its peer until the peer also closes the stream, and the sender must continue to send acknowledgments and receive window adjustments as appropriate for data it receives on a half-open stream.

A host may consider a stream to be "fully closed" in both directions, and may notify the application as such if appropriate and/or garbage collect its internal state for the stream, as soon as the following conditions are met:

- If the host is the initiator of the stream, it has received at least one Reply packet from the responder.
- The host has received a Close segment from its peer and every data segment leading up to it in the incoming data stream.
- The host has received acknowledgments from its

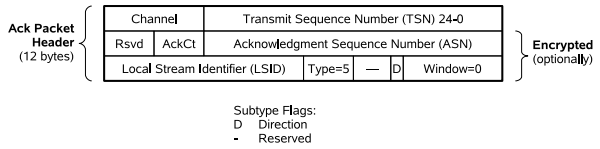

Figure 14: Reset Packet Format

peer for every data segment it has transmitted including for its own final Close segment.

Once a host has determined a stream to be fully closed, it may immediately reuse any LSIDs it had assigned to the stream: SST does not require hosts to delay reuse of old LSIDs as with TCP's TIME-WAIT state [25].

4.14 Forceful Reset

As in TCP, either participant may reset an SST stream forcefully to terminate data transmission in both directions at the same time and discard any outstanding data still in transit. To reset an active stream, a host sends a packet with a ~~Type of 5~~ and the layout shown in Figure 14. The Direction (D) flag in packet's subtype field indicates to which host's LSID space the packet's LSID field refers: if cleared, the LSID is in the sender's LSID space; if set, the LSID is in the receiver's LSID space. The Window field must in a Reset packet must always be zero.

A host should normally specify the LSID of the stream to reset in its own space and clear the Direction flag when resetting a stream it believes to be still active. After sending the first Reset for an active stream, the sender must retain the stream's state and retransmit the Reset packet as necessary until it receives an acknowledgment for one of its Reset packets. At this point the host may discard all internal state associated with the stream and reuse any LSIDs it had assigned to the stream.

If a host receives a data segment referring to an unknown LSID, the host must send a Reset packet in response to clear any stale stream state its peer may be holding. This situation can occur in particular if acknowledgments are lost during a graceful close, for example, as described earlier in Section 4.13. When responding to a data segment with an unknown LSID, the responding host need not retain any state related to the unknown stream: it merely sends one Reset packet and then drops the invalid data segment. Three specific types of data segments may trigger a Reset in this way:

- Upon receiving an Init packet whose Parent Stream ID (PSID) is unknown, the receiver responds with a Reset packet containing the unknown PSID in the LSID field and with the Direction flag set (because the unknown PSID refers to the peer's LSID space).

- Upon receiving a Reply packet whose Initiator Stream ID (ISID) is unknown, the receiver responds with a Reset packet containing the unknown ISID in the LSID field and with the Direction flag cleared (because the unknown ISID refers to the LSID space of the host that received the Reply packet).

- Upon receiving a Data packet whose LSID is unknown, the receiver responds with a Reset packet containing the unknown LSID and with the packet's Direction flag set.

When a host receives a Reset packet referring to an active stream, it acknowledges the Reset packet and then immediately transitions the stream to the "fully closed" state, at which point it may garbage collect the stream's state at any time. The host may notify the application of the forceful reset via some appropriate error indication, except in one situation: if the host has already closed its end of the stream *and* has received every data segment its peer has sent up to the peer's final Close marker, and is merely waiting for acknowledgments to one or more data segments it has sent, then upon receiving a Reset the host must consider the stream to have been "gracefully closed" as far as the application is concerned.

XXX Reset packets should contain optional descriptive "reason" message

4.15 Rules for Reusing LSIDs

To prevent confusion between packets referring to different streams that may successively use the same LSID, as illustrated in Figure 15, hosts *must* observe the following rules:

- When a host receives the first Init packet for a new stream, it must set a sequencing barrier as described in Section 3.4. This sequencing barrier prevents stale Data packets the peer might have sent on a previous stream using the same LSID from arriving after the Init packet and being misinterpreted as data for the new stream.

