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

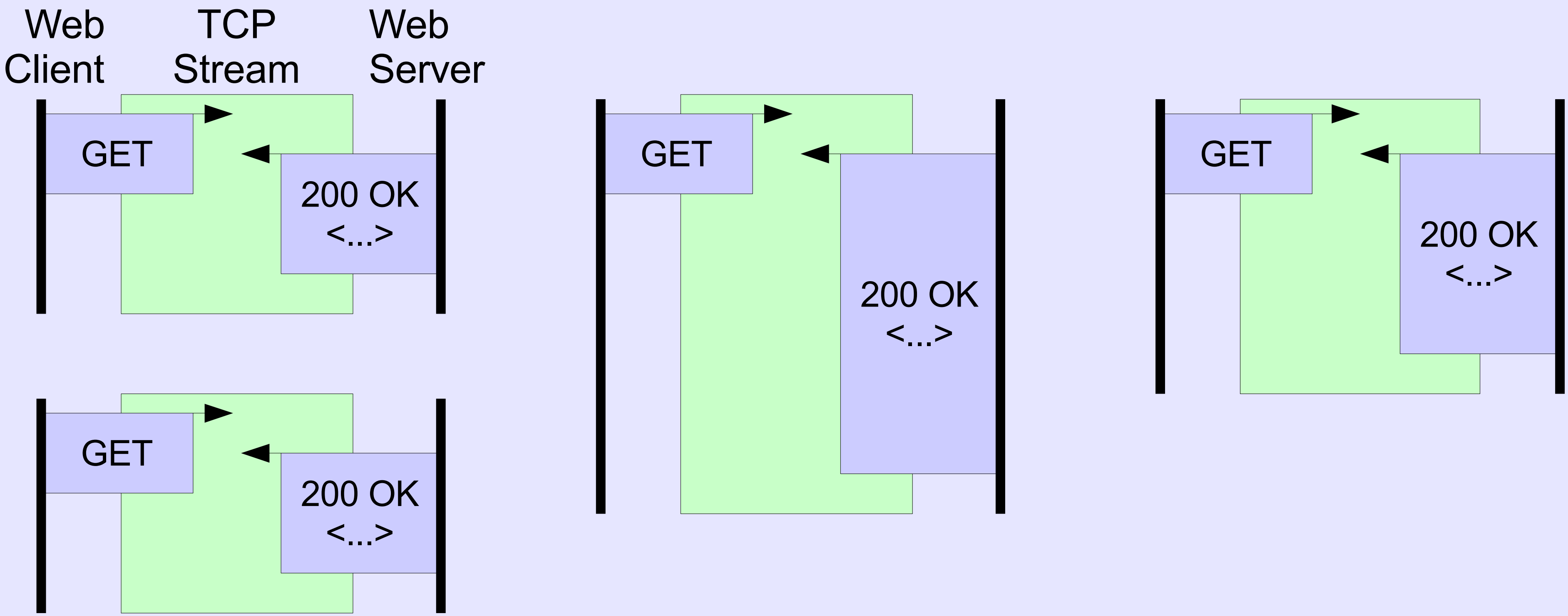
The Solution:

- ***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



TCP Streams as Transactions/ADUs

Advantages:

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

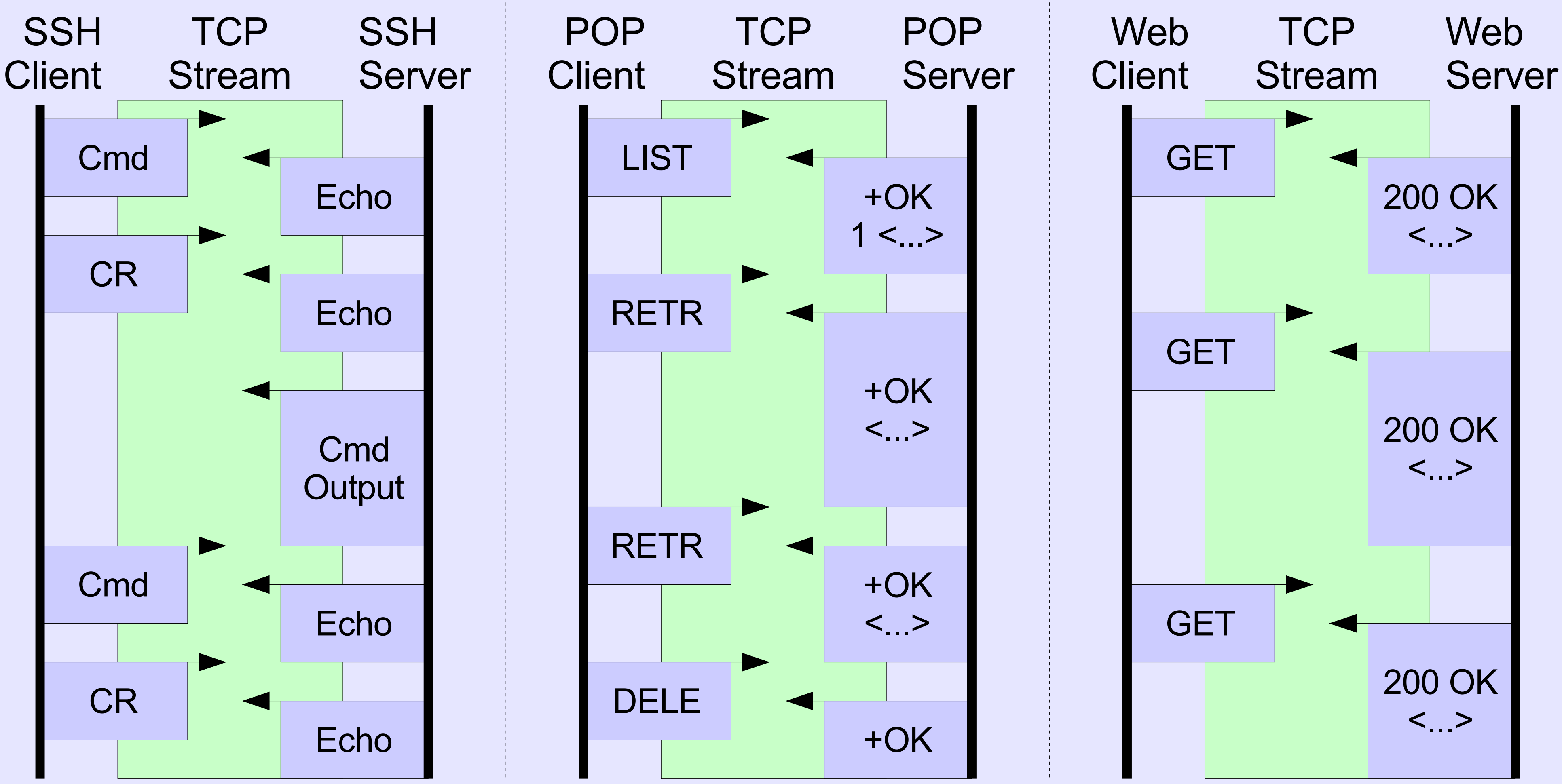
☛ **Application-Layer Framing** [Clark/Tennenhouse]

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)

$\Rightarrow$ **complexity, fragility, duplication of effort...**

Structured Stream Transport

“Don't give up on streams; fix 'em!”

