Moving the Internet’s Transport Layer Ahead

Michael Welzl
Part 1: the problem
It can’t be changed.

• Internet transport layer = TCP (1981), UDP (1980)
  – Service = what these protocols provide
  – Does not match the diversity of today’s applications

• Probably only two truly significant (noticeable for users) changes:
  1. Addition of congestion control to TCP: 1988
  2. Change of default TCP CC. in Linux to BIC: 2004
     (a bit later: CUBIC) ... not IETF-approved!
Many more standards exist

• Getting deployed:
  – Many, many TCP bug fixes

• Hardly getting deployed:
  – New protocols: SCTP, DCCP

• Newer things - can’t evaluate deployment yet (but don’t want this to end up “in the red” !)
  – LEDBAT, MPTCP…
SCTP and DCCP in a nutshell

• SCTP: TCP++ ... mostly by removing “features”!
  – TCP without stream semantics, requirements for ordered or reliable delivery
  – and a few features added, e.g. multistreaming (ordered delivery within streams only) and multihoming

• DCCP: congestion control for real-time (multimedia) applications
  – various CCID specs define different CC behaviors
  – e.g. TFRC for smoother rate (less jitter)
What’s wrong?

• Internet design is inherently insecure
  – hence, tendency to disable/block things that look “strange”
  – TCP, UDP, and special applications, i.e. port numbers, are considered acceptable; everything else is “strange”
    ➔ Application programmers don’t use other transport protocols

• Design was supposed to be open...
  – “Be conservative in what you send, be liberal in what you accept”
  – Reality is different (Deep Packet Inspection, ..)
Internet design flaw: no abstraction

- OSI had the right idea! :-) abstraction.
  - Layers merely provide a service
  - Lower layers + their internal operation hidden → could be replaced
- Transport layer should be especially easy to change!

Source: A. Tanenbaum, Computer Networks
A better Internet transport design

A more abstract transport API

1. Applications say...
   - what kind of service they prefer
   - what kind of traffic they will generate

2. Using its resources (protocols, signaling with the inner network, ...), the transport layer does its best (still best effort!) to provide a good service
   - Could try a new protocol, and give up in case of failure
   - Could maybe also say: “you’re even getting a guarantee here!”
A better Internet transport design /2

• This has been said before

• The problem might not have occurred with this...
  – but this doesn’t help us now.
  – so how can we get there?
What is needed

• Make it attractive to use new protocols
  1. show benefits
     • surprisingly little done so far; some serious implementation issues (e.g. with Linux SCTP)
  2. make it easy to use them
     • minimal change to applications, or even no change at all

• Make sure that there’s no harm in trying them
  – Downward compatibility: fall back to TCP/UDP
  – Use tricks to get packets across
Part 2: things we can do
Transparent beneficial deployment of SCTP


Underlying idea

- SCTP is already (somewhat) attractive
  - resilience can improve if used transparently (automatically use multihoming)

- Can get more benefits via transparent usage: using multi-streaming
  - map short TCP connections onto long SCTP association, exploit large congestion window when this is beneficial
Step 1: general performance check

![Throughput measurement with iperf in 100 Mbit/s network (10 sec)](chart)
Step 2: implementation

**SCTP association with multi-streaming**
- map each connection on different stream
- message based data transmission
- shared flow control
- shared congestion control

**benefits**
- subsequent data transfers have new cwnd - value
- faster startup if association already exists
- multihoming only active if demanded or beneficial

**TCP connection**
- connect to GW
- bytestream transfer
- flow control
- congestion control

**Connection manager gateway**
- connection attempt management
- setup SCTP association
- read/write from TCP connection
- read/write on SCTP association
- open/close new TCP connection
- gateway signaling protocol

original TCP connection (possible to bypass the gateway)
Step 3: Test

![Diagram showing network setup and test results]

- **Endhost A**
  - 100 Mbit
  - 100 Mbit Switch
  - Client applications
  - Network monitoring (wireshark)

- **Linux Router**
  - 100 Mbit
  - Netem (Network Emulator) generates constant path delays

- **Endhost B**
  - 100 Mbit
  - 100 Mbit Switch
  - Server applications
  - Network monitoring (wireshark)