- A stream's initiator must set a sequencing barrier when it receives the first acknowledgment to one of its own Init packets, and must silently ignore any Reply or Reset packets apparently referring to the new stream that arrive before this point. Such a packet may refer to an earlier stream with the same LSID in the initiator's space. (The initiator should of course

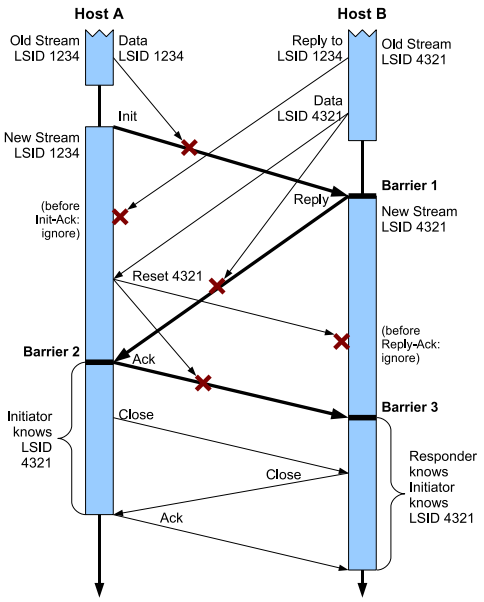

Figure 15: Sequencing Barriers at Critical Points

accept a Reply or Reset that itself carries a piggybacked acknowledgment of one of its Init packets.)

- A stream's responder must similarly set a sequencing barrier when it receives the first acknowledgment to one of its own Reply packets, and must silently ignore any packets it receives before this point that apparently refer to its newly-assigned response LSID.

- Each endpoint must set a sequencing barrier when it determines a stream to be fully closed. The responder in particular relies on this barrier to prevent a stale Init packet from arriving after the responder has garbage collected the stream's sate, causing the creation of a new "phantom" stream.

- A stream's initiator must not send any more Init packets on the stream once it receives the responder's first legitimate Reply packet for that stream. If the initiator must subsequently retransmit any data segments that it originally sent as Init packets, then it must convert them to regular Data packets before retransmission.

4.16 Best-Effort Datagrams

Many applications wish to transmit certain types of data without incurring the overhead of storing and retransmitting lost packets in the sender. In real-time streaming media applications, for example, a lost data frame is likely to be useless after incurring a round-trip retransmission delay, so it is better for the receiver just to skip or try to "fill in" for lost frames. For this purpose, SST allows applications to send *datagrams* with best-effort delivery semantics. In SST, a datagram is semantically just an ephemeral stream that the application creates, uses to transmit a sequence of bytes all at once, and then forcefully resets without ensuring that the data arrives successfully or waiting for any associated response from the receiver. When the application transmits a datagram, the stream layer need not set up or maintain internal state for the ephemeral stream it represents, and need not buffer or retransmit the data segments comprising the stream. SST still guarantees that the bytes comprising a single datagram are delivered completely, accurately, and in-order if the datagram is delivered at all.

Since an SST datagram is semantically just a restricted form of stream, the application on the receiving end cannot generally tell the difference between a received datagram and a regular stream whose content was received all at once. SST can therefore treat the sending application's request for best-effort delivery as merely a hint: the stream layer may choose to ignore this hint and instead send the datagram as an ordinary reliable stream. In particular, the stream layer falls back to reliable delivery if a datagram is so big that best-effort delivery would result in an unacceptably low probability of successful delivery. SST in this way solves transparently to the application the classic "large datagram" problem: because the loss of one fragment of a datagram entails the loss of the whole datagram, the probabiliy of datagram loss increases rapidly with datagram size, to the point where any datagram is almost certain to be lost as its fragment count reaches the inverse of the packet loss rate. Because SST can transparently switch to reliable delivery in spite of the application's best-effort hint, SST supports datagrams of arbitrary size while ensuring that they always arrive with reasonable probability. The transport layer is ideally positioned to make this decision, since it tends to collect the relevant packet loss rate information anyway to implement congestion control.

A host transmits a datagram by sending a series of Datagram Packets, each having the format shown in Figure 16. The series of packets representing a single datagram must be contiguous in packet sequence number space, and thus cannot be intermixed with packets for other streams sent on the same channel. The first packet in the series has the Begin (B) flag set, and the last packet has the End (E) bit set; a datagram that fits entirely in one packet has both flags set. Each packet contains the LSID of the parent stream within which the datagram is to

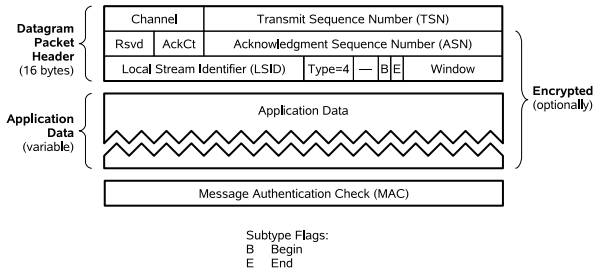

Figure 16: Datagram Packet Format

be transmitted: i.e., the parent of the implicit, ephemeral stream logically containing the datagram's content itself.

