Implementing Real-Time Transport Services over an Ossified Network

Stephen McQuistin
University of Glasgow, UK
sm@smcquistin.uk

Colin Perkins
University of Glasgow, UK
csp@csperkins.org

Marwan Fayed
University of Stirling, UK
mmf@cs.stir.ac.uk

ABSTRACT

Real-time applications require a set of transport services not currently provided by widely-deployed transport protocols. Ossification prevents the deployment of novel protocols, restricting solutions to protocols using either TCP or UDP as a substrate. We describe the transport services required by real-time applications. We show that, in the short-term (i.e., while UDP is blocked at current levels), TCP offers a feasible substrate for providing these services. Over the longer term, protocols using UDP may reduce the number of networks blocking UDP, enabling a shift towards its use as a demultiplexing layer for novel transport protocols.

CCS Concepts

•**Networks** → **Protocol design; Transport protocols;**

Keywords

Transport protocols; real-time multimedia applications

1. INTRODUCTION

Real-time applications are increasingly present in the Internet. We want to make it easier to write these applications, while also improving the quality of experience for users by lowering latency and increasing the quality and robustness of the media delivery. Unfortunately, the limitations of the standard Internet transport protocols make this a challenging target, and the ossified nature of the network makes it increasingly difficult to deploy new transport protocols.

There have been several attempts to standardise and deploy new transport protocols [13, 24]. In practice, however, only UDP and TCP are widely usable in the Internet, since the remaining protocols are blocked by firewalls and other middleboxes. UDP exposes the best-effort IP packet delivery service, offering the flexibility to develop new protocols, but at the cost of requiring new mechanisms to be defined and implemented from scratch. In contrast, TCP mechanisms are well defined, consisting of sophisticated congestion control coupled with a reliable, ordered, byte stream API. These have been proven suitable for many applications, but are inappropriate for real-time traffic. While both protocols are used for real-time applications, neither really provides the right services and API. This forces each application to re-invent or re-interpret mechanisms that should be provided by the transport. The increased costs and complexity of doing so make applications less reliable, and raise barriers to innovation.

In this paper we identify and present the appropriate set of transport services and APIs for real-time applications, and demonstrate their merit by implementing a proof-of-concept. We show that it is possible to realise real-time services and APIs in the context of both TCP and UDP, despite the limitations imposed by their legacies, by middleboxes, and by the ossification of the network. Initial experiments with our implementation suggest that the network has the flexibility to deploy new transport protocols, provided care is taken to reinterpret application and transport layer boundaries in a manner that is not at odds with conventional UDP and TCP layer boundaries.

In doing so we make three main contributions. First, we make explicit the needs of real-time applications, as well as the appropriate transport services and APIs to support those needs. Second, we illustrate an example realisation of those transport services on the current Internet, in the context of UDP and TCP deployments. Finally, we present initial measurement results that suggest the proposed mechanisms ought to be usable in the public Internet.

We structure the remainder of this paper as follows. We begin in Section 2 by discussing transport services for real-time applications, and outlining the common conceptual API that those applications use. This is followed in Section 3 by a review of deployment considerations for new protocols, caused by ossification of the network. Section 4 considers, in particular, how TCP reliability semantics can evolve within the constraints of the existing infrastructure. The semantics are realised and put into practice in Section 5. Finally, Section 6 discusses related work, and Section 7 concludes.

2. REAL-TIME TRANSPORT SERVICES

In the IETF, the Transport Services (TAPS) working group is chartered to (1) develop a taxonomy of *transport services*, that is, to identify the features that comprise, and can be combined to form, complete transport protocols; and (2) to develop an abstract API for applications to request desirable services, allowing the system to select an appropriate transport protocol based on application needs. It is hoped that this will loosen the coupling between application and transport, so enabling deployment of new transport protocols.

2.1 Desirable Transport Services

The work in TAPS provides a vocabulary for discussing the components of transport protocols. The vocabulary is useful when discussing the needs of real-time applications, and the protocols to

ANRW '16, July 16 2016, Berlin, Germany

ISBN 978-1-4503-4443-2/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2959424.2959443

support them. In this section, we use this to describe the transport services we believe are required for real-time multimedia applications. Table 1 summarises the transport services discussed.

