University of Glasgow | School of Computing Science

Transport-Layer Support for Multimedia Applications

Stephen McQuistin (0901634)

April 25, 2014

ABSTRACT

The majority of real-time multimedia traffic on the Internet is transported using the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP). Multimedia applications would benefit from a protocol that better supports their particular requirements, but transport layer ossification has limited the deployment of such protocols. This paper describes unordered time-lined TCP (uTLTCP), a modified version of TCP that is more suitable for real-time multimedia applications, whilst also benefiting from widespread deployability. Proof-of-concept evaluations show that this protocol increases the utility of the network when compared with unmodified TCP and UDP.

1. INTRODUCTION

Multimedia applications, including streaming video-on-demand applications and Voice over Internet Protocol (VoIP) services, account for a significant and growing percentage of all Internet traffic [5]. The popularity of these applications is growing because high-capacity Internet links have become commonplace, and this allows for an increase in the usability of multimedia applications. As these applications grow to take up a larger percentage of all Internet traffic, the properties of the protocols used to support them at the transport-layer become increasingly important to both the applications and the wider network.

At present, applications use either TCP or UDP at the transport layer, depending on their latency bounds. Interactive applications (e.g., VoIP) that have tighter latency bounds typically use UDP, while non-interactive applications (e.g., video-on-demand), where the bounds are weaker, use TCP. These two protocols present a set of trade-offs that developers must consider when deploying their applications: TCP introduces delay as a result of guaranteeing reliability, and while UDP does not introduce delay, but also provides no reliability guarantees or congestion control. Multimedia applications would benefit from a more flexible approach to the trade-off between reliability and latency, while also being congestion controlled. Introducing an entirely new protocol to support a multimedia-specific delivery model would be the prefered solution, but this would limit deployability on the Internet due to the presence of middleboxes, such as firewalls and Network Address Translators (NATs).

With this in mind, this paper presents unordered time-lined TCP (*u*TLTCP), a set of modifications to TCP improve its support for multimedia applications. *u*TLTCP combines, improves and extends on design elements from TCP Minion [25] and Time-lined TCP [23], to provide a partially reliable, time-lined datagram protocol on top of TCP. It makes further novel contributions by providing a sub-stream abstraction to support different types of traffic (e.g., control and data) multiplexed across a single connection. In addition, *uTLTCP* also allows applications to express dependencies between packets, which are coupled with the time-lined selective retransmission algorithm. Together, these changes are designed to allow the transport protocol to make efficient decisions about what data to send in order to increase the utility of the network.

This paper makes a number of contributions:

- *u*TLTCP, a deployable transport protocol that gives partial reliability (in the form of timelines and dependencies) and a datagram abstraction over TCP
- A proof-of-concept implementation of the protocol in FreeBSD, and an example application that uses it
- Evaluations that show that *u*TLTCP increases the utility of the network when compared with unmodified TCP and UDP

The remainder of this paper is structured as follows. Section 2 gives further details of the motivation for *u*TLTCP. Section 3 describes the protocol design, and section 4 presents evaluations conducted using a proof-of-concept implementation. Section 5 outlines existing work related to *u*TLTCP, and section 6 concludes.

2. BACKGROUND & MOTIVATION

At the transport layer, multimedia applications use either TCP or UDP, depending on their timeliness requirements. TCP provides many benefits to developers, including the ability to use existing, scalable infrastructure, such as content delivery networks and HTTP caches. In addition, applications that primarily use UDP can often use TCP as a fallback; this is because UDP is commonly blocked in enterprise firewalls, and can experience difficulties with NATs [31].

Despite its widespread adoption, the delivery model provided by TCP is suboptimal for the needs of multimedia applications. TCP offers guaranteed in-order delivery over a byte-stream abstraction, but this comes at the expense of timeliness – TCP will retransmit lost packets ahead of new data, and therefore introduce delays. While this model is suitable for applications where reliability is essential, this is not the case for real-time multimedia applications. Such applications are better suited to a model that offers *partial* reliability; some loss is tolerable if it reduces the amount of delay introduced.

Given the suboptimality of TCP's delivery model for real-time multimedia applications, a number of such applications

use UDP. In contrast to TCP, UDP offers no reliability or order guarantees, and therefore offers more flexibility to applications wishing to trade-off reliability with timeliness. Applications are able to build partial reliability on top of UDP at the application layer. However, UDP does not include a standardised congestion control algorithm. This, combined with the increasing use of multimedia applications, increases the potential for congestion collapse [8].