Goals:

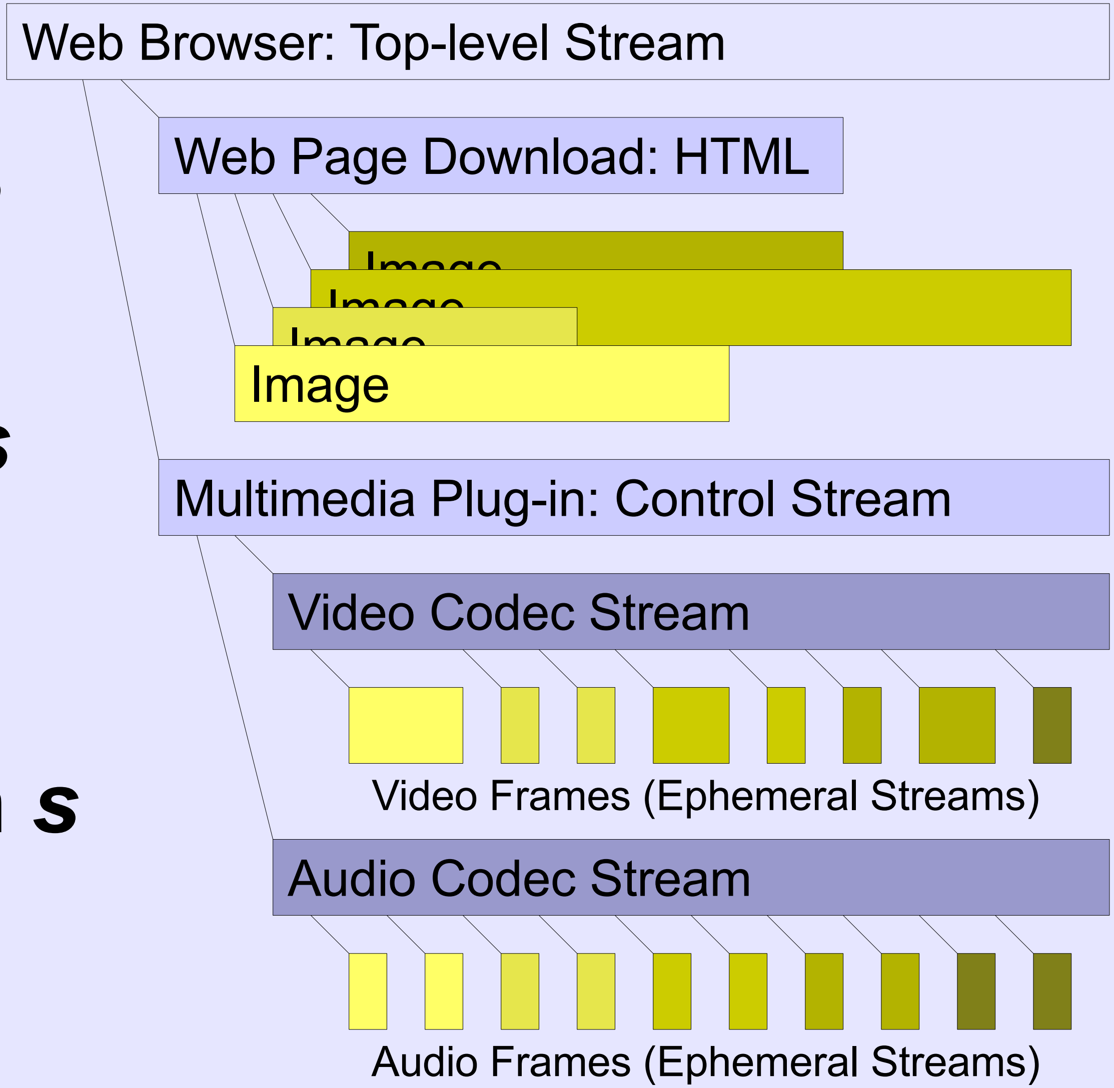
- Make streams **cheap**
  - Let application use one stream per ADU, *efficiently*
- Make streams **independent**
  - *Preserve natural parallelism* between ADUs
- Make streams **easy to manage**
  - Don't have to bind, pass IP address & port number, separately authenticate each new stream

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

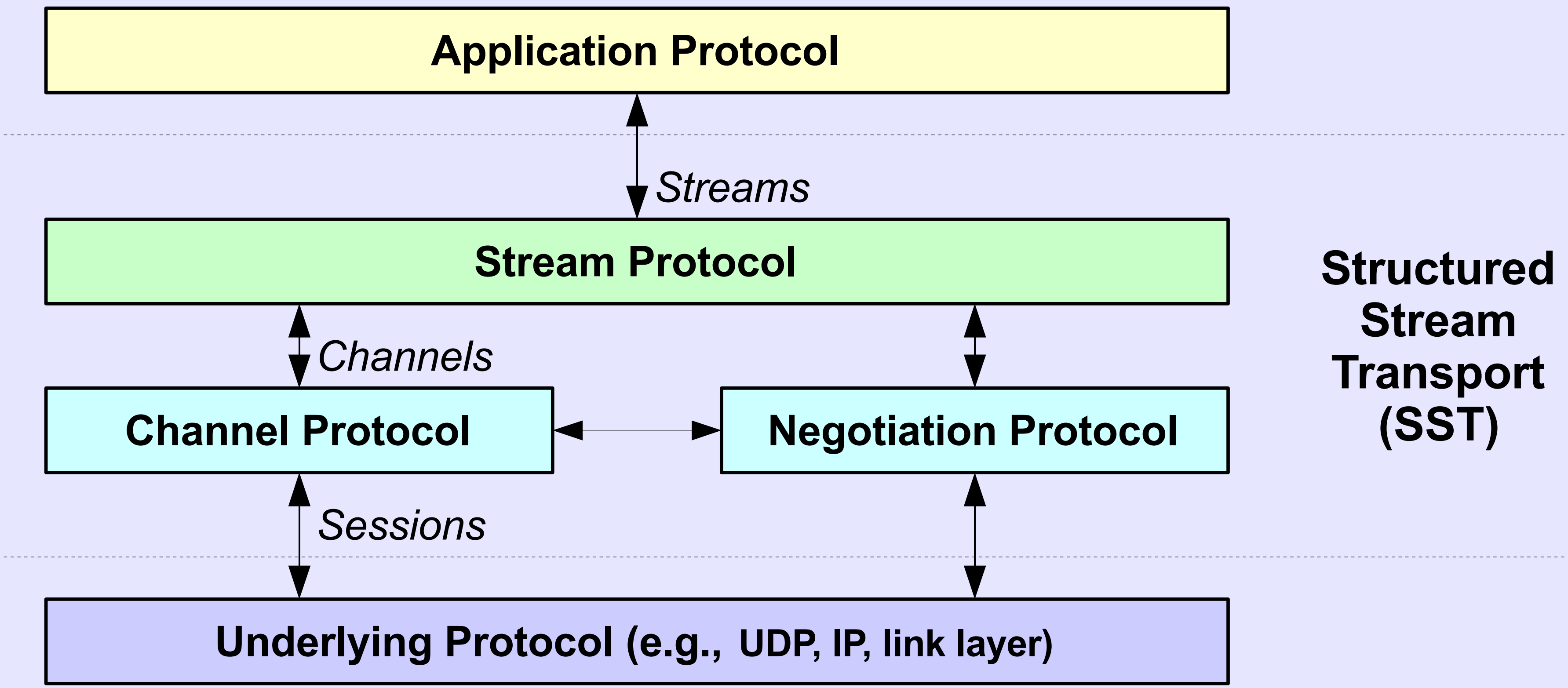
Independent *per stream:*

- Data ordering
- Reliable delivery (optional)
- Flow control (receive window)