Timing and Deadlines: Timing is the most salient feature of real-time applications. Since their data must be conveyed with real-time demands, they all have some concept of a *deadline*. Data that fails to present within the deadline is otherwise useless. The 'slack' in a deadline depends on the application. Interactive applications, such as telephony, video conferencing, or telepresence, require low end-to-end latency. Their deadlines for presenting the media, i.e., playing the audio and displaying the video frame, range from tens to a few hundred milliseconds. Non-interactive application deadlines associated with broadcast and on-demand programming are on the order of seconds.

Networked multimedia deadlines are unusual when compared to other real-time systems. They are simultaneously flexible and strict: flexible in that the exact value of the deadline is typically not important, provided it is of the right order-of-magnitude for the application, but strict in that any particular deadline provides a cut-off, after which the data arrives too late to be rendered to the user (although, again, it is not entirely useless, since it might be used to complete a predictive coding chain, improving the quality of frames decoded later).

Partial Reliability: In a best-effort network, deadlines constrain packet delivery service to *partial reliability*. For example, when used to repair loss, the limits of forward error correction imply some probability that packet will be non-recoverable. By contrast, retransmissions used to recover from loss have potentially unbounded delay (since any retransmission may itself be lost). Accordingly, a transport protocol that meets deadlines should provide partial reliability, acknowledging that it may be unable to deliver all data by its deadline.

Many real-time applications run over TCP today, though TCP offers no partial reliability service. TCP's full reliability can lead to play-out stalls when the application is blocked by retransmissions that take too long. These stalls are one of the primary causes of poor user experience in streaming applications. For the applications under scrutiny, a missed frame that is not delivered by its deadline, while surrounding frames are delivered, is much less disruptive than a stall in play-out waiting for repair.

Message-oriented Dependencies: The combination of deadlines and partial reliability makes *dependency management* an important transport service. In particular, data should never be sent when it relies on a previous transmission that was never received. Providing this service is complicated by the two ways in which data can be *useful* to applications: it may itself be played out, or it may be needed as part of the application's decoding chain. Interdependencies between frames of video exist within a number of codecs. The original MPEG-1 codec [14] divided video frames into three types. I-frames were independently encoded, while P- and B-frames contain only the changes since the previous frame (P), or between frames (B), and so could only be decoded dependent on the successful arrival of other frames. Newer codecs, such as H.264 [25], use more complex and sophisticated versions of the same idea. A consequence is that the sender might know that a frame will not arrive in time to be played out, but may need to send it anyway to ensure that the receiver can decode any dependent frames sent later in the stream.

In the context of both deadlines and dependencies coupled with packet loss, partial reliability requires application-level framing [5] to make the best use of payload data. At the transport layer, this implies a *message oriented* service, that maintains application data unit (ADU) boundaries. Messages are delivered to the application in the order they arrive. As seen in TCP, in-order delivery can introduce significant latency: incoming segments may be head-of-line blocked waiting for the delivery of an earlier segment.

Transport Service	Requirement
Deadlines	Core
Partial reliability	Core
Dependencies	Core
Message-oriented	Core
Sub-streams	Core
Congestion controlled	Core
Connection oriented	Subsidiary
Keep-alive	Subsidiary

Table 1: Transport services for real-time multimedia

Message orientation may also be used to construct a *sub-stream* service. Many multimedia applications make use of multiple data streams. For example, a simple IPTV application will maintain separate audio and video stream. These could be sent across multiple transport-layer connections, but overheads can be reduced by multiplexing these flows on a single connection.

Connections and Congestion Control: We note the importance of congestion control. Historically, real-time many applications have required an isochronous channel, and have not implemented congestion control. This is impractical on the Internet. Further, while some applications are non-adaptive or constant bit-rate, an increasing number are either, or both, of adaptive and variable bit-rate. Users would be better served by applications that adapt to available bandwidth. This is especially true of mobile applications, where channel capacity can vary significantly over time.

We note that a connection-oriented transport is a lesser requirement for many real-time multimedia applications. Indeed, flexibility to change the destination within a call is beneficial for applications that support mobile users, and for some forms of multiparty session. On the other hand, maintaining per-connection state at the endpoints is helpful for the implementation of many forms of congestion control. Signalling messages indicating start and end of connections can also ease NAT traversal, and help dynamically manage firewall pinholes, by indicating when in-network state should be created and can be torn down. Accordingly, it is often desirable for the transport to be connection oriented.

