Structured Streams:
A New Transport Abstraction

Bryan Ford
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

ACM SIGCOMM, August 30, 2007

http://pdos.csail.mit.edu/uia/sst/
Current Transport Abstractions

**Streams**
- Extended lifetime
- In-order delivery

Examples:
- TCP
- SCTP

**Datagrams**
- Ephemeral lifetime
- Independent delivery

Examples:
- UDP
- RDP
- DCCP
Simplistic Overview

The Problem:

- *Streams* don't quite match applications' needs
- *Datagrams* make the application do everything

The Solution:

- *Structured Streams*: like streams, only better
How Applications Use TCP

Natural approach: streams as transactions or application data units (ADUs) [Clark/Tennenhouse]

Example: HTTP/1.0

Web Client  TCP Stream  Web Server

GET  200 OK  GET

GET  200 OK  GET

GET  200 OK  GET
TCP Streams as Transactions/ADUs

Advantages:

- Reliability, ordering *within each ADU*
- Independence, parallelism *between ADUs*

☞ **Application-Layer Framing** [Clark/Tennenhouse]

Disadvantages:

- Setup cost: 3-way handshake per stream
- Setup cost: slow start per stream
- Shutdown cost: 4-minute TIME-WAIT period
- Network cost: firewall/NAT state per stream
- Network cost: unfair congestion control behavior
How Applications Use TCP

Practical approach: streams as sessions
TCP Streams as Sessions

Advantages:
- Stream costs amortized across many ADUs

Disadvantages:
- TCP's reliability/ordering applies across many ADUs
  
  **Unnecessary serialization:** no parallelism between ADUs
  **Head-of-line blocking:** one loss delays everything behind
  ⇒ TCP unusable for real-time video/voice conferencing
  ⇒ HTTP/1.1 made web browsers slower! [Nielsen/W3C]
- Makes applications more complicated
  Pipelined HTTP/1.1 still not widely used after 7 years!
What about Datagrams?

“Do Everything Yourself”:
- Tag & associate related ADUs
- Fragment large ADUs (> ~8KB)
- Retransmit lost datagrams (except w/ RDP)
- Perform flow control
- Perform congestion control (except w/ DCCP)

⇒ complexity, fragility, duplication of effort...
Structured Stream Transport

“Don't give up on streams; fix 'em!”

Goals:

- Make streams **cheap**
  - Let application use one stream per ADU, **efficiently**
- Make streams **independent**
  - *Preserve natural parallelism* between ADUs
- Make streams **easy to manage**
  - Don't have to bind, pass IP address & port number, separately authenticate each new stream
What is a Structured Stream?

Unix “fork” model for stream creation

Given parent stream s between A and B

- B listens on s
- A creates child s’ on s
- B accepts s’ on s
Talk Outline

- Introduction to Structured Streams
  - SST Protocol Design
  - Prototype Implementation
  - Evaluation, Related Work
  - Conclusion
SST Protocol Design
SST Transport Services

**Independent per stream:**
- Data ordering
- Reliable delivery (optional)
- Flow control (receive window)

**Shared among all streams:**
- Congestion control
- Replay/hijacking protection
- Transport security (optional)
SST Organization

Application Protocol

Stream Protocol

Channel Protocol

Negotiation Protocol

Underlying Protocol (e.g., UDP, IP, link layer)

Streams

Channels

Sessions

Structured Stream Transport (SST)
Streams, Channels, Packets

Streams
- Top-level Application Stream
  - Substream 1
    - 1.1
    - 1.2
    - ...
  - Substream 2
  - Substream 3
    - 3.1
    - 3.2

Channels
- Channel 1
  - multiplex streams onto channel 1
  - channel 1 nears end of life; migrate streams to channel 2
- Channel 2
  - multiplex streams onto channel 2

Packets
**SST Packet Header**

(Typical header overhead: 16 bytes + MAC)
Channel Protocol Design

- Sequencing
- Acknowledgment
- Congestion Control
- Security (see paper)
Channel Protocol: Sequencing

Every *transmission* gets new packet sequence #
- Including acks, retransmissions [DCCP]

![Diagram showing transmissions and arrivals with retransmissions marked]

Transmissions:
1  2  3  4  5  6  7  8  9

Arrivals:
1  3  6  4  5  8  9

(retransmit #2)
Channel Protocol: Acknowledgment

- All acknowledgments are *selective*  [DCCP]
  - No cumulative ack point as in TCP, SCTP
Channel Protocol: Acknowledgment

- All acknowledgments are **selective** [DCCP]
- Each packet acknowledges a **sequence range**

<table>
<thead>
<tr>
<th>Time</th>
<th>Packet Received</th>
<th>Acknowledgment Sent in Return Packet (acknowledged sequence number range)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>Ack 1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Ack 1–2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Ack 1–3</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>(packet 4 dropped)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Ack 5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Ack 5–6</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Ack 5–7</td>
</tr>
</tbody>
</table>

Sequence Number Space
Channel Protocol: Acknowledgment

- All acknowledgments are *selective* [DCCP]
- Each packet acknowledges a *sequence range*
  - Successive ACKs usually overlap
    ⇒ redundancy against lost ACKs
  - No variable-length SACK headers needed
    ⇒ all info in fixed header
Channel Protocol: Acknowledgment

- All acknowledgments are *selective* [DCCP]
- Each packet acknowledges a *sequence range*
- Congestion control at *channel granularity*
  - Many streams share congestion state
Stream Protocol Design