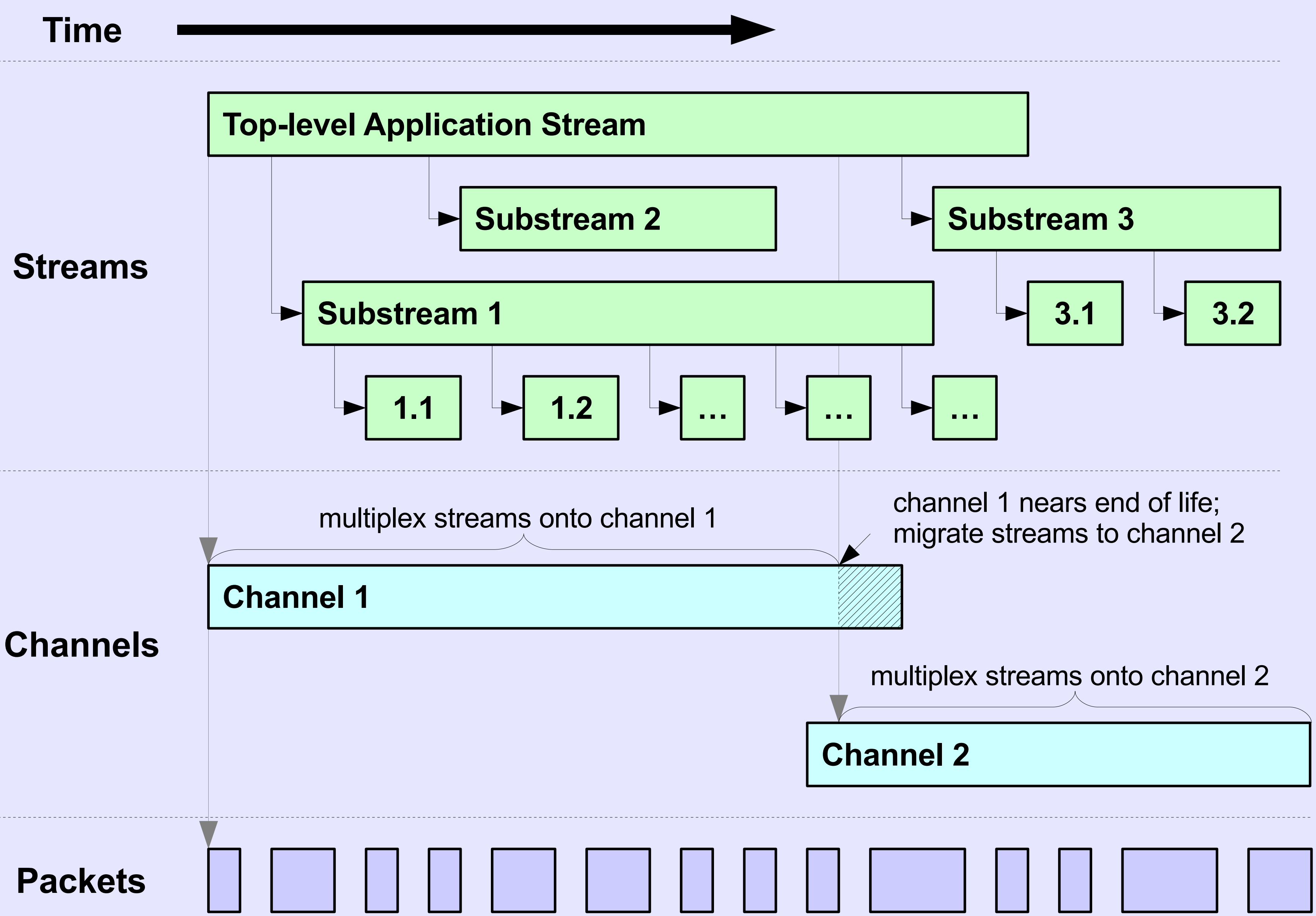
Shared *among all streams:*

- Congestion control
- Replay/hijacking protection
- Transport security (optional)

