Our pre-TAPS work on transport services
Michael Welzl
Outline / disclaimer

• Overview of results documented in MSc. thesis + paper

• Not a proposal for how things should be: TAPS work should be more extensive, more up to date, make better, more informed decisions
  – But we learned some lessons back then, perhaps useful
Design method

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

• **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. TCP, SCTP: 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></td>
<td>t</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></td>
<td>t</td>
<td>m</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCCP</td>
<td>x</td>
<td>x</td>
<td>2/3/4</td>
<td></td>
<td></td>
<td>t</td>
<td>m</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCTP</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>0/1</td>
<td></td>
<td>t</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? (but not purely because of environment conditions)

  – 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 \( \Rightarrow \) “flow characteristic”

- Multi-homing kept although not an app. service
  - We felt this is a more complex discussion / decision
  - could still 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>p1</td>
<td></td>
<td>u</td>
<td></td>
<td>u</td>
</tr>
<tr>
<td>4</td>
<td>TCP-like</td>
<td>x</td>
<td>[p2]</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Smooth</td>
<td>x</td>
<td></td>
<td>u</td>
<td></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-SP</td>
<td>x</td>
<td></td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>TCP-like</td>
<td>x</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Smooth-SP</td>
<td>x</td>
<td></td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>13</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>14</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>TCP-like</td>
<td>x</td>
<td>t</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TCP-like</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>x</td>
<td>t</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>TCP-like</td>
<td>x</td>
<td>t</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>TCP-like</td>
<td>x</td>
<td>t</td>
<td>u</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>20</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>o</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>22</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>TCP-like</td>
<td>x</td>
<td>p2</td>
<td>u</td>
<td></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
  1. **static**: only chosen upon socket creation
      • flow characteristic
  2. **configurable**: chosen upon socket creation + adjusted later with `setsockopt`
      • error detection, reliability, multi-homing
  3. **dynamic**: no need to specify in advance
      • application PDU bundling (Nagle in TCP)
      • delivery order: socket option or flags field
Backup slides
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 cmmsglen = 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 = cmmsglen;

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!):
    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);
Thank you!

Questions?