We believe these concerns outweigh the benefits of connectionless transport, and so add a requirement for connection oriented service. Similarly, while not strictly needed by the applications, it is beneficial if the transport provides a keep-alive service to refresh NAT and firewall bindings if the application goes silent.

2.2 Abstract API

Given the set of transport services outlined in Table 1, we sketch an abstract API in Table 2. The primitives divide into five categories:

- Hosts setup and tear-down sockets using the `socket()` and `close()` functions, as in the standard Berkeley sockets API.
- Socket options can be set and read using the `setsockopt()` and `getsockopt()` functions respectively, again, mirroring the standard Berkeley sockets API. A socket option may be used to select the desired congestion control algorithm (e.g., as with the `DCCP_SOCKOPT_CCID` socket option in DCCP [13]).
- The connection primitives are the same as those of TCP sockets. Servers `bind()` to a particular address and port, then `listen()` for and `accept()` incoming connections. Clients `connect()` to a server.

Transport Service	Function	Parameters	Return Value(s)
	socket	af – Address family st – Socket type	Socket descriptor
	close	sd – Socket descriptor	0 (success), -1 (error)
	getsockopt/setsockopt	sd – Socket descriptor level – Protocol level option – Option name value – Option value len – Option length	0 (success), -1 (error)
Connection oriented	bind	sd – Socket descriptor addr – Address to bind to addrlen – Length of addr	0 (success), -1 (error)
	listen	sd – Socket descriptor	0 (success), -1 (error)
	accept	sd – Listening socket descriptor addr – Address of peer addrlen – Length of addr	Connection socket descriptor
	connect	addr – Address to connect to addrlen – Length of addr	0 (success), -1 (error)
Deadlines	set_po_delay	delay – Playout delay (in ms)	0 (success), -1 (error)
Message oriented	send_message	sd – Socket descriptor buf – Message data len – Length of message data seq_num – Sequence number	Number of bytes sent
Deadlines		deadline – Relative deadline of message (in ms)	
Dependencies		depends_on – seq_num of dependency	
Sub-streams		substream – Substream identifier	
	recv_message	sd – Socket descriptor buf – Buffer for message data len – Size of buf	Number of bytes received Substream identifier

Table 2: Outline transport API for real-time applications. Return values shown are for successful calls; in all cases, -1 is returned in the event of an error

- Once the connection is established, the receiver then indicates its media play-out delay, in milliseconds, via the `set_po_delay()` call. This specifies the time that the application will buffer data, to compensate for network timing jitter, before it is rendered to the user. The play-out delay is fed back to the sender, for use as part of the media deadline estimation.
- Finally, message-oriented data transmission is exposed by the `send_message()` and `recv_message()` functions. These expose a partially reliable message delivery service to the application, framing data such that either a complete message is delivered, or it is lost in its entirety.

It is instructive to compare the partially reliable send and receive functions to their Berkeley Sockets API counterparts. The `send_message()` call takes four additional parameters. These are 1) a message sequence number, that can be used to re-order messages and detect message loss; 2) a relative deadline, which is combined with an estimate of the current round-trip-time, and the time that the message has spent in the sending buffer, to determine if a message will arrive in time to be played-out; 3) the message sequence number of any message on which this depends, for example, of a video I-frame on which a P-frame is predicted; and 4) a sub-stream identifier, used, for example, to differentiate audio, video, sub-title, control, and repair streams. Of this metadata, only the sub-stream identifier is sent on the wire. The sequence number, deadline, and dependency information is used only by the sender to provide the partially reliable service.

The `recv_message()` call returns the sub-stream identifier and length of the message, along with the received message data. This allows the receiver to direct the message to the correct decoding queue.

A message that won't arrive within its lifetime is considered to have *expired*. A message is also considered to have expired if its message sequence number dependency, `depends_on`, has expired. A partial reliability service follows from this deadline and dependency service: messages will be reliably transmitted until they expire.