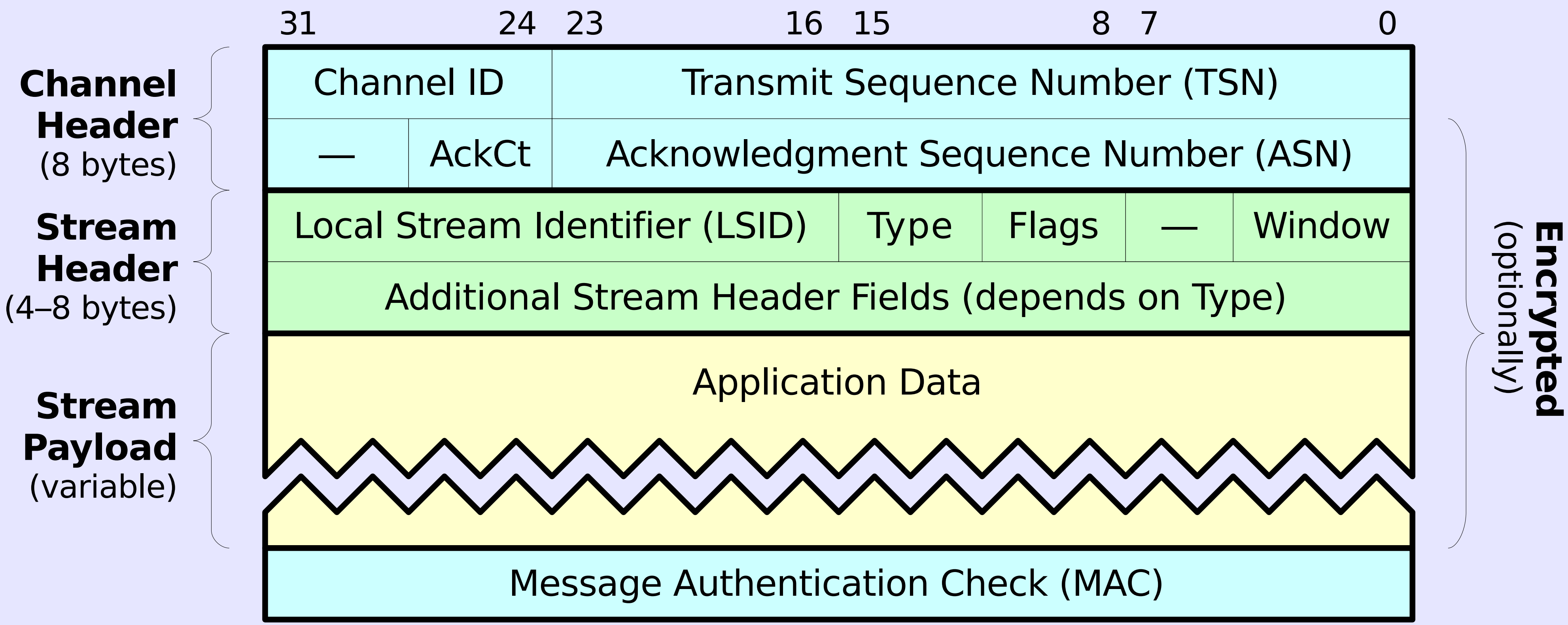
SST Organization



Streams, Channels, Packets



SST Packet Header



(Typical header overhead: 16 bytes + MAC)

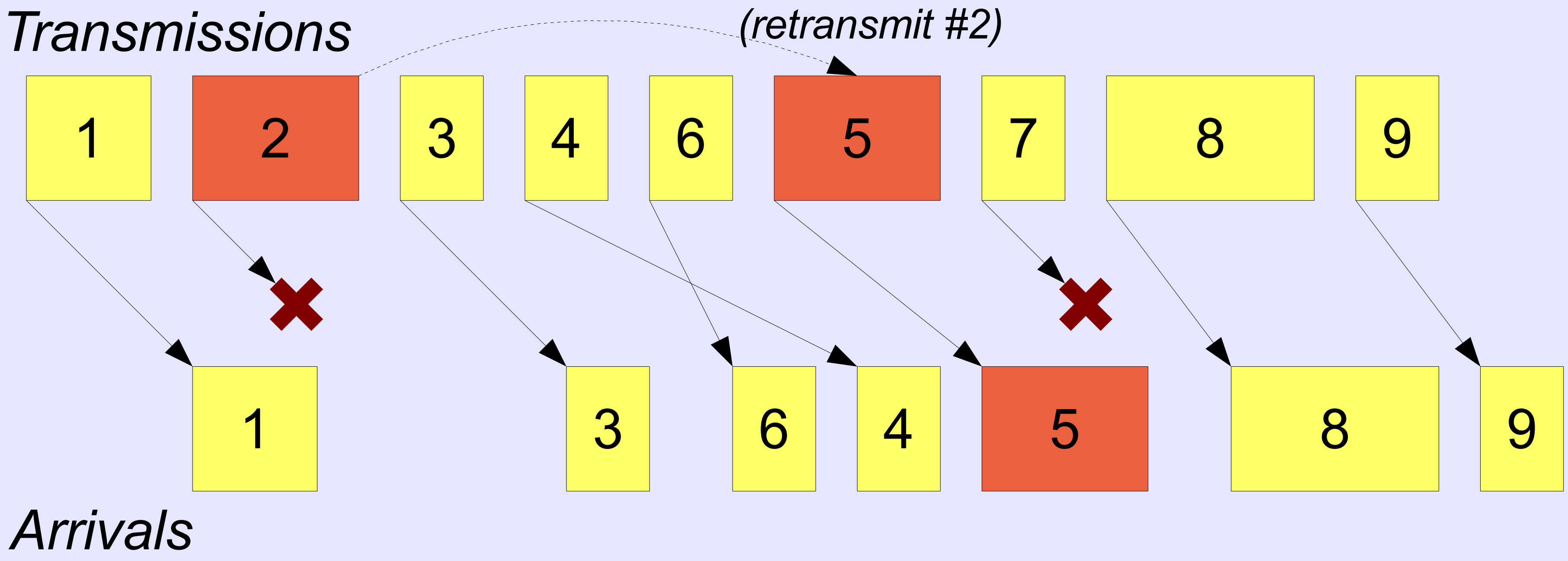
Channel Protocol Design

- Sequencing
- Acknowledgment
- Congestion Control
- Security (see paper)

Channel Protocol: Sequencing

Every *transmission* gets new packet sequence #

- Including acks, retransmissions [DCCP]