- Stream Creation
- Data Transfer
- Best-effort Datagrams
- Stream Shutdown/Reset (see paper)
- Stream Migration (see paper)
Stream Protocol: Creating Streams

Goal:
Create & start sending data on new stream \textit{without} round-trip handshake delay

Challenges:
1. What happens to subsequent data segments if initial “create-stream” packet is lost?
2. Flow control: may send how much data \textit{before} seeing receiver's initial window update?
Stream Protocol: Creating Streams

Solution:

- *All segments* during 1st round-trip carry “create” info (special segment type, parent & child stream IDs)
- Child *borrows* from parent stream's receive window (“create” packets belong to parent stream for flow control)
Stream Protocol: Data Transfer

Regular data transfer (after 1st round-trip):

- 32-bit wraparound byte sequence numbers (BSNs) (just like TCP)
- Unlimited stream lifetime (just like TCP)
Stream Protocol: Best-effort Datagrams

“Datagrams” are ephemeral streams

Semantically equivalent to:
1. Create child stream
2. Send data on child stream
3. Close child stream

...but without buffering data for retransmission
(like setting a short SO_LINGER timeout)
Stream Protocol: 
Best-effort Datagrams

When datagram is *small*:

- Stateless best-effort delivery optimization
  (avoids need to assign stream identifier to child)

![Diagram of Stream Protocol]

**Flags:**
- F  First Fragment
- L  Last Fragment
Stream Protocol: 
Best-effort Datagrams

When datagram is *small*:
- Stateless best-effort delivery optimization

When datagram is *large*:
- Fall back to delivery using regular child stream

*Makes no difference to application; datagrams of any size “just work”!*
Implementation & Evaluation
Current Prototype

User-space library in C++

- Application-linkable ⇒ simple deployment
- Runs atop UDP ⇒ NAT/firewall compatibility
- ~13,000 lines; ~4,400 semicolons
  (including crypto security & key agreement)

Available at:

http://pdos.csail.mit.edu/uia/sst/
Performance

Transfer performance vs native kernel TCP
- Minimal slowdown at DSL, WiFi LAN speeds

TCP-friendliness
- Congestion control fair to TCP within $\pm 2\%$

Transaction microbenchmark: SST vs TCP, UDP

Web browsing workloads
- Performance: HTTP on SST vs TCP
- Responsiveness: request prioritization
Transaction Microbenchmark

![Graph showing performance comparison between UDP, TCP, and SST protocols for transaction times with varying object sizes. The x-axis represents the size of the object transferred in bytes: 32B, 128B, 512B, 2K, 8K, 32K, 128K, 512K, 2M. The y-axis represents the request + response time in milliseconds: 10s, 6s, 4s, 2s, 1s, 600ms, 400ms, 200ms, 100ms, 60ms, 40ms. The graph shows that SST generally has the lowest request + response time across all object sizes.]
Web Browsing Workloads

Performance of transactional HTTP/1.0 on SST:

- Much faster than HTTP/1.0 on TCP
- Faster than persistent HTTP/1.1 on TCP [most browsers]
- As fast as pipelined HTTP/1.1 on TCP [Opera browser]
Web Browsing Workloads

HTTP/1.0 over SST can be *more responsive*
- No unnecessary request serialization
- Simple out-of-band communication via substreams

» Easy to *dynamically prioritize* requests

(Demo)
Related Work

- **Application-Layer Framing** [Clark/Tennenhouse]
- Transports: **TCP, RDP, VMTP, SCTP, DCCP**
- Multiplexers: **SSL, SSH, MUX, BXXP/BEEP**
- **T/TCP: TCP for Transactions** [Braden]
- **TCP congestion state sharing** [Touch], **Congestion Manager** [Balakrishnan]
- Transport-layer **migration** support [Snoeren]
- Network-layer **prioritization** for QoS [...many...]
Conclusion

SST enables applications to use streams as:

- **Sessions** (as in legacy TCP apps), or
- **ADUs/Transactions** (as in HTTP/1.0), or
- **Datagrams** (as in VoIP, RPC over UDP)

...without:

- TCP's per-stream costs, unnecessary serialization
- UDP's datagram size limits

http://pdos.csail.mit.edu/uia/sst/
“Can't HTTP/1.1 over TCP do this?”

Answer: “Sort of, if you work really hard.”

1. Enable HTTP/1.1 pipelining
   - Most browsers still don't because servers get it wrong!

2. Fragment large downloads via **Range** requests
   - Pummel server with many small HTTP requests
   - Risk atomicity issues with dynamic content

3. Track **round-trip time, bandwidth** in application
   - Try to keep pipeline full without adding extra delay

But:

Still get **head-of-line blocking** on TCP segment loss!
Comparing SST to SCTP

**SCTP:**
- No dynamic stream creation/destruction
- No per-stream flow control (just per session)
- Best-effort datagrams limited in size

**SST:**
- No multihoming/failover (yet)
  ...but channel/stream split should facilitate
Comparing SST to DCCP

DCCP:
- No reliability, ordering, flow control
- No association between packets
- No cryptographic security

SST:
- No congestion control negotiation (yet)
Channel Protocol: Security

Design based on **IPsec**

- **Cryptographic security mode**: Encrypt-then-MAC + replay protection [IPsec]
- **TCP-grade security mode**: No encryption
  - MAC = 32-bit checksum + 32-bit “key”
  - depends on system time [Tomlinson], secret data [Bellovin]

**stronger protection than TCP**: “validity window” size = 1