Implementing Real-Time Transport Services over an Ossified Network

Stephen McQuistin and Colin Perkins
University of Glasgow

Marwan Fayed
University of Stirling
Talk Overview

- Multimedia Applications and the Transport Layer
- Ossification and Innovation
- Transport Services
- … for Real-Time Multimedia Applications
- Realising Transport Services
- Example: TCP Hollywood
Multimedia Applications

- 64% of consumer Internet traffic in 2014 → 80% by 2019 (Cisco VNI)

- Difficult to develop and standardise

- WebRTC and DASH standardisation work highlights challenges
Transport Layer

- Neither TCP or UDP provides all the features we require

- UDP adds minimal features beyond those of IP

- TCP adds many desired services (e.g., congestion control), but includes others we don’t want (e.g., reliability)

- Can build the features we need within UDP’s payload — large amount of effort, lacks reusability

- In principle, we could build a new protocol that provides the features we need
Ossification

• Middleboxes expect packets that look like either TCP or UDP: rejecting everything else is a common security policy

• New protocols (e.g., DCCP, SCTP) see little deployment on the public Internet

• TCP and UDP can be used as substrates for new protocols

• Need to ensure that middlebox compatibility is maintained
Innovation at the transport layer

• Two broad architectural approaches

• Develop a new, monolithic protocol that uses TCP or UDP as a substrate — e.g., QUIC

• Add a layer of indirection, and develop reusable building blocks — transport services
Transport Services

- “an end-to-end facility provided by the transport layer”

- Need to define the set of services required by applications

- Determine how these services can be realised by transport protocols

- Map the set of services on to an appropriate transport protocol (TCP, UDP, and others where available)

- Results in a set of reusable services that help application developers, and improve performance
Real-Time Multimedia Applications

- Maximum delay, depending on interactivity
- Interactive applications: low hundreds of milliseconds (for VoIP) — depends on human perceptibility
- Non-interactive: tens of seconds (for VoD) — depends on desired experience
- Services need to respect timeliness constraint, and add minimal latency
Timing and Deadlines

- Data has set time that it needs to have arrived by, otherwise it is skipped, and not useful
- If the transport layer doesn’t know about this deadline, useless data might be sent
- With the deadline, likelihood of data arriving on time can be estimated
- Requires network delay estimate, receive buffer size
- Fundamental service: others follow from this
Partial Reliability

• IP provides best-effort packet delivery, so some packets will be lost

• Timeliness constraint means that data is useless after its deadline

• Guaranteed reliability would result in useless data being sent, deadlines not being met

• Need *partial* reliability: retransmit lost packets, but only if they will arrive within their deadline
Message-oriented

- Partial reliability means that some packets may not be delivered
- The packets that do arrive need to be independently useful
- Implies application-level framing, with application data units (ADUs) being sent
- Given independent utility, and need to reduce latency, ADUs can be delivered in the order they arrive
- Support for multiple sub-streams
Dependencies

- Partial reliability means that not all data will arrive successfully
- Interdependencies exist within data
- Data shouldn’t be sent if it relies on a previous transmission that was not received
- Utility difficult to define for some applications

Figure 3: On-the-wire representation of application data units

Figure 4: MPEG-1 video frame prediction between I-frames (red), P-frames (orange) and B-frames (yellow)

size_t send_dgram(int fd, char *buf, size_t len, int seq);
size_t send_dgram(int fd, char *buf, size_t len, uint32_t expiryTime, uint16_t seq, uint16_t dep);
size_t send_dgram(int fd, char *buf, size_t len, uint16_t seq, uint16_t dep);
size_t recv_dgram(int fd, char *buf, size_t len);
size_t recv_dgram(int fd, char *buf, size_t len, uint8_t *substream);
void setClockrate(uint32_t clockrate);
size_t getPMTU(int sockfd);

Figure 5: uTLTCP API including dependency support

Experimental design and methodology

For the most part, the performance evaluations are carried out using the testbed topology shown in figure 6. In order to evaluate the performance of uTLTCP with respect to other protocols, the protocols used at the sender and receiver hosts are varied as shown in table 1. Where TCP is used, the TCP_NODELAY socket option is enabled, as this is not a contribution of the work presented here. The performance of a uTLTCP sender and a TCP receiver is evaluated because this configuration is more deployable than having a uTLTCP receiver; userspace libraries can be used on the TCP receiver to allow it to decode COBS-encoded datagrams, although without benefiting from the decrease in latency.

Broadly, the methodology is to send a number of packets using the listed protocols between the sender and receiver and measure performance with respect to a set of metrics, with the packet loss rates being varied between each evaluation. More specifically, 10,000 packets will be sent, with 20ms between each packet; where timelines are being tested, the clockrate is 8000Hz. The size of datagrams will alterate between 550 and 650 bytes to allow for padding to be tested. The packet loss rates being tested are 0%, 2%, 4%, 8% and 16%. Finally, evaluations will be run 10 times for each metric and protocol combination.

The size of datagrams has been selected to simulate that of audio transmission. However, in such applications, packet sizes are usually constant. They vary here only to allow padding to be tested. The choice of clock rate and packet size means that TCP’s flow and congestion control algorithms may not be exercised during these evaluations; this may affect the throughput and goodput metrics being measured.