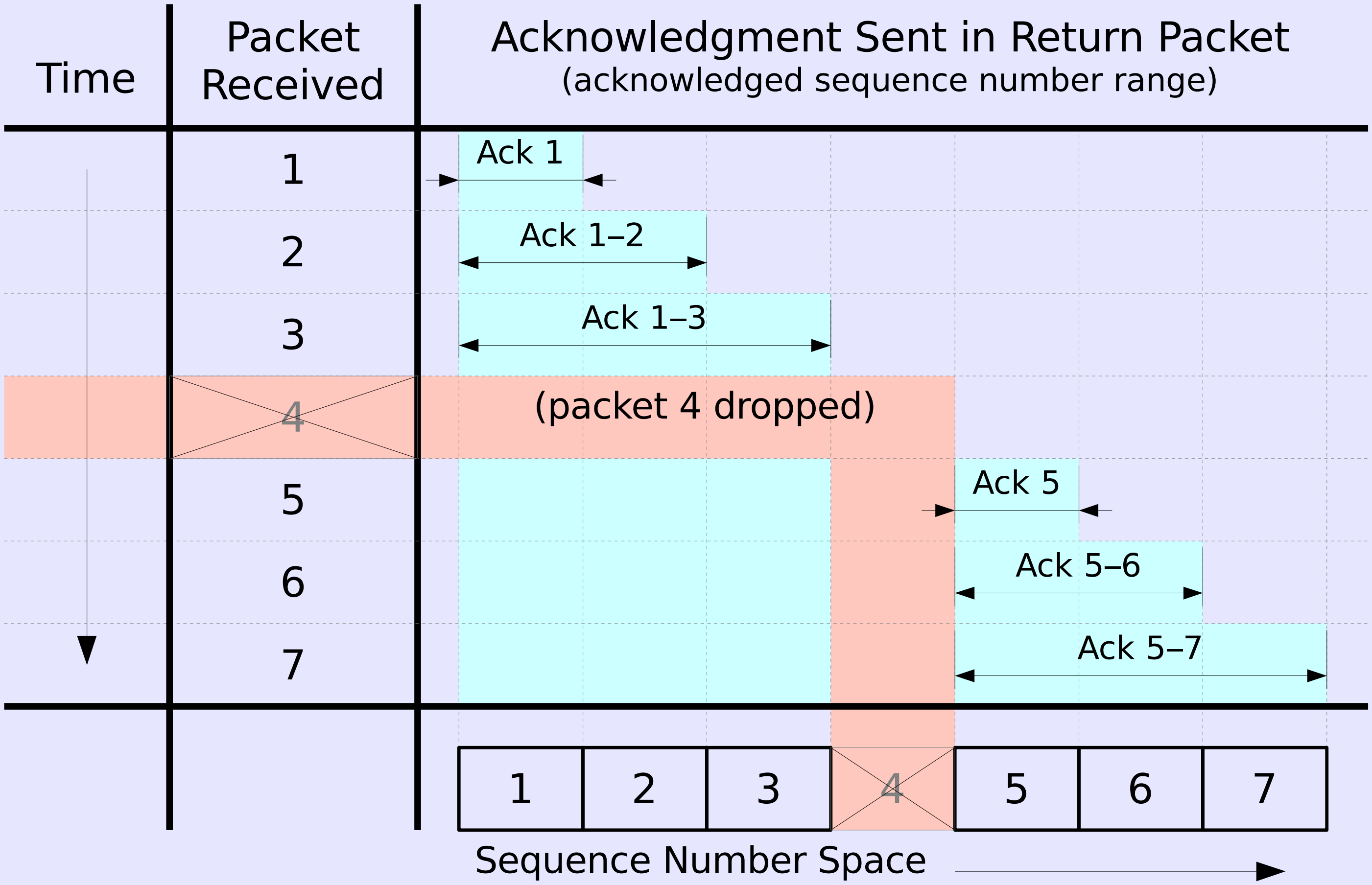


Channel Protocol: Acknowledgment

- 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*



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 *without* round-trip handshake delay

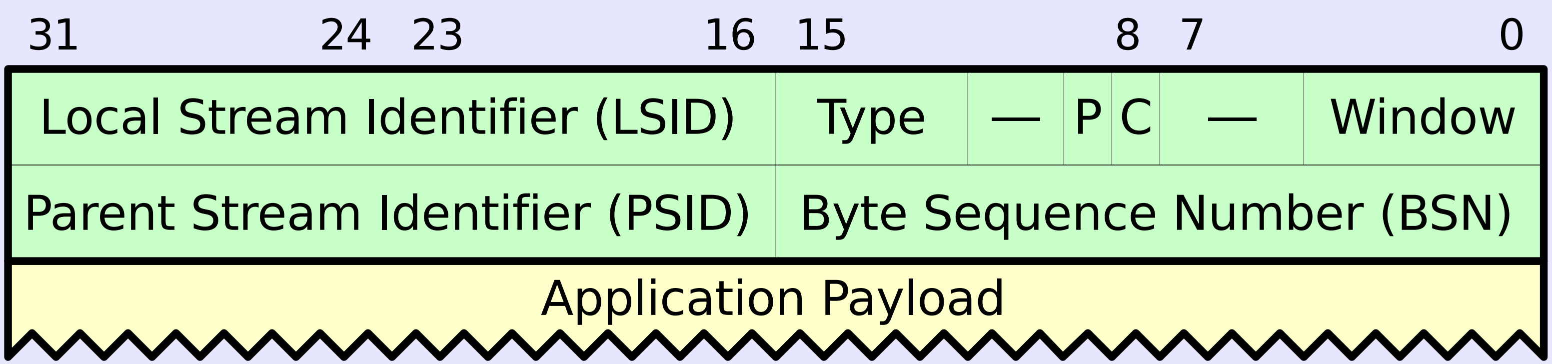
Challenges:

1. What happens to subsequent data segments if initial “create-stream” packet is lost?
2. Flow control: may send how much data *before* seeing receiver's initial window update?

Stream Protocol: Creating Streams

Solution:

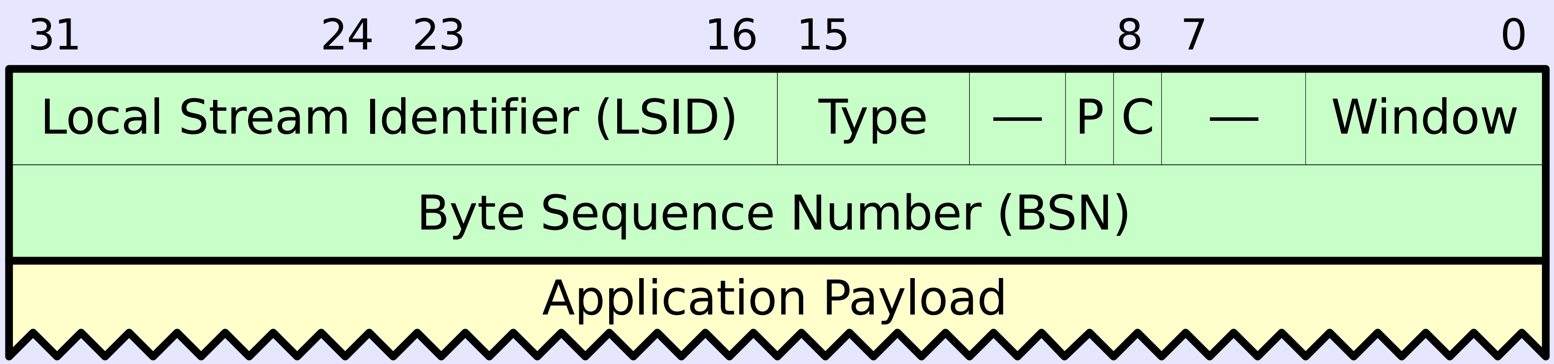
- *All segments* during 1st round-trip carry “create” info
  (special segment type, parent & child stream IDs)
- Child *borrows* from parent stream's receive window
  (“create” packets belong to parent stream for flow control)



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

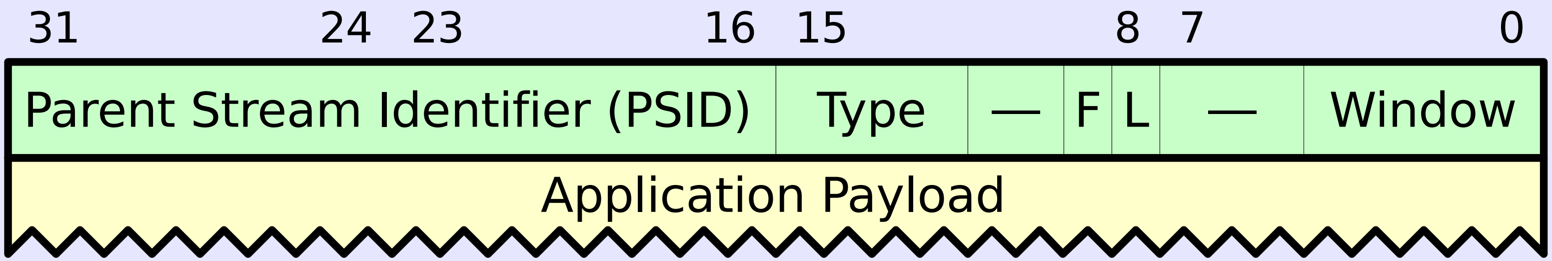
(like setting a short SO_LINGER timeout)

Stream Protocol: Best-effort Datagrams

When datagram is *small*:

– Stateless best-effort delivery optimization

(avoids need to assign stream identifier to child)



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)