It is to be noted that this API is not dissimilar to the PR-SCTP abstract API, which provides *timed reliability*, using a "lifetime" specified by the application.

3. INNOVATION AND OSSIFICATION

The Internet architecture, in principle, allows free innovation at the transport layer, provided the underlying network (IP) layer is unchanged. Routers should inspect the source addresses of packets to perform network ingress filtering [6], and the destination addresses to route packets to the correction destination, but should not inspect their contents. This is not, of course, how the real network operates.

There are performance and security benefits that can be attained by adding transport-layer functionality *within* the network. For example, a firewall can better protect the network if it can detect payload anomalies.

The implication of this reality is that it is difficult to deploy new transport protocols. The installed base of NATs, firewalls, and other middleboxes is such that packets that do not look like

TCP or UDP are unlikely to pass the network. We may innovate all we like, provided the transport of the future looks like TCP or UDP to middleboxes. This is inconvenient, certainly, but is not necessarily a bad thing. The Internet is critical infrastructure. It support emergency services, healthcare applications, infrastructure components, financial services, and so on, many of which are essential to the functioning of society. Making changes to this type of infrastructure *should* require careful consideration of backwards compatibility [16].

UDP is the obvious base for future protocol development, since it provides minimal additional services over the IP layer, allowing great flexibility in innovation for protocols tunnelled on top. Provided middleboxes do not inspect the payload too carefully, the only real cost to innovation, when compared to a native transport protocol running over IP, is a few bytes of additional header. Examples in this space include RTP [23], one of the most widely deployed real-time transport protocols; the WebRTC Data Channel [11], which tunnels peer-to-peer SCTP associations over a DTLS association over UDP; and QUIC [7], which provides a modern alternative to TCP, implemented over UDP.

Despite these advantages, UDP can be problematic as a substrate for new protocol development. UDP traffic is blocked by some enterprise firewalls, and some in the operations community have a strong distrust of UDP-based protocols and applications [2]. In part this is due to ignorance. Outside specific niches, such as DNS, UDP has not been widely used in enterprise environments, and hence is widely misunderstood. Blocking the unknown is a rational response. In addition, UDP traffic has been widely used as a component of distributed denial of service (DDoS) attacks, leading some to install blanket blocks of UDP as a safety measure (blanket blocking, rather than the more targeted blocks used when TCP traffic is used in DDoS attacks, are justified using the argument that UDP is not widely used). These issues are slowly changing, as UDP-based applications penetrate the enterprise consciousness, but UDP is not universally available (Google report 90-95% of endpoints are reachable with QUIC running over UDP [22], but it is not clear that the set of hosts running their Chrome browser is representative of all Internet environments).

Beyond the availability of UDP, it is often necessary to use TCP because HTTP is being used at the application-layer. For real-time systems, this is likely to be an HTTP adaptive streaming (HAS) protocol, such as MPEG-DASH or Apple's HLS. Using TCP as a substrate enables the use of these protocols, allowing applications to benefit from the existing infrastructure that supports them.

TCP is a more complex choice for innovation. It is a more sophisticated protocol than UDP, with complex headers, and a protocol state machine that mandates much more behaviour and is widely understood, and policed, by in-network middleboxes. This does not mean that TCP cannot evolve, or form the basis for new transport services. Rather, it means that any innovation or development must be done carefully, paying very careful attention to backwards compatibility.

We identify a number of places where TCP can evolve with comparative freedom. These include congestion control, the end-point API, and data segmentation. If care is taken, there is also the possibility to change the reliability semantic.

The TCP congestion control algorithm is executed by the end points, and can be changed, provided the new version requires no new information to be exchanged. We note that, while standardised TCP congestion control has followed the goal of maximising throughput at the expense of latency and variability, this is not required by the protocol. TCP Vegas [1] is perhaps the best known approach that changes these constraints, with a delay-based algorithm that reduces latency, although it is known to be less aggressive than standard TCP, and is prone to starvation. FAST TCP [12] is a more modern delay-based algorithm that competes well with standard TCP in many environments, and is seeing commercial deployment. The development of TCP congestion control shows that there might be fairness issues as new algorithms are deployed, but the network does not prevent the deployment of those algorithms.