Upon receiving one or more Datagram packets, the receiver can tell easily from the packet sequence numbers and the Begin and End flags whether it properly received the entire datagram or if one or more packets were lost. As with Init packets containing data for a new stream, both participants borrow from the parent stream's receive window for flow control purposes.

~~Issue~~**:** should the datagram-oriented delivery mechanism support record marking? It certainly could (just add an M flag), but it is unclear whether it is worth the added complexity given that most datagram-oriented protocols effectively transmit exactly one "record" per datagram.

4.17 Attaching Streams

To attach a stream, a host sends an Ack packet containing the LSID to assign to the attachment and the stream's full USID as the packet's variable-length payload. Each stream may have up to two attachments at once. *(XXX fill out.)*

4.18 Detaching Streams

To detach a stream, just re-assign the LSID with a new Init or Reply packet? *(XXX fill out.)*

5 The Negotiation Protocol

SST's negotiation protocol is responsible for setting up new channels for use by the channel and stream protocols. When cryptographic integrity and/or privacy protection is desired, the negotiation protocol is responsible for performing symmetric key agreement and host identity verification.

```
typedef opaque  Chunk<>;

struct Message {
    int     magic;     // 24-bit magic value
    Chunk   chunks<>;  // Message chunks
};
```

Figure 17: Top-level negotiation protocol message format

5.1 Basic Design Properties

The negotiation protocol is asymmetric in that the two participants have clearly delineated "initiator" and "responder" roles. The protocol supports peer-to-peer as well as client/server styles of communication, however, and the channels resulting from negotiation are symmetric and can be used by either endpoint to initiate new logical streams to the other endpoint.

SST currently defines two specific methods of negotiation: a non-cryptographic method that provides fast, simple connection setup usable when the underlying network is sufficiently trusted, and a cryptographic method using Diffie-Helman key exchange, based on the Just Fast Keying (JFK) algorithm [1]. Additional negotiation methods may be added in the future, and SST is designed so that two hosts can successfully negotiate as long as they support at least one negotiation method in common.

In general, the host initiating a connection should always support secure cryptographic negotiation whenever it knows the cryptographic identity of the target host, and in this case should *only* support cryptographic negotiation unless specifically configured for an insecure mode of operation. If the initator knows the target host only by an IP address or un-authenticated domain name records, it may use either negotiation method, but the negotiation will in any case be vulnerable to man-in-the-middle attacks. The cryptographic negotiation method is capable of setting up non-cryptographic channels, thereby providing secure endpoint authentication during initial channel setup while avoiding the costs of encrypting and/or cryptographically authenticating every subsequently transmitted packet.

5.2 Message Structure

All packets used by the negotiation protocol have a common top-level layout, described by the XDR [23] definition shown in Figure 17:

In order to support NAT traversal via UDP hole punching [15], the negotiation protocol must be able to use the same local UDP port as the registration protocol uses for registering the client host, and must again use the same lo-

cal UDP port for data communication via the channel and stream protocols once the negotiation protocol has completed and set up the channel. The `magic` field serves to distinguish among packets for the different SST subprotocols that must share the same underlying session.

Although the SST negotiation and channel protocols were designed as a substrate for SST's stream protocol, they provide basic services useful to many transport protocols and thus could in theory be used as well by upper-level protocols other than the SST stream protocol. If this occurs, there is a risk that one upper-level protocol might accidentally connect to and try to negotiate a channel with a remote endpoint on which an entirely different upper-level protocol is listening, which would cause confusion at the upper layer even if both endpoints are using compatible channel and negotiation protocols. For this reason, the `magic` value the negotiation protocol uses to identify its packets is not fixed by the negotiation protocol itself, but is instead a parameter to be filled in by the upper-level protocol. A negotiation protocol instance running on behalf of the SST stream protocol uses a `magic` value of 0x00535354, or '`SST`' in ASCII; negotiation protocol instances running on behalf of different upper-level protocols must use other magic values. In any case, the upper 8 bits of the magic value (the first byte transmitted in the packet) must always be zero—an illegal channel number—to distinguish the negotiation protocol's channel setup packets from data packets transmitted over already-negotiated channels.