### Test Results

- **File 1 (64M)**
  - Time: 6.614 s
  - SCTP Gateway: 6.317 s
  - TCP: 4.522 s

- **File 2 (1M)**
  - Time: 6.614 s
  - SCTP Gateway: 6.317 s
  - TCP: 4.522 s

- **Total**
  - Time: 7.022 s
  - SCTP Gateway: 6.614 s
  - TCP: 4.522 s

- **Graph Legend**
  - SCTP Gateway
  - TCP
Conclusion from SCTP experiment

• Doing this right is probably worth it, but it’s hard
  – kernel implementation required
  – fixes to SCTP required
    • per-stream flow control, improving SCTP performance via auto-buffer-tuning and pluggable congestion control
  – protocol setup / TCP fall-back mechanism required
  – either decide when to map (hard / ugly?) or let SCTP with multiple streams be more aggressive (like MulTCP); research required for doing this right
Trying to use a new protocol with TCP as a fall-back


[Discussions with Bryan Ford, Jana Iyengar, Michael Tüxen, Joe Touch]
Just try!

• Happy Eyeballs
  – most prominent (and maybe most realistic) among several suggested methods
  – Originally proposed for IPv6 / IPv4 and SCTP / TCP, then split into two drafts
  – We focus on SCTP / TCP (and keep DCCP in mind)

• Algorithm:
  – Send a TCP SYN + SCTP INIT; use the first answer
  – Optional: delay TCP processing a bit in case TCP SYN/ACK arrives before SCTP INIT-ACK
The early-TCP-SYN-ACK-problem

• When Florian started his thesis, he tried this, and he told me that TCP always returned first
  – Obvious: more efficient processing in server

• How to cope with this?
  – Ignore late SCTP INIT-ACK: SCTP often not used
  – Switch to SCTP on-the-fly: does not seem feasible
  – Delay TCP processing: always affects TCP connections
  – Use SCTP for later connections: delayed effect of SCTP, must know that there will be later connections
An idea: “NOT-TCP” option / bit in TCP SYN

Case 1: New server
- SCTP INIT, TCP SYN: “NOT-TCP”
- TCP SYN-ACK: “NOT-TCP”
- SCTP INIT-ACK
- WAIT
- SCTP COOKIE-ECHO

Case 2: Old server
- SCTP INIT, TCP SYN: “NOT-TCP”
- TCP SYN-ACK
- TCP ACK
- SCTP INIT-ACK

Ignore, or use for next connection
Not-TCP concerns

• Encoding issues
  – TCP option space limited
  – Overloading a bit (e.g. CWR): will middle-boxes kill such TCP SYNs? Bit overloading also limits protocol choice – e.g. option can carry “use any other protocol from this list”
  – even processing an unknown TCP option is work (might be a problem for a busy server)
    ➔ to be evaluated – state needed?

• Not-TCP binds TCP+SCTP/DCCP/.. ports together
NOT-TCP and state

- Both sides can immediately tear down all TCP state after exchanging “Not-TCP” and an SCTP packet
- (Only?) advantage over TCP SYN Cookies: no need for teardown (FIN / FIN/ACK)
Better negotiation for later connections?

• Simply trying protocol x instead of y may not be good enough in the long run
  – Scalability concerns when we try:
    SIP/TLS/DCCP/IPv6 vs. SIP/UDP/IPv4 vs. ....

• Hosts should be able to *negotiate* the protocol combination; proposals exist
  – separate protocol w/ preference graphs (HotNets Ford / Iyengar)
  – signaling over HTTP (draft-wood-tae-specifying-uri-transports-08)
  – out-of-band (e.g. DNS server) [Iyengar @ Hiroshima IETF]

• but: must also try if chosen combination works!
Towards a Protocol-Independent Transport API

[Stefan Jörer: A Protocol-Independent Internet Transport API, MSc. thesis, University of Innsbruck, December 2010]

Two approaches

• Top-down: start with application needs ("QoS-view")
  + flexible, good for future use of new protocols
  – loss of service granularity: some protocol features may never be used
  – been there, done that, achieved nothing...