Protocols that combine features of both TCP and UDP, such as the Datagram Congestion Control Protocol (DCCP) [16] and the Stream Control Transmission Protocol (SCTP) [32], have been developed. However, deployment of these protocols has been limited by ossification within the transport layer [12]. In this context, ossification describes the entrenchment of TCP and UDP. This has occured because of the widespread use of middleboxes in the network to provide functionality, such as firewalls, and severely restricts the deployability of new transport protocols because of their use of unanticipated transport-layer headers.

The following subsections describe the delivery model that multimedia applications want from the transport layer, and further motivate supporting this delivery model by modifying TCP, rather than designing an entirely novel protocol.

2.1 Transport-layer support for multimedia

For many applications, such as file transfer mechanisms, the reliable byte-stream abstraction provided by TCP is appropriate. However, multimedia applications would benefit from framing data in application data units (ADUs) [6][11]. These application data units, or datagrams in UDP and *u*TLTCP terminology, allow greater flexibility for applications to define the relationship between the multimedia codecs used at the application layer, and the protocol responsible for the data at the transport layer. Applications should be able to process the contents of each datagram independently, and it is this that allows them to be more robust to packet loss; each datagram is self-contained and should provide some utility to the receiving application regardless of the delivery of other packets.

In real-time multimedia applications, the time by which packets are received is as important as the rate of packet loss, as packets are not useful after the time they are to be played out. This is in contrast to other applications, such as file transfer systems, where the utility of a given packet is constant over time. To motivate this, consider a video conferencing application. Each packet has a limited period of time during which it must be delivered, otherwise it can no longer be used; a frame of video arriving after it was to be played out cannot be reinserted into the stream. For other applications, such as video-on-demand streaming, the timeliness requirements may be relaxed, but they are still present.

The underlying packet switched networks of the Internet force protocols to trade-off between reliability and timeliness. This trade-off is intuitive: in order to guarantee successful delivery, packets that are lost or corrupted by the network are retransmitted. Given that congestion and flow control mechanisms restrict the number of packets in transit at a given time, it follows that packets will be delayed as a result of retransmissions. As discussed above, packets of real-time multimedia data have implicit arrival deadlines after which they are no longer useful. Therefore, transmitting packets that have exceeded these deadlines (or, will have exceeded their deadline before arrival) is futile. Going further than this, to transmit these packets actually contributes to a type of congestion collapse: the network is not performing useful work. Taken to the extreme, the network may be kept occupied by the transmission of packets that are not useful to the receiving application and that will be discarded because they have arrived too late to be used.

The use of a datagram abstraction in *u*TLTCP rather than a byte-stream means that out-of-order delivery is more practical than in standard TCP. Out-of-order delivery removes the increase in latency associated with head-of-line blocking. Head-of-line blocking occurs when the delivery of later packets is delayed by those sent earlier. This can be seen in TCP; to enforce in-order delivery, TCP will not deliver segments that have arrived if an earlier segment is missing. Those segments that have arrived may have to wait until the missing segment has been retransmitted, incurring a round-trip time of additional latency. In addition to out-of-order delivery, a datagram abstraction allows for TCP's reliability guarantees to be relaxed. Multimedia applications are tolerant to a certain amount of packet loss; small amounts of loss will be imperceptible to users.

In order to support the time sensitivity of multimedia traffic, *u*TLTCP allows applications to specify the deadline associated with each packet. This, combined with an estimate of the round-trip time between the sender and receiver, is used to determine whether a packet should be sent. If a packet is unlikely to be received in time to be useful, it is not sent. A sub-stream abstraction is used in order to allow applications to mix timelined and non-timelined traffic across a single connection; this is helpful for applications that send control information in addition to multimedia traffic. As an example, when the Real-Time Protocol (RTP) [30] is used over TCP, it is multiplexed with the RTP Control Protocol (RTCP) over the same TCP connection [18]; however, RTP and RTCP packets do not share the same timeliness requirements. The sub-stream abstraction accommodates this.

The introduction of timelines is at odds with the desire to only insert useful packets into the network. In many encoding formats a frame structure is in place that generates interdependencies between packets. For example, in MPEG-1 video [19], P-frames are dependent on the successful delivery of the previous I-frame. However, timelines may mean that this I-frame is not delivered, and so sending the P-frame would be futile (this paper uses MPEG-1 as an example because it is relatively straightforward; frame interdependencies also exist in newer codecs, such as H.264 [33]). To accomodate these interdependencies, *u*TLTCP allows applications to express the dependencies between packets. Where a given datagram's dependency has not been successfully transmitted (i.e., it has expired before being sent), then that datagram should not be sent, if a suitable alternative is available.