It would also be possible to implement alternative congestion control algorithms that seek stability, or compatibility with the dictates of a video codec, rather than traditional "TCP Friendly" congestion control, even if implemented within TCP. To do this effectively might require changes to the interface between the application and the TCP stack, even if the on-the-wire format remains the same. For example, video applications generate data periodically, and it might help the congestion control to know the period, so it can pace out data; video traffic is less elastic than many TCP bulk flows, and it might be benefical to inform the stack of an upper rate beyond which there is no point increasing the congestion window, and a lower rate beyond which the flow cannot proceed; and informing the codec of the RTT and congestion window might allow it to better schedule bursts of traffic to match the available capacity. In the interactive video conferencing community, [26] addresses this issue for congestion control over RTP on UDP/IP, but there is no analagous document for TCP congestion control interactions as yet.

The API that is exposed to applications using TCP is invisible to the network, and can be changed. For example, TCP Fast Open [3] has been implemented by overloading the connectionless `sendto()` call to trigger an implicit `connect()` when used on an unconnected TCP socket. Relaxing the API to enable out-of-order delivery of segments is trivial: segments are delivered to the application in the order that they arrive, with their TCP sequence attached. The TCP sequence number can be passed to the application using the existing Berkeley sockets API, either with the received data, or using `getsockopt()`. Out-of-order delivery is not useful when using a byte-stream abstraction, and so the API should be further modified to provide a message-oriented abstraction. The Berkeley sockets API already supports such an abstraction for datagram protocols.

These changes could address many of the transport service needs for real-time applications, but still leave a critical issue of how to improve timing behaviour. Specifically, how to enable partial reliability for TCP, after which it is possible to layer-on support for managing deadlines and dependencies.

4. PARTIAL RELIABILITY AND TCP

Partial reliability (i.e., reliability conditional on timing and dependency information) can be implemented by relaxing TCP's reliability guarantee. The implication of this is that we need to offer a message-oriented abstraction to applications. If the arrival of a segment cannot be guaranteed, then it is not possible to offer a byte stream abstraction.

To offer a message-oriented abstraction, the boundaries between each message must be maintained between sender and receiver. This means that a framing mechanism is required: it is not sufficient to send each message in a single segment, as this mapping will not necessarily be maintained by the network. A framing marker is added to the start and end of each message before transmission, and removed on reception, and an encoding algorithm is used to escape all occurrences of the framing marker within the message data. This process does not impact on the data that can be sent or received by applications. As discussed in Section 5, COBS framing [4] is suitable for this purpose.

Middleboxes in the network have ossified around TCP's reliability mechanism: they do not expect gaps in the TCP sequence number

space. Honda *et al.* [9] test the behaviour of middleboxes in response to gaps in the TCP sequence number space, showing that middleboxes interfere with flows in up to 29% of tested paths. This depends on the mechanism and port number used, with ports used by common applications impacted most. To ensure compatibility with these middleboxes, offering partial reliability requires using *inconsistent retransmissions*. Retransmissions will be triggered as under standard TCP to ensure that the sequence number space is filled, but the data in a retransmitted segment may not be the same as the original. This means that the mapping between message data and TCP sequence numbers is no longer static: a given TCP sequence number may be relate to different messages at different times. Therefore, an application-level sequence number is required to allow messages to be uniquely identified (multipath TCP has a similar requirement).

When a TCP segment is to be retransmitted, the mapping between its sequence number and application-level sequence numbers is used to determine which messages within the segment are to be retransmitted. A liveness check is performed on these messages, to determine that (i) the message will arrive on time to be played out; and (ii) the message does not depend on an expired message. For (i), we combine the time that the message has spent in a sending queue, with an estimate of the round-trip time and the current play-out delay. This is then compared against the lifetime of the message, as expressed by the application. For (ii), we maintain metadata about sequence numbers that have expired, and check this metadata for the dependency expressed by the application.

This mechanism – inconsistent retransmissions – is visible to middleboxes on the network that are performing payload inspection. These middleboxes may interpret this behaviour as relating to an attack. For example, a man-on-the-side attack exhibits similar behaviour, where a malicious host is injecting data into an existing TCP flow. As a result, our connection may be disrupted. Honda *et al*. [9] conducted experiments across 135 paths on the Internet, to determine support for inconsistent retransmissions. They observed that the majority of paths delivered inconsistent retransmissions successfully. On Port 80 (HTTP), the original segment was delivered on 7% of paths tested. Only one connection reset was observed.