• Bottom-up: unified view of services of all existing Internet transport protocols
  + No loss of granularity, all services preserved
  + Totally new approach, concrete design space
  – May need to be updated in the future
Our chosen design method

• **Bottom-up**: TCP, UDP, SCTP, DCCP, UDP-Lite
  – start with lists from key references

• **Step 1**: from list of protocol features, carefully identify application-relevant services
  – features that would not be exposed in APIs of the individual protocols are protocol internals (e.g. ECN)

• Result: table with a line for every possible combination of features
  – 43 lines: 32 SCTP, 3 TCP/UDP
Our chosen design method

• **Step 2:** carry out obvious further reductions
  – e.g. flow control coupled with congestion control
  – duplicates, subsets

• Apply common sense to go beyond purely mechanical result of step 1
  – Question: would an application have a reason to say “no” to this service under certain circumstances?
  – Features that are just performance improvements if they are used correctly (i.e. depending on environment, not app) are **not** services
# Result of Step 2

x = always on  
empty = never on  
P1 = partial error detection  
t = total reliability  
p2 = partial reliability  
o = ordered  
u = unordered

<table>
<thead>
<tr>
<th>service no.</th>
<th>flow characteristic</th>
<th>app. PDU bundling</th>
<th>error detection</th>
<th>reliability</th>
<th>delivery</th>
<th>order</th>
<th>multi-homing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TCP-like</td>
<td></td>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>TCP-like</td>
<td></td>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Smooth</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Smooth</td>
<td></td>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Smooth-SP</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Smooth-SP</td>
<td></td>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>t</td>
<td>o</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>t</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>t</td>
<td>u</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>18</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>u</td>
<td>x</td>
</tr>
<tr>
<td>20</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>21</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td>x</td>
</tr>
<tr>
<td>22</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td>x</td>
</tr>
<tr>
<td>23</td>
<td>TCP-like</td>
<td></td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td>x</td>
</tr>
</tbody>
</table>
API Design

• Goal: make usage attractive = easy
  – stick with what programmers already know: deviate as little as possible from socket interface
• Most services chosen upon socket creation
  – int socket(int domain, int service)
  – service number identifies line number in table
  – understandable aliases: e.g. PI_TCPLIKE_NODELAY, PI_TCPLIKE, PI_NO_CC_UNRELIABLE for lines 1-3
• Sending / receiving: provide sendmsg, recvmsg
API Design /2

• We classified features as
  – static: only chosen upon socket creation
    • flow characteristic
  – configurable: chosen upon socket creation +
    adjusted later with setsockopt
    • error detection, reliability, multi-homing
  – dynamic: no need to specify in advance
    • application PDU bundling (Nagle in TCP)
    • delivery order: socket option or flags field
Implementation example: unordered reliable message delivery with SCTP

• Variant on the right: Based on draft-ietf-tsvwg-sctpsocket-23

```c
struct sctp_sndrcvinfo *si;
struct cmsghdr *cmsg;
char cbuf[sizeof (*cmsg) + sizeof (*si)];
size_t cmsglen = sizeof (*cmsg) + sizeof (*si);

cmsg = (struct cmsghdr *)cbuf;
cmsg->cmsg_level=IPPROTO_SCTP;
cmsg->cmsg_type= SCTP_SNDRCV;
si = (struct sctp_sndrcvinfo *)(cmsg + 1);
si->sinfo_stream = 1;
si->sinfo_flags = SCTP_UNORDERED;

msg.msg_control = cbuf;
msg.msg_controllen = cmsglen;

sendmsg(sockfd, &msg, 0);
```
Implementation example: unordered reliable message delivery with SCTP /2

- SCTP, version 2 (this worked)
  - `socket(PF_INET, SOCK_STREAM, IPPROTO_SCTP)`
  - set `SCTP_NODELAY` with `setsockopt`
  - followed by (10 parameters!):
    ```
    sctp_sendmsg(sockfd, textMsg, msgLength, NULL, 0, 0, SCTP_UNORDERED, 1, 0, 0);
    ```

- PI_API version
  - `pi_socket(PF_INET, 12)`
  - `pi_sendmsg(sockfd, &msg, 0);`
Tricks to get packets across

[Minion—an All-Terrain Packet Packhorse to Jump-Start Stalled Internet Transports, Janardhan Iyengar, Bryan Ford, Dishant Ailawadi, Syed Obaid Amin, Michael Nowlan, Nabin Tiwari, and Jeff Wise. 8th International Workshop on Protocols for Future, Large-Scale & Diverse Network Transports (PFLDNeT), November 2010.]

[Communication with David Ros Sanchez]
UDP vs. TCP

• Tunneling over UDP = obvious choice: UDP doesn’t do much (ports + checksum)
  – What if UDP can’t pass through?