To complete the discussion of support for multimedia applications at the transport layer, congestion control and avoidance mechanisms should be considered. These are algorithms that are designed to prevent congestion collapse from occuring by limiting the rate at which new packets can enter the network. From the discussion about timeliness, it follows that lowering the rate will increase the delay to packets being queued up to be sent, and that this has a negative impact on multimedia applications. Congestion control algorithms should be developed to balance the timeliness requirements

of multimedia traffic, the nature of multimedia codecs and their sensitivity to changes in sending rate, and the other applications using the network. The development of such an algorithm is outwith the scope of the work presented here. *u*TLTCP does not change TCP's mechanisms; however, these are suboptimal for multimedia systems [27] and incorporating multimedia-friendly congestion control mechanisms into *u*TLTCP should be the subject of future work.

2.2 Transport-layer ossification

The discussion to this point has given the argument that an optimal delivery model for multimedia applications is sufficiently different from that of other applications that it should be supported at the transport layer. It remains to motivate the particular design direction taken by *u*TLTCP, that is, to provide this delivery model by way of modifications to TCP rather than by designing a new transport protocol optimised for the class of application. Initially, certainly from an ideological standpoint, designing a novel transport protocol for multimedia applications would be the desirable strategy.

However, as outlined in the previous section, the deployability of new protocols is severely limited by ossification in the transport layer [10]. There are a number of causes of this, including the desire to recoup investment in existing systems – organisations are not motivated to adopt new protocols when existing ones work. The primary, and perhaps the most intractable, cause of ossification, however, is the widespread use of middleboxes. Firewalls and NATs often violate the end-to-end principle [29] by inspecting packets beyond the IP headers; for example, firewalls often drop packets based on port numbers. The presence of middleboxes in the Internet can be justified by the essential services (e.g., firewalls and NATs) they provide, and their use is likely to continue.

This discussion then leads to the idea of modifying an existing protocol to provide support for multimedia applications. Ossification has led to two choices: TCP and UDP. Beyond the reasons already given, the choice between these two protocols is derived from either "building up" from UDP or "tearing down" from TCP to the delivery model described in the previous section. Many features of TCP, such as reliability (albeit partial) and congestion control are desirable for the design of *u*TLTCP. Modifying or relaxing these in TCP is more practical than implementing them entirely on top of UDP. For example, partial reliability can be achieved in TCP by changing the data that is retransmitted; in UDP, retransmissions would need to be added, and this is arguably more challenging [17].

The issue of ossification does not only limit the development of new protocols, but it also restricts the extent to which existing protocols can be modified. Honda et al. [13] present a study of the extensibility of TCP. This work provides a strong argument for the feasibility of the design decisions made in *u*TLTCP described in the next section.

3. UNORDERED TIME-LINED TCP

In section 2, three sets of modifications were given as being beneficial to multimedia applications: the use of application data units, timelines, and dependencies. In this section, each of these modifications will be described in detail.

The modifications are presented in order. Datagrams enable partial reliability, which allows timelines to be implemented. With timelines comes the possibility of data being sent that is not useful to the receiver; this motivates the inclusion of dependencies. Alongside an explanation of the modifications, the API that is exposed to developers is also given.

It is important to note that none of the modifications proposed change the wire protocol of TCP. It would not be possible to determine if a host is running TCP or *u*TLTCP from the packet headers. Wire-compatibility with TCP is essential to maximising the deployability of *u*TLTCP.

3.1 Datagrams and framing

Timelines and dependencies are only possible if partial reliability is enabled; expired packets, or packets dependent on expired packets, aren't sent. Providing partial reliability is achieved by adopting application data units, or datagrams, at the transport-layer. The use of datagrams allows applications to decide what data can be processed independently, and therefore the impact of a datagram not being received is reduced.

Building a datagram abstraction on top of the byte-stream used in TCP requires modifications on both the sender and the receiver; the sender must bypass optimisations in TCP that send longer segments where possible, while the receiver must be aware of the boundaries between datagrams. The modifications to TCP required at both the sender and receiver are detailed in the sections that follow.

Sender-side modifications

The sender-side modifications required to provide a datagram abstraction are based around ensuring that the application is able to define the contents of an outgoing TCP segment, or in other words, that a single `write()` call at the application-layer translates into a packet being sent on the network. In modifying the existing TCP protocol, there are three changes that need to be made: (i) disabling the Nagle [24] algorithm; (ii) marshalling the data before sending; and, (iii) exposing the path maximum transmission unit (PMTU) to allow applications to make efficient use of the protocol.