The metrics that will be measured are:

- Average throughput: This is the amount of data delivered to the receiving host, divided by the time taken to deliver it. This includes protocol headers, padding, and duplicate packets, where appropriate.
- Average goodput: This is the amount of data delivered to the receiving application, divided by the time taken to deliver it. This excludes protocol headers, padding, and retransmissions, where appropriate. In addition, the goodput metric used here has a narrower definition than presented elsewhere [8]; packets that arrive after the time that they are to be played out will not be counted.
- Average latency: This is the average one-way latency between the sender and receiver as measured at the receiver.
- Average interarrival jitter: This is the average delay variation between consecutive packets.
Connections & Congestion Control

- Congestion control important to protect the network and other applications
- Need to select algorithm appropriate to application
- Connection-oriented service is useful in some scenarios
- Enables explicit setup and teardown of in-network state (e.g., for NAT mappings)
# Real-Time Transport Services

<table>
<thead>
<tr>
<th>Transport Service</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlines</td>
<td>Core</td>
</tr>
<tr>
<td>Partial reliability</td>
<td>Core</td>
</tr>
<tr>
<td>Dependencies</td>
<td>Core</td>
</tr>
<tr>
<td>Message-oriented</td>
<td>Core</td>
</tr>
<tr>
<td>Sub-streams</td>
<td>Core</td>
</tr>
<tr>
<td>Congestion controlled</td>
<td>Core</td>
</tr>
<tr>
<td>Connection oriented</td>
<td>Subsidiary</td>
</tr>
<tr>
<td>Keep-alive</td>
<td>Subsidiary</td>
</tr>
</tbody>
</table>
Abstract API Lifecycle

Socket creation and connection primitives inherited from Berkeley API

Server
- socket()
- bind()
- listen()
- accept()
- close()

Client
- socket()
- connect()
- close()
Abstract API Lifecycle

Server

- socket()
- bind()
- listen()
- accept()
- close()

Client

- socket()
- connect()

Sets play-out delay, in milliseconds, and sends to server

- set_po_delay()
Abstract API Lifecycle

Server
- socket()
- bind()
- listen()
- accept()
- close()

Client
- connect()
- send_message()
- recv_message()
- close()

(send_message) Sends message; requires sequence number, sub-stream, deadline, and dependency information

(set_po_delay)
Abstract API Lifecycle

Server
socket()
bind()
listen()
accept()
close()

Client
socket()
connect()
set_po_delay()
recv_message()

Retrieves next message in arrival order, with its sub-stream identifier

send_message()
close()

recv_message()
close()
Realising transport services

• Need to support this combination of transport services

• Ossification restricts us to using either TCP or UDP — might change over time

• UDP first → fallback to TCP

• UDP not always available (5-10% - Google, 1-5% MAMI)

<table>
<thead>
<tr>
<th>Transport Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlines</td>
</tr>
<tr>
<td>Partial reliability</td>
</tr>
<tr>
<td>Dependencies</td>
</tr>
<tr>
<td>Message-oriented</td>
</tr>
<tr>
<td>Sub-streams</td>
</tr>
<tr>
<td>Congestion controlled</td>
</tr>
<tr>
<td>Connection oriented</td>
</tr>
<tr>
<td>Keep-alive</td>
</tr>
</tbody>
</table>
UDP as a substrate

- Already supports the sending of datagrams/messages
- Support for partial reliability requires detecting loss, retransmitting if message will arrive before deadline
- Need an estimate of one-way network delay
- Sub-stream support requires small header in each message
- Connections and congestion control can be added
TCP as a substrate

- Messaging requires a framing mechanism, to support resegmenting middleboxes — e.g., COBS, as in Minion/uTCP
- Sub-stream support requires small header in each message
- Already supports connections
- Congestion control supported, but algorithm fixed: support for other algorithms as in DCCP
Relaxing reliability in TCP

- Middleboxes ossified around TCP do not expect gaps in the TCP sequence space

- Need to “retransmit” missing TCP sequence numbers, without retransmitting payloads — inconsistent retransmissions

- Mapping between data and TCP sequence number is no longer constant
TCP Hollywood

- Unordered, partially reliable message-oriented delivery
- Intermediary layer: COBS encoding to maintain message boundaries
- Kernel: unordered delivery of incoming segments
TCP Hollywood

- Uses inconsistent retransmissions to support partial reliability
- Evaluation between TCP Hollywood server and 14 clients around the UK
- Evaluations also conducted by Honda et al.
- Small scale — more evaluations needed

<table>
<thead>
<tr>
<th>ISP</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrews &amp; Arnold</td>
<td>80</td>
</tr>
<tr>
<td>BT</td>
<td>80</td>
</tr>
<tr>
<td>Demon</td>
<td>80</td>
</tr>
<tr>
<td>EE</td>
<td>80</td>
</tr>
<tr>
<td>Eclipse</td>
<td>80</td>
</tr>
<tr>
<td>Sky</td>
<td>80</td>
</tr>
<tr>
<td>TalkTalk</td>
<td>80</td>
</tr>
<tr>
<td>Virgin Media</td>
<td>80</td>
</tr>
<tr>
<td>EE</td>
<td>80</td>
</tr>
<tr>
<td>O2</td>
<td>80</td>
</tr>
<tr>
<td>Three</td>
<td>80</td>
</tr>
<tr>
<td>Vodafone</td>
<td>80</td>
</tr>
</tbody>
</table>
Summary

• Services can be implemented on TCP and UDP

• TAPS WG formulating list of services by breaking down existing protocols

• Here, top down: start with application, define services without constraints of existing protocols

<table>
<thead>
<tr>
<th>Transport Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadlines</td>
</tr>
<tr>
<td>Partial reliability</td>
</tr>
<tr>
<td>Dependencies</td>
</tr>
<tr>
<td>Message-oriented</td>
</tr>
<tr>
<td>Sub-streams</td>
</tr>
<tr>
<td>Congestion controlled</td>
</tr>
<tr>
<td>Connection oriented</td>
</tr>
<tr>
<td>Keep-alive</td>
</tr>
</tbody>
</table>