• TCP approach #1: Minion
  – **Extend TCP implementation** with (most) SCTP functions, extending the header into the payload
  – Key function: message based processing; achieved via a separation marker
TCP approach #2: polymorphic headers

- Correct TCP on the wire, but changed semantics
  - extend new protocol’s implementation to reformat headers as packets are transmitted / received

<table>
<thead>
<tr>
<th>Bits on the wire</th>
</tr>
</thead>
<tbody>
<tr>
<td>0101010101010101 0101010101010101</td>
</tr>
<tr>
<td>1010101010101010 1010101010101010</td>
</tr>
<tr>
<td>0101010101010101 0101010101010101</td>
</tr>
<tr>
<td>1010101010101010 1010101010101010</td>
</tr>
<tr>
<td>0101010101010101 0101010101010101</td>
</tr>
<tr>
<td>1010101010101010 1010101010101010</td>
</tr>
<tr>
<td>0101010101010101 0101010101010101</td>
</tr>
<tr>
<td>1010101010101010 1010101010101010</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Middle-box interpretation: TCP</th>
<th>End system interpretation: DCCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>Sequence Number</td>
</tr>
<tr>
<td>Destination Port</td>
<td>Acknowledgement Number</td>
</tr>
<tr>
<td>Header Length</td>
<td>Reserved</td>
</tr>
<tr>
<td>Checksum</td>
<td>Options (if any)</td>
</tr>
<tr>
<td>Source Port</td>
<td>Data Offset</td>
</tr>
<tr>
<td>Destination Port</td>
<td>Checksum</td>
</tr>
<tr>
<td>Res</td>
<td>Type</td>
</tr>
<tr>
<td>Sequence Number (low bits)</td>
<td></td>
</tr>
</tbody>
</table>
Conclusion

• There’s a chance to get a more flexible Internet transport layer
  – A lot of useful, fun work to be done

• Join me! 😊
Thank you!

Questions?
Backup slides
Ensure that SCTP and DCCP are attractive


[Communication with Michael Tuexen and Morten Hustveit]
SCTP (as of February 2011)

• Essentially, SCTP = TCP++
  – what’s the point if it performs worse than TCP?
  – so that should never happen

• Two sides to this
  – Implementation issues: FreeBSD is well maintained, but Linux has many problems
  – Specification issues: One SCTP association with N streams should never perform worse than N TCP connections
    (and sometimes better)
Linux SCTP implementation

• Florian Niederbacher detected problems
  – mainly: lack of auto-buffer tuning and pluggable congestion control

• Auto-buffer tuning now available:
  http://tinyurl.com/4bhxt74 and patch submitted:
  http://tinyurl.com/45ng5d6

• Pluggable congestion control has been implemented but doesn’t work yet
More (smaller) issues

- Missing sender-dry-event (functional shortcoming, necessary for DTLS over SCTP)
  - Linux only; pointed out by Michael Tüxen

- Sending too little data in Slow-Start
  - Linux only; detected by Florian, probably wrong implementation of ABC or side-effect from burst mitigation

- Wrong calculation of header overhead
  - General problem, presented by Michael Tüxen @ PFLDNeT10
More (smaller) issues /2

• Message size > MTU needed for efficient operation in small-RTT environments
  – Linux only; detected by Florian, confirmed by Stefan Jörer, probably caused by overhead of send/receive system calls

• Flow control might be buggy
  – Pointed out by Michael Tüxen; not confirmed yet

• Making implementation more up-to-date: “improving robustness against non-congestion events”, Andreas Petlund’s thin stream work, spurious loss event detection
DCCP

<table>
<thead>
<tr>
<th></th>
<th>DCCP</th>
<th>TCP-like</th>
<th>p1</th>
<th>u</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>DCCP</td>
<td>TCP-like</td>
<td>p1</td>
<td>u</td>
</tr>
<tr>
<td>7</td>
<td>DCCP</td>
<td>Smooth</td>
<td>x</td>
<td>u</td>
</tr>
<tr>
<td>8</td>
<td>DCCP</td>
<td>Smooth</td>
<td>p1</td>
<td>u</td>
</tr>
<tr>
<td>9</td>
<td>DCCP</td>
<td>Smooth-SP</td>
<td>x</td>
<td>u</td>
</tr>
<tr>
<td>10</td>
<td>DCCP</td>
<td>Smooth-SP</td>
<td>p1</td>
<td>u</td>
</tr>
</tbody>
</table>