The `magic` field is followed by a list of *chunks* containing the content of the negotiation protocol message. Each chunk is separately XDR-encoded and packaged into an XDR `opaque` field, consisting of a 32-bit length followed by a sequence of bytes padded to a 32-bit boundary, allowing hosts to skip chunks that they cannot decode.

5.3 Chunk Structure

The XDR definition in Figure 18 describes the currently defined types of negotiation protocol chunks. Additional chunk types may be added in the future.

5.4 Initiation and Method Selection

To begin negotiation of a new channel, the initiator sends a message containing one `I1` chunk for each method of negotiation it is willing to use. In the current version of SST supporting the simple checksum (`Sum`) and Diffie-Hellman (`Dh`) methods, this means that the initiator may include a `ChunkSumI1`, a `ChunkDhI1` chunk, or both.

Upon receiving the initiation message, the responder examines the received chunks in order and processes the first one it supports, ignoring the rest. If the initiator prefers quick, insecure channel setup but is willing to perform cryptographic negotiation if the responder requires it, for example, the initiator's first message contains a `ChunkSumI1` followed by a `ChunkDhI1`. If the initiator requires secure communication, it sends only a `ChunkDhI1`.

```
enum ChunkType {
    // Lightweight checksum negotiation
    ChunkSumI1   = 0x0011,
    ChunkSumR1   = 0x0012,

    // Diffie-Hellman key agreement via JFK
    ChunkDhI1    = 0x0021,
    ChunkDhR1    = 0x0022,
    ChunkDhI2    = 0x0023,
    ChunkDhR2    = 0x0024
};
union Chunk switch (ChunkType type) {
    case ChunkSumI1:  ChunkSumI1Data sumi1;
    case ChunkSumR1:  ChunkSumR1Data sumr1;

    case ChunkDhI1:   ChunkDhI1Data dhi1;
    case ChunkDhR1:   ChunkDhR1Data dhr1;
    case ChunkDhI2:   ChunkDhI2Data dhi2;
    case ChunkDhR2:   ChunkDhR2Data dhr2;
};
```

Figure 18: Negotiation protocol chunk format

That's why a text based negotiation vector is needed, so we can pick from multiple types. We can make it non-text the same way as done here.

Note that although the initiator is technically free to send a `ChunkDhI1` followed by a `ChunkSumI1`, doing so is not recommended, because a man-in-the-middle attacker could easily force the negotiation protocol into its insecure mode even when both "real" parties in fact support secure mode. If security is desired, then the initiator must *only* include `I1` chunks for secure negotiation methods in its initial message.

5.5 Lightweight Checksum Negotiation

The lightweight checksum negotiation method consists of a simple one-phase request/response protocol, which can be extended on demand by the receiver to a cookie-based challenge/response protocol for DoS protection. The initiator includes a `ChunkSumI1` chunk in its initial message to indicate its support for lightweight checksum negotiation, and the responder replies with a message containing a `ChunkSumR1` chunk to indicate acceptance of this negotiation method. The structure of each of these chunks is shown in Figure 19.

```
struct SumI1Data {
    unsigned int  ni;       // Initiator nonce
    unsigned char chani;    // Initiator channel
    opaque        cookie<>; // Responder cookie
    opaque        ulpi<>;   // Upper-layer info
    opaque        cpkt<>;   // Channel packet
};

struct SumR1Data {
    unsigned int  ni;       // Initiator nonce
    unsigned int  nr;       // Responder nonce
    unsigned char chanr;    // Responder channel
    opaque        cookie<>; // Responder cookie
    opaque        ulpr<>;   // Upper-layer info
    opaque        cpkt<>;   // Channel packet
};
```

Figure 19: Lightweight checksum negotiation chunks

5.5.1 Initiating Checksum-Authenticated Channels

The initiator includes in its `SumI1Data` chunk a 32-bit nonce `ni` that serves to identify this run of the negotiation protocol uniquely for a given set of source and destination endpoints and a given channel ID at the initiator. Once the channel is established, this nonce also serves as a "key" for 32-bit checksum in each data packet the initiator transmits on the channel, to provide protection against packets for old channel instances being misinterpreted as packets for newer instances. Keying the data packet checksums this way also provides protection against connection hijacking by "off-path" attackers who can blindly inject forged packets into the network but cannot eavesdrop on legitimate packets between the endpoints. The receiver computes a similar nonce to protect packets on the return path, as described below.