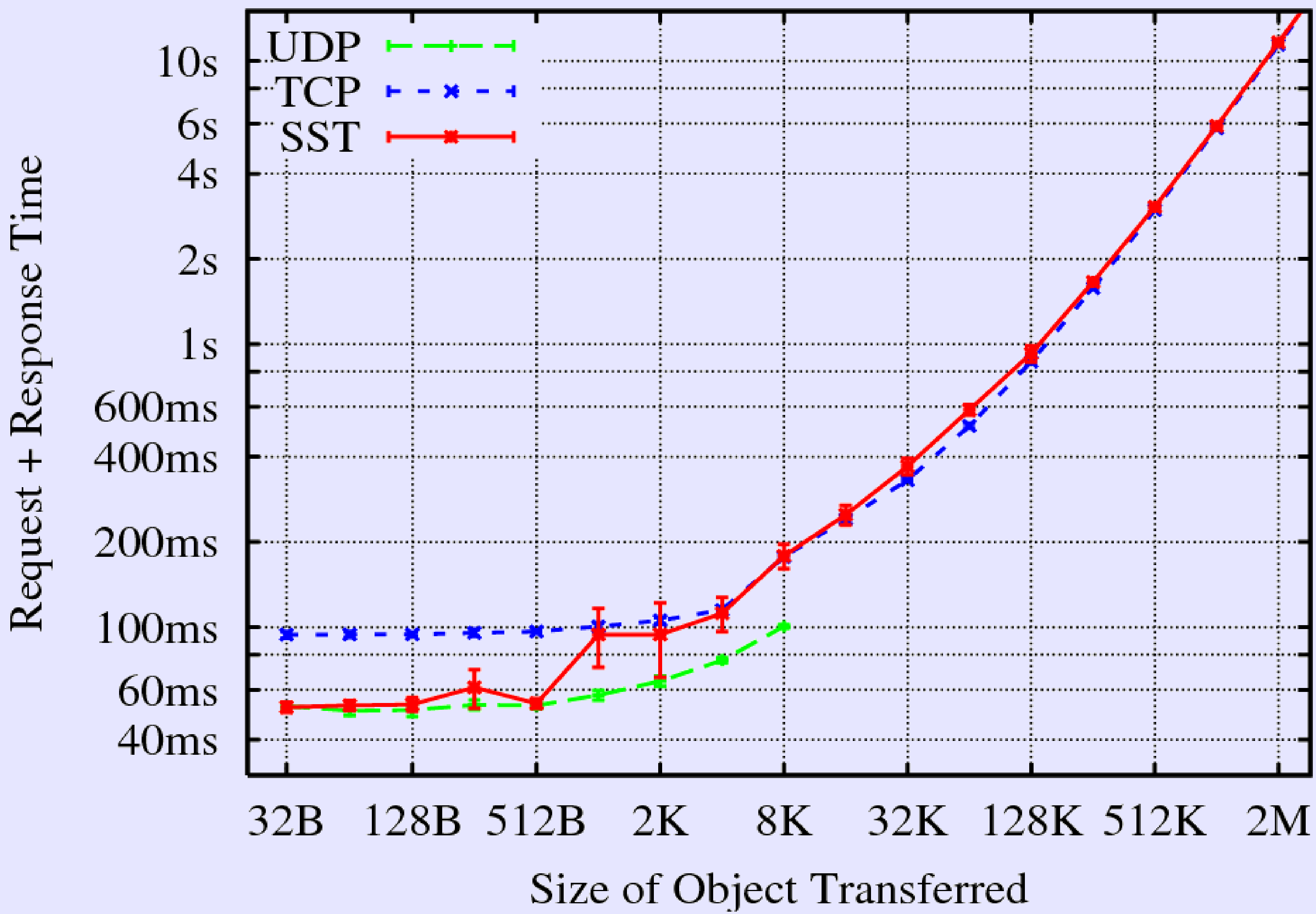
Available at:

`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

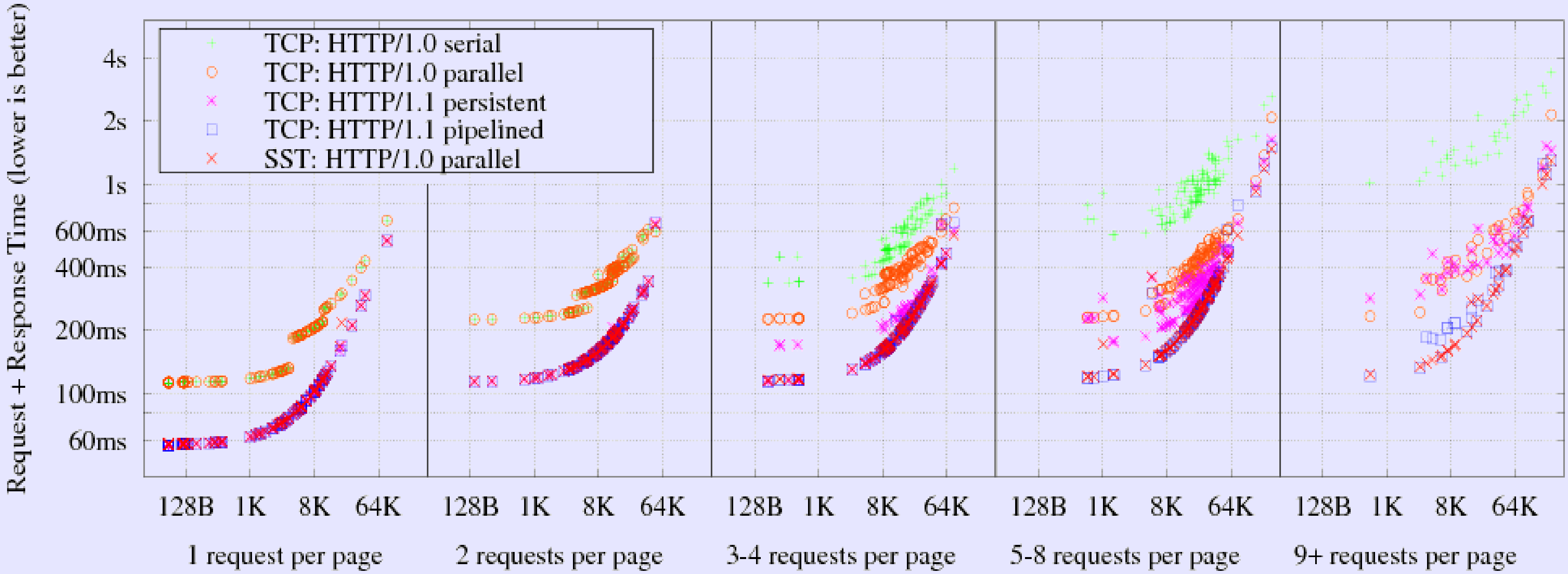
Transaction Microbenchmark



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*...]

Summary

<u>A New Transport Mindset</u>

TCP: *“think serial”*

SST: *“think parallel”*

Future Work

Lots of stuff to do

- Fill holes in spec, code
- Efficient implementation(s)

Protocol improvements/extensions

- Fat headers for high-BDP paths
- Chunk bundling
- “Widening the endpoints”: multihoming, etc.