- TCP-like, Smooth, and Smooth-SP flow characteristic, each with unordered delivery and either partial or full error protection
  - Smooth-SP is not a “feature” (!), it’s a necessity
  - partial error correction is rather experimental

- So, “smooth” (TCP-friendly) behavior is the only real news
  - TFRC congestion control; is this enough as a selling argument?
MulTFRC

• TFRC in a nutshell
  – smooth (⇒ less jitter) yet TCP-friendly sending rate
  – receiver ACKs, sender gets measurements
  – sender constantly calculates TCP steady-state throughput equation, sends at calculated rate

• We derived an extension of this equation which yields the rate of N flows
  – plug that into TFRC, and it becomes MulTFRC
Who cares?

• Bottleneck saturation of N TCPs (w/o queues):
  100-100/(1+3N) %
  – 1: 75%. 2: 85.7% ... 6: 95% ➔ MulTFRC with N=6 nicely saturates your bottleneck (except high bw*delay link)
  – no need to stripe data across multiple connections
  – less overhead (also only one setup/teardown)
• N ∈ R⁺ - can also be 0 < N < 1
  – useful for less important traffic and multi-path (cf. MPTCP)
• can give a knob to users
Towards a Protocol-Independent Transport API

[Stefan Jörer: A Protocol-Independent Internet Transport API, MSc. thesis, University of Innsbruck, December 2010]

Two approaches

• Top-down: start with application needs (“QoS-view”)
  + flexible, good for future use of new protocols
  – loss of service granularity: some protocol features may never be used
  – been there, done that, achieved nothing...

• Bottom-up: unified view of services of all existing Internet transport protocols
  + No loss of granularity, all services preserved
  + Totally new approach, concrete design space
  – May need to be updated in the future
Our chosen design method

• **Bottom-up**: use TCP, UDP, SCTP, DCCP, UDP-Lite
  – start with lists from key references

• **Step 1**: from list of protocol features, carefully identify application-relevant services
  – features that would not be exposed in APIs of the individual protocols are protocol internals
  – e.g. ECN, selective ACK
### Result of step 1

<table>
<thead>
<tr>
<th>Transport Protocol</th>
<th>Connection Oriented</th>
<th>Flow Control</th>
<th>Congestion Control</th>
<th>App. PDU Bundling</th>
<th>Error Detection</th>
<th>Reliability</th>
<th>Delivery Type</th>
<th>Delivery Order</th>
<th>Multi Streaming</th>
<th>Multi Homing</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0/1</td>
<td>x</td>
<td>t</td>
<td>s</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UDP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>m</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UDP-Lite</td>
<td></td>
<td></td>
<td>2/3/4</td>
<td></td>
<td>x/p1</td>
<td>m</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCCP</td>
<td>x</td>
<td>x</td>
<td>2/3/4</td>
<td>x/p1</td>
<td></td>
<td>m</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCTP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0/1</td>
<td>x</td>
<td>t/p2</td>
<td>m</td>
<td>o/u</td>
<td>0/1</td>
<td>0/1</td>
</tr>
</tbody>
</table>

- x = always on, empty = never on; 0/1 = can be turned on or off
- 2/3/4 = choice between CCIDs 2, 3, 4
- P1 = partial error detection; t = total reliability, p2 = partial reliability
- s = stream, m = message; o = ordered, u = unordered
Expansion

- A line for every possible combination of features
  - 43 lines: 32 SCTP, 3 TCP/UDP

- List shows reduction possibilities (**step 2**)
  - e.g. flow control coupled with congestion control
  - duplicates, subsets

<table>
<thead>
<tr>
<th>service no.</th>
<th>transport protocol</th>
<th>connection-oriented</th>
<th>flow control</th>
<th>congestion control</th>
<th>SACK</th>
<th>PDU handling</th>
<th>stream reliability</th>
<th>delivery type</th>
<th>delivery order</th>
<th>multi-streaming</th>
<th>multi-homing</th>
</tr>
</thead>
</table>
Reduction method for step 2

• Remove services that seem unnecessary as a result of step 1 expansion
• Apply common sense to go beyond purely mechanical result of step 1
  – Question: would an application have a reason to say “no” to this service under certain circumstances?
  – Features that are just performance improvements if they are used correctly (i.e. depending on environment, not app) are not services
Step 2

• Connection orientation
  – Removing it does not affect service diversity
  – User view: API is always connection oriented
  – on the wire, non-congestion-controlled service will always use UDP or UDP-Lite
  – static distinction, clear by documentation

• Delivery type
  – easy for API to provide streams on top of message transport
  – no need to expose this as a service
Step 2, contd.