To compute its nonce, the initiator first uses a keyed cryptographic hash algorithm such as HMAC-SHA-256 [12], keyed on a secret known only to the initiator and ideally stored persistently across restarts, to generate a secure hash of an information block containing at least the responder's endpoint identifier (e.g., the responder's IP address if SST is running directly over IP, or the responder's IP address and UDP port number if SST is running atop UDP), and the Channel ID the initiator is assigning to its end of the new channel. The initiator then extracts 32 bits of the resulting hash, then adds the current value of a timer that increments approximately every 4 microseconds, and finally clears the least-significant bit (bit 0) to zero to produce its value for the `ni` field in its `SumI1Data` chunk. All initiator nonces have their least-significant bit cleared to distinguish them from responder nonces. The initiator subsequently adds the value of this `ni` field to the 32-bit MAC field in every packet it transmits on the channel, including the MAC in any channel packet piggybacked onto the `SumI1Data` chunk as described below.

This method of "keying" SST's channel packet checksums essentially corresponds to Bellovin's method of "keying" TCP's initial sequence numbers [6], the main difference being that in SST the resulting "key" will be added to the channel protocol's checksum field instead of the sequence number as in TCP. Another subtle difference from TCP is that the initator cannot include the receiver's channel ID in its hash, because it does not know the receiver's channel ID until the receiver has responded with its `SumR1Data` chunk. This difference should not cause a security weakness unless different, mutually antagonistic entities were to control different channel IDs at the same receiver endpoint, which does not seem like a likely or even viable design point in practice.

In the first `SumI1Data` chunk the initiator sends, it assigns a Channel ID to its end of the new channel and sends this channel number in the `chani` field, but sends an empty `cookie` field (zero bytes in length). The `ulpi` field may contain arbitrary data the upper-layer protocol on the initiator wishes to pass to the upper-layer protocol on the responder as part of the connection negotiation process. The initiator may set the `pkt` field to empty, or it may include the contents of a channel packet it wishes to "piggyback" onto the `SumI1Data` chunk. This piggybacked packet includes the standard channel headers including packet sequence number and acknowledgment information, except with the Channel ID field set to zero since the initiator does not yet know the Channel ID the receiver will assign to the new channel. The piggybacked packet also includes the trailing 32-bit MAC field, computed as described below in Section 5.5.4.

5.5.2 Responding to Checksum Negotiation Chunks

When the responder receives the initiator's `SumI1Data` chunk, it first checks the chunk for syntactic validity: in particular, the least-significant bit of the initiator's nonce must be cleared to zero, otherwise the responder must reject the chunk by ignoring it. If the responder is not under heavy load, it may ignore the `cookie` field in the `SumI1Data` chunk and allocate state for the new channel immediately. Alternatively, to protect itself from DoS attacks, the responder may return a cookie challenge to verify bidirectional connectivity with the initiator before allocating state for the new channel.

To generate its cookie, the responder maintains a secret known only to itself, which is distinct from the secret used above to compute nonces, and which changes periodically to prevent cookie-jar attacks. Using a keyed secure hash

algorithm keyed with this secret, the responder hashes an information block containing at least the initiator's endpoint identifier from which the `SumI1Data` chunk arrived, and the values of the `cki` and `chani` fields of that chunk. The responder then checks the resulting hash against the `cookie` field in the received `SumI1Data` chunk, and if they don't match, returns the correct hash in the `cookie` field of a `SumR1Data` chunk with its `ni` field set to the initiator's `ni`, with its `nr` and `chanr` fields set to zero, and with its `ulpr` and `pkt` fields empty. When the initiator receives such a `SumR1Data` chunk with a `ni` matching its prior request and a `chanr` field of zero, it saves the returned `cookie` in its negotiation state and sends a new `SumI1Data` chunk containing the cookie. (XXX specify maximum cookie length?)

Once the responder receives a `SumR1Data` chunk with a valid cookie, it sets up its internal state for the new channel, allocates a Channel ID for the channel at its own endpoint, and computes its own nonce to protect subsequent packets it sends on the channel's return path. The responder computes its nonce in the same fashion as the initiator does, combining the result of a keyed hash of an information block specific to the initiator with the value of a 4-microsecond timer, but the responder sets the least-significant bit (bit 0) in its nonce instead of clearing it. The responder then replies with a `SumR1Data` chunk containin the initiator's `ni` value, the responder's nonce in the `nr` field, the responder's chosen (nonzero) channel ID in the `chanr` field, an empty `cookie` field, and any data it wishes to pass to the upper-layer protocol on the initiator side in the `ulpr` field.