We conducted further deployment experiments using inconsistent retransmissions, testing all major UK providers, with the sender at the University of Glasgow [15]. The results are shown in Table 3. We found that 100% of tested fixed-line networks delivered inconsistent retransmissions successfully. However, delivery of the original segment is common on cellular networks, with only 25% of tested networks delivering inconsistent retransmissions successfully and reliably. The behaviour observed when evaluating cellular networks was consistent with that of a transparent, split-connection TCP cache. Segments were lost, but were retransmitted (with the IP address of the sender) by a middlebox in the network. It is likely that these caches are deployed close to the wireless link, given its relatively high rate of non-congestive loss.

These deployment experiments suggest that our protocol should be flexible: inconsistent retransmissions might not be delivered, and we should handle reception of the original segment. If the protocol detects that inconsistent retransmissions are not being delivered, they can be disabled for the connection. Further, if a connection reset occurs, then the connection should be retried with the mechanism disabled.

Use of inconsistent retransmissions can interact negatively with middleboxes that cache and re-segment TCP streams, resulting in the corruption of messages between sender and receiver. The result can be a message formed from some combination of the original message and an inconsistent retransmission. To protect against this, a checksum must be attached to each message, to allow the receiver to verify its integrity. The role of a checksum may also be fulfilled by using a secure transport, such as DTLS [20].

	ISP	Port 4001	Port 80
Fixed-line	Andrews & Arnold	●	●
	BT	●	●
	Demon	●	●
	EE	●	●
	Eclipse	●	●
	Sky	●	●
	TalkTalk	●	●
	Virgin Media	●	●
Cellular	EE	▲	▲
	O2	▲	▲
	Three	●	●
	Vodafone	●	▲

Table 3: Deployability of inconsistent retransmissions, where ● indicates successful delivery, ▲ indicates delivery of the original data, and ■ indicates connection failure (none observed). We note that the campus firewall near the server blocks UDP traffic, so all are examples where fallback from UDP to TCP is beneficial for real-time traffic.

5. REALISING TRANSPORT SERVICES

While further measurement studies are required to confirm the ability to deploy wire-visible changes to TCP (such as inconsistent retransmissions) in the wider Internet, we have shown that we can provide all of the transport services needed by real-time applications, using either TCP or UDP.

Evidence that these services can be deployed above UDP exists in the form of the WebRTC data channel [11] and QUIC protocol [7]. The former is a peer-to-peer protocol, comprising an SCTP association running over DTLS, itself running over a UDP flow negotiated via an SDP [8] offer/answer exchange [21] as part of a WebRTC session [10] (WebRTC media uses RTP over UDP also, further showing the utility of UDP-based data). This has been deployed in popular web browsers, with global deployment, and demonstrated to be effective. The latter is implemented by Google in their Chrome browser, and used as an alternative to TCP has a significant fraction of web traffic downloads from their domain.

Deployments using UDP are popular, and work well. However, as described in Section 3, there are also reasons for providing these services over TCP, since there are a significant fraction of networks that block UDP traffic. It is clearly possible to run real-time traffic over TCP, as demonstrated by applications such as NetFlix or the BBC iPlayer that comprise the majority of Internet traffic. However, TCP has a inconvenient API that imposes lots of work on application developers, and introduces higher than desired latency. We have shown how to address these issues, and provide the full set of transport services we propose in Section 2 in previous work, with our TCP Hollywood proposal [15].

The architecture of TCP Hollywood is shown in Figure 1. TCP Hollywood implements all of the services described in Section 2, splitting functionality across an intermediary layer in user-space, and a set of modifications to the kernel. This split allows applications to program against one API, whether or not the kernel modifications are available: the intermediary layer functions in both cases.