• Multi-streaming
  – Performance improvement, depending on environment conditions / congestion control behavior, not an application service

• Congestion control renamed ➔ “flow characteristic”

• Multi-homing kept although not an app. service
  – this is part of a different discussion
  – could be removed above our API
## Result of Step 2

<table>
<thead>
<tr>
<th>service no.</th>
<th>flow characteristic</th>
<th>app. PDU bundling</th>
<th>error detection</th>
<th>reliability</th>
<th>delivery order</th>
<th>multi-homing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TCP-like</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TCP-like</td>
<td></td>
<td></td>
<td>pl</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>[p2]</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>TCP-like</td>
<td></td>
<td></td>
<td>pl</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Smooth</td>
<td>x</td>
<td></td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Smooth</td>
<td>x</td>
<td></td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Smooth-SP</td>
<td></td>
<td></td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Smooth-SP</td>
<td>pl</td>
<td></td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TCP-like</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>t</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>t</td>
<td>u</td>
<td>x</td>
</tr>
<tr>
<td>14</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>p2</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>p2</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>16</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>p2</td>
<td>u</td>
<td>x</td>
</tr>
<tr>
<td>17</td>
<td>TCP-like</td>
<td>x</td>
<td>t</td>
<td>t</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>18</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>t</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>t</td>
<td>u</td>
<td>x</td>
</tr>
<tr>
<td>20</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>p2</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>p2</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>22</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>p2</td>
<td>u</td>
<td>x</td>
</tr>
<tr>
<td>23</td>
<td>TCP-like</td>
<td>x</td>
<td></td>
<td>p2</td>
<td>u</td>
<td>x</td>
</tr>
</tbody>
</table>
API Design

• Goal: make usage attractive = easy
  – stick with what programmers already know: deviate as little as possible from socket interface

• Most services chosen upon socket creation
  – int socket(int domain, int service)
  – service number identifies line number in table
  – understandable aliases: e.g. PI_TCPLIKE_NODELAY, PI_TCPLIKE, PI_NO_CC_UNRELIABLE for lines 1-3

• Sending / receiving: provide sendmsg, recvmsg; for services 1,2,11,17: send, recv
API Design /2

• We classified features as
  – static: only chosen upon socket creation
    • flow characteristic
  – configurable: chosen upon socket creation + adjusted later with setsockopt
    • error detection, reliability, multi-homing
  – dynamic: no need to specify in advance
    • application PDU bundling (Nagle in TCP)
    • delivery order: socket option or flags field
Implementation example: unordered reliable message delivery with SCTP

- Variant on the right: Based on draft-ietf-tsvwg-sctpsocket-23

```c
struct sctp_sndrcvinfo *si;
struct cmsghdr *cmsg;
char cbuf[sizeof (*cmsg) + sizeof (*si)];
size_t cmsslcn = sizeof (*cmsg) + sizeof (*si);

cmsg = (struct cmsghdr *)cbuf;
cmsg->cmsg_level=IPPROTO_SCTP;
cmsg->cmsg_type= Sctp_SNDRCV;
si = (struct sctp_sndrcvinfo *)(cmsg + 1);
si->sinfo_stream = 1;
si->sinfo_flags = SCTP_UNORDERED;

msg.msg_control = cbuf;
msg.msg_controllen = cmsslcn;

sendmsg(sockfd, &msg, 0);
```
Implementation example: unordered reliable message delivery with SCTP /2

• SCTP, version 2 (this worked)
  – `socket(PF_INET, SOCK_STREAM, IPPROTO_SCTP)`
  – set SCTP_NODELAY with `setsockopt`
  – followed by (10 parameters!):
    ```
    sctp_sendmsg(sockfd, textMsg, msgLength, NULL, 0, 0, SCTP_UNORDERED, 1, 0, 0);
    ```

• PI_API version
  – `pi_socket(PF_INET, 12);`
  – `pi_sendmsg(sockfd, &msg, 0);`