High Bandwidth-Delay Product Paths

w/ Regular Headers

- ~2^{22} packets...
- ~2^{15} new streams...
- ~2^{30} stream bytes...
- ...per round-trip

w/ Fat Headers

- ~2^{46} channel packets...
- ~2^{23} new streams...
- ~2^{46} stream bytes...
- ...per round-trip

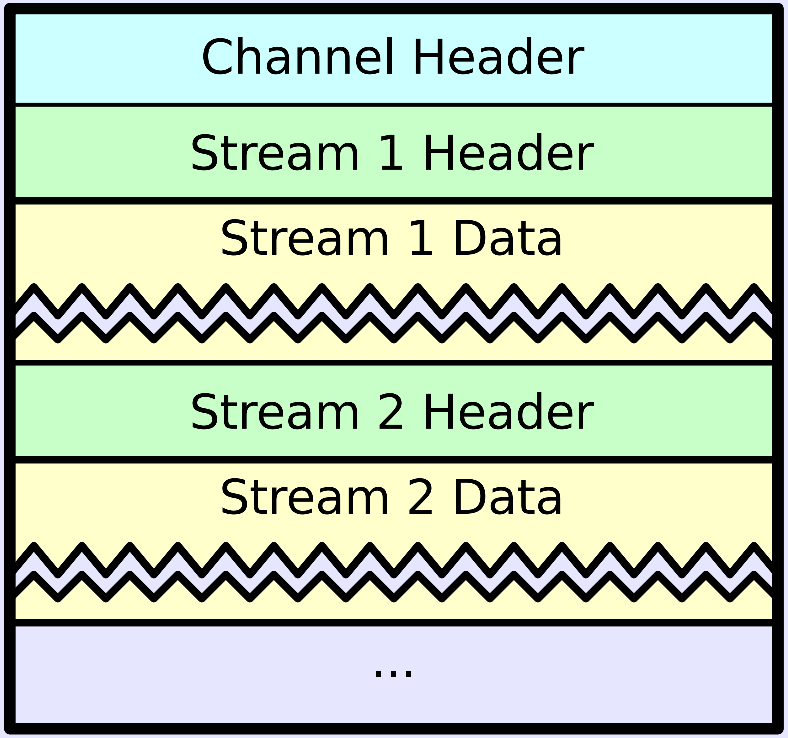
<table>
<tr><td colspan="2">Channel ID</td><td colspan="4">Transmit Sequence Number (TSN)</td></tr>
<tr><td>—</td><td>AckCt</td><td colspan="4">Acknowledgment Sequence Number (ASN)</td></tr>
<tr><td colspan="2">Local Stream Identifier (LSID)</td><td>Type</td><td>Flags</td><td>—</td><td>Window</td></tr>
<tr><td colspan="6">Additional Stream Header Fields (depends on Type)</td></tr>
</table>

<table>
<tr><td colspan="3">Channel ID</td><td colspan="2">TSN high bits</td></tr>
<tr><td colspan="5">Transmit Sequence Number (TSN) low bits</td></tr>
<tr><td colspan="2">—</td><td colspan="2">AckCt</td><td>ASN high bits</td></tr>
<tr><td colspan="5">Acknowledgment Sequence Number (ASN) low bits</td></tr>
<tr><td>Type</td><td>Flags</td><td colspan="3">Local Stream Identifier (LSID)</td></tr>
<tr><td colspan="2">Window</td><td colspan="3">Parent Stream Identifier (LSID)</td></tr>
<tr><td colspan="3">Chunk Size</td><td colspan="2">BSN high bits</td></tr>
<tr><td colspan="5">Byte Sequence Number (BSN) low bits</td></tr>
</table>

Chunk Bundling

Bundle segments from multiple streams into one channel packet

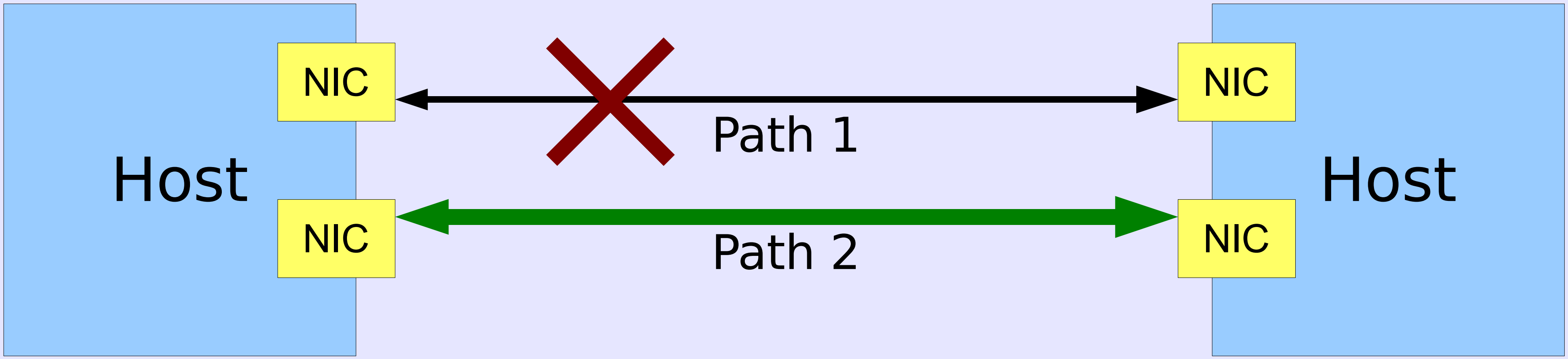
- e.g., VoIP trunking, multiplayer gaming, etc.



Widening the Endpoints

Multihoming: multiple interfaces, multiple paths

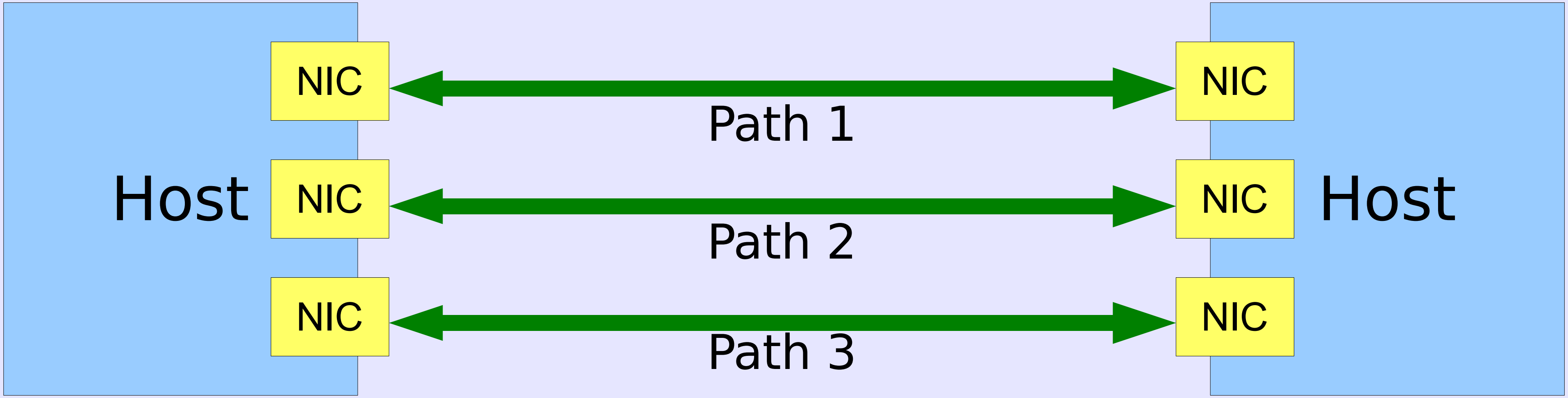
- Redundancy: fail-over across paths [SCTP]



