How to truly improve the Internet’s transport layer, part 2
(the technical details)
CAIA, Swinburne University, Melbourne, 10. 2. 2011

Michael Welzl
Setting the stage... a reminder

• I have already told you that
  – the Internet’s transport layer is as flexible as a rock
  – New, useful protocols like SCTP and DCCP remain unused

• A more flexible design would use abstraction
  – applications don’t specify protocol, just service (still best effort model) ➔ “transport system” could make the best possible choice out of what is available
    • Can gradually evolve without changing applications, via OS upgrades and growing protocol support by ISPs
How to get there?

• Build the system
  – benefit from protocol features without application involvement
  – API design
  – try to use a new protocol with TCP as a fall-back

• Get protocols deployed so that the system makes sense: ensure that they are attractive (SCTP implementation, DCCP service issues)

• Measure protocol availability, provide tools

You’ve seen proof-of-concept results
API Design

[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></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
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>supported by transport protocol(s)</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/SCTP</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/SCTP</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>UDP/Lite</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>4</td>
<td>UDP-Lite</td>
<td></td>
<td></td>
<td>p1</td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>5</td>
<td>DCCP/SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>[p2]</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>DCCP</td>
<td>TCP-like</td>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>7</td>
<td>DCCP</td>
<td>Smooth</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>8</td>
<td>DCCP</td>
<td>Smooth</td>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>9</td>
<td>DCCP</td>
<td>Smooth-SP</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>10</td>
<td>DCCP</td>
<td>Smooth-SP</td>
<td>p1</td>
<td></td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>11</td>
<td>SCTP</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>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>t</td>
<td></td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>13</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>t</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>14</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>15</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>16</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>17</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>18</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>t</td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>19</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>t</td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>20</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>21</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td>x</td>
</tr>
<tr>
<td>22</td>
<td>SCTP</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>SCTP</td>
<td>TCP-like</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; 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
  – removes head-of-line (HOL) blocking delay

• Local testbed, 2 Linux PCs
How is this achieved?

- Based on draft-ietf-tsvwg-sctpsocket-23

- Could not make this work in our testbed (suspect: bug in SCTP socket API)

```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);
```
How is this achieved? /2

- SCTP, version 2 (this worked)
  - `socket(PF_INET, SOCK_STREAM, IPPROTO_SCTP)`
  - `set SCTP_NODELAY with setsockopt`
  - followed by (10 parameters!):
    ```c
    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)`
Trying to use a new protocol with TCP as a fall-back


[Discussions with Bryan Ford, Jana Iyengar, Michael Tü xen, Joe Touch]
The larger problem

• 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. ....

• In fact, hosts should be able to *negotiate* the protocol combination; proposals exist
  – separate protocol describing preference graphs (HotNets Ford / Iyengar)
  – signaling over HTTP (draft-wood-tae-specifying-uri-transports-08)
  – out-of-band (e.g. DNS server) [Iyengar presentation at Hiroshima IETF]
The immediate problem

• Hosts A and B can agree on whatever they wish, but then that may just not work
  – no way to bypass trying

• Proposed negotiation methods must be used, then tested ➔ no point in doing this *initially*

• So let’s look at the “just try it” approach
Happy Eyeballs

• Originally proposed as a method to solve the IPv6 / IPv4 setup problem 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 what comes back first
  – 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
  – Likely to stay the same for a while

• 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
My proposal
A “NOT-TCP” option or bit in the TCP SYN

Case 1:
New server

Case 2:
Old server

Client

Server

Client

Server

SCTP INIT, TCP SYN: “NOT-TCP”

TCP SYN-ACK: “NOT-TCP”

SCTP INIT-ACK

SCTP COOKIE-ECHO

WAIT

TCP SYN-ACK

SCTP INIT-ACK

TCP ACK

Ignore, or use for next connection
Not-TCP concerns

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

• The scheme binds TCP+SCTP/DCCP/.. ports together forever
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)
Ensure that SCTP and DCCP are attractive


[Dragana Damjanovic, Michael Welzl: "MulTFRC: Providing Weighted Fairness for Multimedia Applications (and others too!)", CCR 39(3), July 2009.]

SCTP

- 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 is ongoing work; probably not going to be finished in this project
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 @ PFIDNeT10
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/recv 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 (simplified Eifel possible?)
DCCP

- 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 here
  - TFRC congestion control; is this enough as a selling argument?
MuLTFRNC

• 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 MuLTFRNC
Who cares?

• Bottleneck saturation of N TCPs (w/o queues): $100-100/(1+3N)$ %
  – 1: 75%. 2: 85.7% ... 6: 95% \(\Rightarrow\) 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)

• \(N \in \mathbb{R}^+\) - can also be \(0 < N < 1\)
  – useful for less important traffic and multi-path (cf. MPTCP)

• can give a knob to users

Future work?
- e.g. Mul-CUBIC-FRC?
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?