At the sender, applications pass messages (using an API similar to that given in Table 2) to the intermediary layer, with their metadata, including deadline and dependency information. At the intermediary

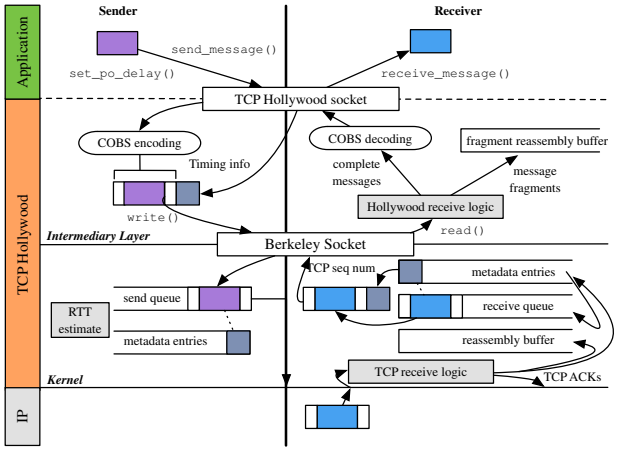

Figure 1: TCP Hollywood architecture

layer, COBS encoding [4] is used to escape all zero bytes in the message data, allowing them to be used as framing markers. The message's metadata is then attached to the encoded and framed message, before being passed to the kernel using the standard Berkeley sockets API.

At the kernel, the message data is queued in TCP's sending buffer, while the metadata is held in a separate structure. Nagle's algorithm, designed to coalesce smaller writes into larger segments, is disabled to minimise latency. As segments are (re-)transmitted, their deadlines and dependencies are checked to ensure that the message will be useful on arrival. In the current version of TCP Hollywood, the dependency check does not overrule the deadline check: only data that can be played out will be sent. If the message does not pass the liveness check, the next message in the queue that is live will be sent instead. If this is a retransmission, then inconsistent retransmissions will be used: the replacement message will be sent with the same TCP sequence number as the original.

At the receiver, segments are passed to the kernel, where they are initially processed as under standard TCP: duplicate acknowledgements are generated for out-of-order segments, for example. After this, a metadata entry is created, and placed in FIFO queue. When the intermediary layer reads from the socket, it receives the segment associated with the metadata entry at the head of the queue, with its TCP sequence number attached. This means that segments are delivered in the order that they arrive, removing the latency associated with head-of-line blocking in TCP [15].

At the intermediary layer on the receiver, incoming segments are scanned for complete messages (i.e., data between two zero bytes), which are decoded and passed to the application. While segments are sent containing only one message, these may be resegmented or coalesced in the network. A segment may arrive containing *fragments* of message data. These fragments are buffered, alongside their TCP sequence number, awaiting the arrival of the remainder of the message. Once the message has been reassembled, it is decoded, and delivered to the application.

Taken together, the wide experiences with the WebRTC Data Channel and QUIC demonstrate that the transport services necessary to support real-time traffic could be deployed running over UDP. Our work prototyping the TCP Hollywood protocol, and earlier measurements by Honda *et al*. [9] also suggest that deployment over TCP is possible.

6. RELATED WORK

Related changes to TCP are made by Minion protocol [18], that uses TCP as a substrate to provide an unordered, message-oriented service to applications, enabling some of the transport services described in Section 2, but without support for partial reliability, deadlines, and dependencies. Time-Lined TCP (TLTCP) [17] similarly provides a message-oriented service, but allows applications to attach a time-line to messages. Messages are (re-)transmitted as under standard TCP within their time-line, after which they are discarded. The mechanism by which this service is provided (introducing gaps in the sequence space) hinders deployment.

QUIC [7] demonstrates that similar services can be provided by a new protocol running over UDP, while [19] and [11] demonstrate that existing protocols, DCCP and SCTP, can also be effectively tunnelled over UDP. Fallback to TCP is discussed in this paper, and on our previous work [15].

7. CONCLUSIONS

The standard transport protocols, TCP and UDP, are not well-suited for real-time applications. Both can be made to work, but the existence of numerous papers exploring how to make media play-out over TCP reliable, and almost as extensive a collection discussing UDP-based protocol design, suggests that this is difficult to do well. To make effective use of the network, and simplify real-time application design and implementation, we need to deploy new transport services and protocols that allow innovative applications to be developed by users who are not experts in transport protocol design. We discussed requirements for such a new transport, in the context of the TAPS framework, and outlined a straw-man abstract API, in Section 2.