Widening the Endpoints

Multihoming: multiple paths per logical host

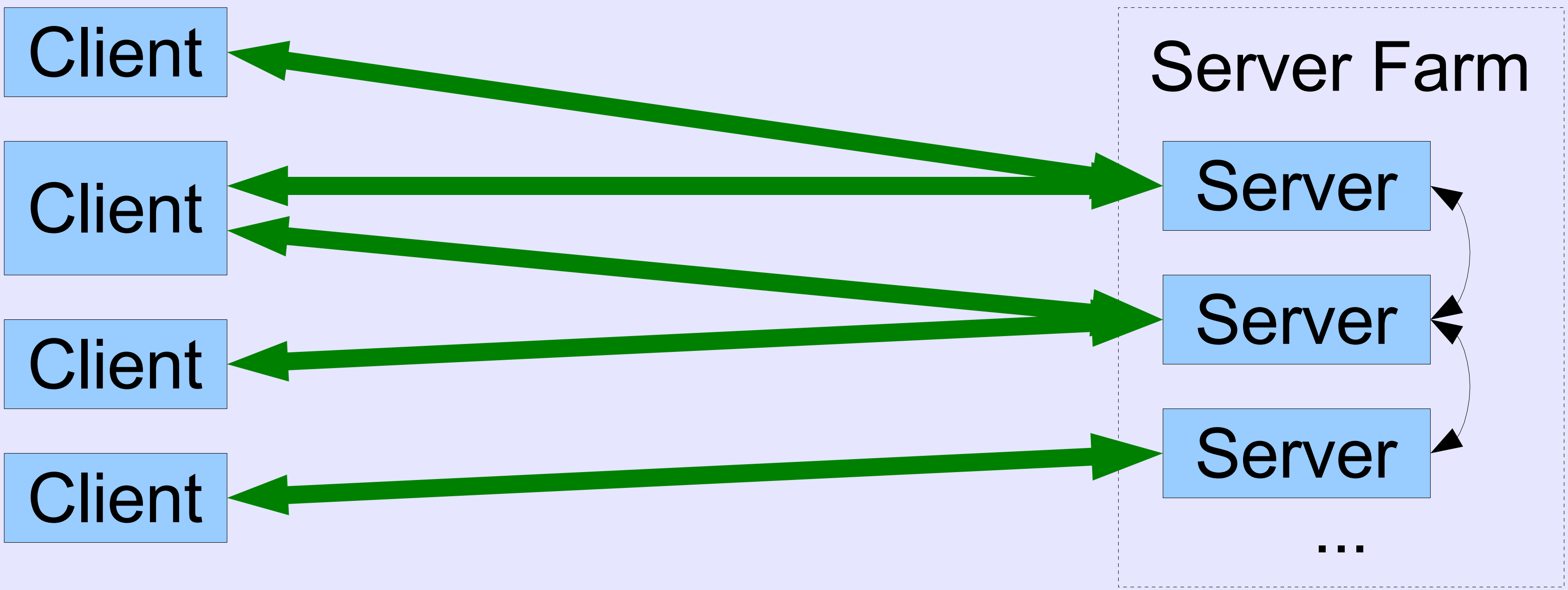
- Redundancy: fail-over across paths [SCTP]
- Parallelism: sharing load across paths



Widening the Endpoints

Multihoming: multiple paths per logical host

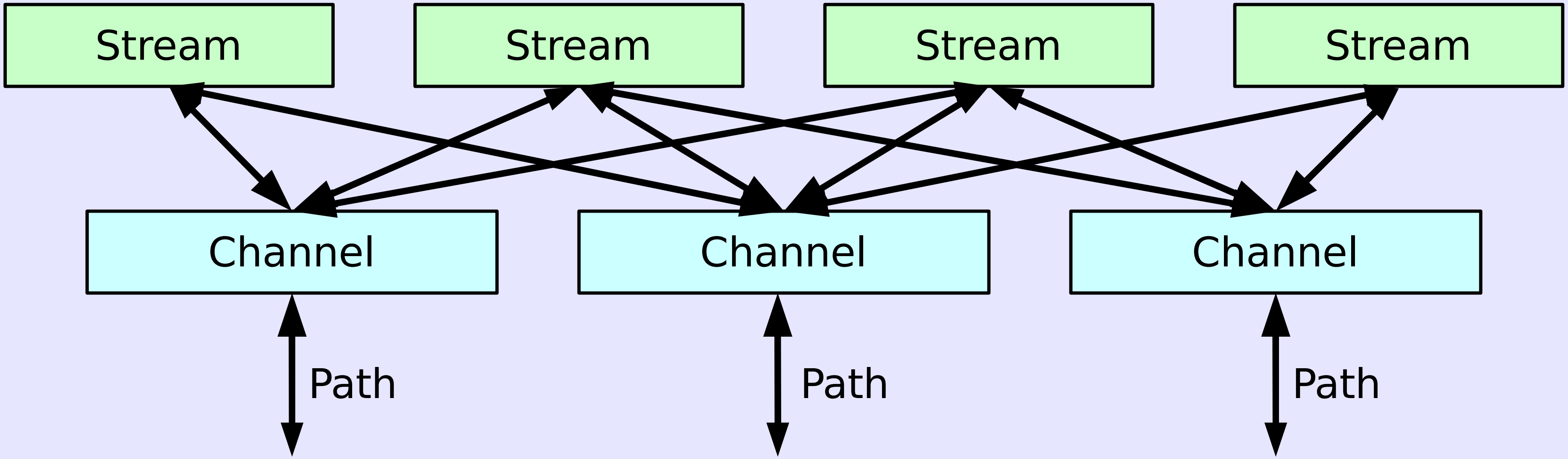
- Redundancy: fail-over across paths [SCTP]
- Parallelism: sharing load across paths
- Scaling: clustered/distributed implementations?



Widening the Endpoints

Facilitated by SST's design:

- **Channel** represents physical path
- **Stream** represents logical transaction/activity
- *Many-to-many* relationship



Widening the Endpoints

Facilitated by SST's design:

- **Channel** represents physical path
- **Stream** represents logical transaction/activity
- *Many-to-many* relationship

Applications use many streams, not just one
>> *fewer inherent concurrency bottlenecks*

Congestion control is per-channel/path,
>> *no confusion from varying path delays*

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