The responder may also include a piggybacked channel packet to send to the initiator as part of the `SumR1Data` chunk; in this case the piggybacked packet must be formatted exactly as it would have been if the responder were to send it as a separate packet, including the initiator's selected Channel ID in the packet's channel header and a 32-bit MAC keyed on the responder's nonce. Piggybacking a channel packet does not save a round-trip in case as it can for the initiator, since the responder could instead simply send the channel packet immediately after its `SumR1Data` negotiation response, but it does permit an initiator and responder to set up a new channel and complete an entire application-level request/response transaction with only one packet in each direction (probably followed later by a delayed ACK from the initiator to the responder's initial application-level data packet).

5.5.3 Retransmitting Negotiation Packets

XXX initiator keeps same nonce for retransmits. After receiver accepts SumR1Data and creates channel state, it caches the initiator's nonce and its reply and just replays its old reply if it receives further duplicates of the initiator's request.

XXX Wesley wrote: The responder could also already be sending me packets via the channel (if only his negotiation response was lost). I can't ack these packets because I don't know his channel id. Should I just ignore them?

Yes. So this may be a reason the responder might want to piggyback its first few return-path channel packets into (otherwise-duplicate) negotiation response packets, potentially up until it gets the first valid "raw" channel packets from the initiator and thus knows that the initiator got the negotiation response - that way, no matter which subset of these channel packets get lost, the initiator still obtains the negotiation response as soon as any of them get through. The same applies to the initiator, of course, which could in theory fire off multiple I1 negotiation packets from the start with the same negotiation content but different piggybacked channel packets.

The extent to which either endpoint *should* do this of course brings up congestion control issues, in particular the fact that both sides are presumably at the beginning of slow-start and thus "in theory" shouldn't be sending out more than maybe two packets during the first round trip anyway. On the other hand, in practice I would think it should be acceptable to send at least 8-9KB worth of packets in an initial burst, the same as what happens when you send a multi-fragment UDP datagram containing an RPC request or reply at widely-accepted NFS message sizes, for example. Also, if SST was working in cooperation with a Congestion Manager on the local host, it might not have to slow start even a brand-new channel from scratch but could reuse congestion state from prior channels or even other transport protocols.

5.5.4 Computing Checksums

The responder can commence sending regular packets on the new channel as soon as it accepts a valid `SumR1Data` chunk and allocates the new state for the channel. The initiator can commence sending regular packets on the channel once it receives a valid `SumR1Data` response chunk with a nonzero `chanr` field, at which point it knows the correct receiver channel ID to include in the packets it sends. For greater network efficiency during channel setup, either side can additionally piggyback channel data packets into the negotiation packets they send, as described above.

To compute the 32-bit keyed MAC for a channel packet, the sender first computes a basic checksum over the pseudo-header and packet as described in Section 3.6 and illustrated in Figure 4, then "keys" the checksum by

```
checksum(int64 seqno,
         byte<> payload,
         int32 nonce):

  // Build data block on which to run checksum
  data <- concat(seqno, payload)
  data <- concat(data, "\0x80")
  if length(data) is odd:
    data <- concat(data, "\0x00")

  // Compute unkeyed checksum
  int32 a <- 0
  int32 b <- 0
  for each int16 word w in data:
    a <- (a + w) % 65537
    b <- (b + a) % 65537
  cksum <- (a & 0xffff) | ((b & 0xffff) << 16)

  // Computed keyed MAC based on ni or nr
  mac <- cksum + nonce
  return mac
```

Figure 20: 32-bit Lightweight Checksum Algorithm

adding its own 32-bit nonce: `ni` for channel packets sent by the initiator, and `nr` for channel packets sent by the responder. The receiver drops any packet it receives having an invalid checksum.

The checksum algorithm used is inspired by the Adler-32 algorithm [10], but modified to increase its entropy for small payloads, which are likely to be common in the case of SST's channel protocol. In essence, the modified algorithm operates on sequence of 16-bit words insstead of bytes, padded with a trailing 0x80 byte or 0x8000 word depending on whether the length of the input is odd or even, and it uses a prime modulus of 65537 instead of 65521 to ensure that all 2^{16} input data values are distinguishable. Since this is a 17-bit rather than a 16-bit prime modulus, the final checksum consists of the low 16 bits of each of the two resulting 17-bit counters. Pseudocode for the algorithm is shown in Figure 20.