It seems likely that the right long-term approach for doing this is to repurpose UDP as a demultiplexing layer for higher-layer protocols. We can then deploy an appropriate transport protocol framework as a user-space library, that can be reused as appropriate. In the short-term, however, there are sufficient networks that block UDP, that any new transport protocol needs to be able to run over TCP. Sections 3 and 4 discuss how this can be done, and suggest from some initial measurement studies that this may be feasible to deploy. Section 5 considers prototypes that present such services over UDP, and presents our initial prototype demonstrated for TCP-based use.

The challenge for the future is in combining such techniques below a common API, so that an application can transparently switch between UDP-based and TCP-based transport, depending on what is supported by the underlying network. This is the promise of the TAPS API, that we have shown ought to be feasible for real-time applications.

8. REFERENCES

[1] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the SIGCOMM Conference*, pages 24–35, London, UK, August 1994. ACM.

[2] C. Byrne and J. Kleberg. Advisory Guidelines for UDP Deployment. Internet Engineering Task Force, July 2015. Work in Progress.

[3] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP Fast Open. Internet Engineering Task Force, December 2014. RFC 7413.

[4] S. Cheshire and M. Baker. Consistent Overhead Byte Stuffing. In *Proceedings of the SIGCOMM Conference*, Cannes, France, September 1997. ACM.

[5] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the SIGCOMM Conference*, Philadelphia, PA, September 1990. ACM.

[6] P. Ferguson and D. Senie. Network Ingress Filtering. Internet Engineering Task Force, May 2000. RFC 2827.

[7] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. Internet Engineering Task Force, January 2016. Work in Progress.

[8] M. Handley, V. Jacobson, and C. S. Perkins. SDP: Session Description Protocol. Internet Engineering Task Force, July 2006. RFC 4566.

[9] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the Internet Measurement Conference*, Berlin, Germany, November 2011. ACM.

[10] C. Jennings, T. Hardie, and M. Westerlund. Real time communications for the web. *IEEE Communications Magazine*, 51(4), April 2013.

[11] R. Jesup, S. Loreto, and M. Tuezen. WebRTC Data Channels. Internet Engineering Task Force, January 2015. Work in Progress.

[12] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: Motivation, architecture, algorithms, performance. In *Proceedings of the Infocom Conference*, Hong Kong, China, March 2004. IEEE.

[13] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340, March 2006.

[14] D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, 1991.

[15] S. McQuistin, C. Perkins, and M. Fayed. TCP Hollywood: An Unordered, Time-Lined, TCP for Networked Multimedia Applications. In *Proceedings of the Networking Conference*, Vienna, Austria, May 2016. IFIP.

[16] S. McQuistin and C. S. Perkins. Reinterpreting the Transport Protocol Stack to Embrace Ossification. In *Proceedings of the IAB Workshop on Stack Evolution in a Middlebox Internet*, Zürich, Switzerland, January 2015.

[17] B. Mukherjee and T. Brecht. Time-lined TCP for the TCP-friendly delivery of streaming media. In *Proceedings of the International Conference on Network Protocols*, Osaka, Japan, November 2000. IEEE.

[18] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, San Jose, CA, April 2012. USENIX.

[19] T. Phelan, G. Fairhurst, and C. S. Perkins. DCCP-UDP: A datagram congestion control protocol UDP encapsulation for NAT traversal. Internet Engineering Task Force, November 2012. RFC 6773.

[20] E. Rescorla and N. Modadugu. Datagram Transport Layer Security version 1.2. Internet Engineering Task Force, January 2012. RFC 6347.

[21] J. Rosenberg and H. Schulzrinne. An offer/answer model with the Session Description Protocol (SDP). Internet Engineering Task Force, June 2002. RFC 3264.

[22] J. Roskind. Quick UDP Internet Connections: Design Document and Specification Rationale, December 2013.

[23] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, July 2003.

[24] R. Stewart. Stream control transmission protocol. Internet Engineering Task Force, September 2007. RFC 4960.

[25] T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[26] M. Zanaty, V. Singh, S. Nandakumar, and Z. Sarker. Congestion control and codec interactions in RTP applications. Internet Engineering Task Force, March 2016. Work in Progress.