As with Adler-32, the modulo operations can be removed from the inner loop by using larger counters and performing the modulo operation less frequently. With unsigned 32-bit counters, the modulo operation needs to be performed once every 361 words to prevent the counters from overflowing; with unsigned 64-bit counters, the modulo operation is necessary only once every 23,730,841 words. If the counters are signed, the period is 254 words for 32-bit counters and 16,781,438 words for 64-bit counters.

5.6 Reusing Channel IDs

XXX A given channel ID cannot be reused more than once every 8 milliseconds, to ensure that successive channel instances have distinct nonces.

6 Registration Protocol

SST's registration protocol provides a simple but efficient method for clients that are potentially mobile and/or connected by NATs to register securely with one or more designated servers in order to rendezvous with other clients. The registration protocol supports NAT traversal through UDP hole punching [15], and provides strong security by indexing clients by their self-assigned cryptographic SST identities rather than using forgeable or centrally-administered names.

To be written...

References

[1] William Aiello et al. Just Fast Keying: Key Agreement In A Hostile Internet. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):1–32, May 2004.

[2] M. Allman, V. Paxson, and W. Stevens. TCP congestion control, April 1999. RFC 2581.

[3] F. Audet, ed. and C. Jennings. Nat behavioral requirements for unicast udp, October 2006. Internet-Draft (Work in Progress).

[4] Hari Balakrishnan et al. TCP behavior of a busy Internet server: Analysis and improvements. In *IEEE INFOCOM*, March 1998.

[5] Hari Balakrishnan, Hariharan S. Rahul, and Srinivasan Seshan. An integrated congestion management architecture for Internet hosts. In *ACM SIGCOMM*, September 1999.

[6] S. Bellovin. Defending against sequence number attacks, May 1996. RFC 1948.

[7] E. Blanton and M. Allman. On making TCP more robust to packet reordering. *Computer Communications Review*, 32(1), January 2002.

[8] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *IEEE Journal on Selected Areas in Communication*, 13(8):1465–1480, October 1995.

[9] David D. Clark. Window and acknowledgement strategy in TCP, July 1982. RFC 813.

[10] P. Deutsch and J-L. Gailly. ZLIB compressed data format specification version 3.3, May 1996. RFC 1950.

[11] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. RFC 2246.

[12] D. Eastlake 3rd and T. Hansen. US secure hash algorithms (SHA and HMAC-SHA), July 2006. RFC 4634.

[13] R. Fielding et al. Hypertext transfer protocol — HTTP/1.1, June 1999. RFC 2616.

[14] Bryan Ford. Peer-to-peer (P2P) communication across network address translators (NATs), June 2004. Internet-Draft (Work in Progress).

[15] Bryan Ford. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[16] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance, May 1992. RFC 1323.

[17] S. Kent and R. Atkinson. Security architecture for the Internet Protocol, November 1998. RFC 2401.

[18] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (DCCP), March 2006. RFC 4340.

[19] M. Mathis, J. Mahdav, S. Floyd, and A. Romanow. TCP selective acknowledgment options, October 1996. RFC 2018.

[20] M. Mathis and J. Mahdavi. Forward acknowledgement: Refining TCP congestion control. In *ACM SIGCOMM*, August 1996.

[21] J. Postel. User datagram protocol, August 1980. RFC 768.

[22] E. Rescorla and N. Modadugu. Datagram transport layer security, April 2006. RFC 4347.

[23] R. Srinivasan. XDR: external data representation standard, August 1995. RFC 1832.

[24] R. Stewart et al. Stream control transmission protocol, October 2000. RFC 2960.

[25] Transmission control protocol, September 1981. RFC 793.

[26] J. Touch. TCP control block interdependence, April 1997. RFC 2140.

[27] Bin Ye, Anura P. Jayasumana, and Nischal M. Piratla. On monitoring of end-to-end packet reordering over the Internet. In *International Conference on Networking and Services*, July 2006.

[28] Ming Zhang, Brad Karp, Sally Floyd, and Larry Peterson. Improving TCP's performance under reordering with DSACK. Technical Report TR-02-006, International Computer Science Institute, July 2002.