The Nagle algorithm is designed to increase the efficiency of TCP senders by reducing the overhead of the protocol. The size of a typical TCP/IP header is 40 bytes. This means that an application sending only 1 byte of data will send a packet 41 bytes in length, excluding additional headers from lower layers. The Nagle algorithm aims to amortise the 40 byte overhead of sending a TCP segment by buffering up smaller blocks of data into larger segments. Intuitively, this increase in efficiency comes at the expense of timeliness; small amounts of data sent at the application layer will be queued while waiting for subsequent writes. *u*TLTCP disables the Nagle algorithm by sending segments immediately, avoiding the resegmentation queue and therefore eliminating the delay introduced by the algorithm; this is the same behaviour as the `TCP_NODELAY` socket option. Performance evaluations conducted on disabling the Nagle algorithm [22] show that performance, in terms of latency, can be improved. However, they also serve as a warning: poor buffering at the application layer will lead to the problems that motivated the Nagle algorithm. Some of the multimedia applications suitable for *u*TLTCP, such as VoIP services, can generate small payloads; however, these applications benefit from making the trade-off between timeliness and efficient bandwidth use.

```
size_t send_dgram(int fd, char *buf, size_t len);
size_t recv_dgram(int fd, char *buf, size_t len);
size_t getPMTU(int sockfd);
```

Figure 1: Minimal *u*TLTCP API for datagram support

While bypassing the Nagle algorithm results in most application layer writes being sent as individual TCP segments, it does not eliminate the possibility that two or more writes may be coalesced. This may occur, for example, when two consecutive segments are retransmitted; rather than transmitting each of these as separate segments, these may be coalesed. While this is desirable behaviour (it does not introduce unnecessary delay, while also increasing bandwidth efficiency), it does mean that a read at the receiver may not correspond to a single write at the sender. In addition to this, Honda et al. [13] show that the behaviour of some middleboxes may include resegmenting packets; for example, they may receive two smaller segments, and forward only one larger segment. Therefore, data sent from the application-layer must be marshalled in order to preserve its boundaries. If segments are combined, for example, then the receiver must have sufficient knowledge to separate these into datagrams to deliver to the application.

In *u*TLTCP, marshalling is achieved by prepending and appending a zero byte to the data before it is sent on the network. To ensure that zero does not appear in the data being marshalled, *u*TLTCP encodes the data using the Consistent Overhead Byte Stuffing [4] (COBS) algorithm. This is a low overhead encoding algorithm that removes occurences of the zero byte, allowing it to be used as the marshalling character. Using two characters (i.e., appending and prepending) ensures that if middleboxes resegment packets in a way that splits datagrams across segments, it is still possible to reconstruct these at the receiver. The COBS encoding of applications writes takes place in a userspace library before the segments are written to the socket.

The sender-side modifications introduce two of the functions shown in figure 1: `send_dgram()` and `getPMTU()`. The `send_dgram()` API call is placed in a userspace library; it takes the socket file descriptor, data buffer, and length of data to be written. The data to be sent is COBS-encoded, and then written to the socket, with the length of the data written is returned. The `getPMTU()` API call returns the path MTU for the connection. If a segment is larger than this value, then it will be resegmented. By providing this value to applications, the number of resegmentations can be reduced, and this increases the efficiency of the COBS decoding algorithm at the receiver.

Receiver-side modifications

With the sender modified to send a segment for each write at the application layer, it remains for the receiver to be modified to turn each incoming segment into a read at the application layer. In modifying the TCP protocol at the receiver, there are two changes to be made: (i) avoiding reassembly of out-of-order segments into an in-order byte-stream; and (ii) unmarshalling the data before delivering it to the application.

As TCP supports an in-order byte-stream abstraction, it deals with the reception of out-of-order segments by buffering them in a reassembly queue, until prior segments have been received. For applications that are dependent on reliability, this makes sense: applications would themselves have to reassemble delivered segments. However, where application data units are being used, each segment contains a unit of data that is useful to the application independently of other units. Applications using a datagram abstraction are designed to deal with lost and out-of-order datagrams; there is no value in delivering datagrams in order.

In *u*TLTCP, the kernel is modified to ensure that each read from the socket buffer delivers a single TCP segment to the application. In order to maintain wire-compatibility with TCP, while incoming segments bypass the reassembly queue, the standard TCP responses to out-of-order packet reception are maintained. This includes sending acknowledgements; while the segment is delivered to the application, reception statistics are maintained as if the segment had not been received.

Given that the sender-side modifications cannot guarantee that a TCP segment corresponds to a single datagram, a userspace library is used to decode incoming segments into datagrams for delivery to the application. In most cases, incoming segments from a *u*TLTCP sender will contain one or more complete COBS-encoded datagrams; in this case, the zeroes are removed from the beginning and end of the datagram, and the remaining data is decoded using the COBS algorithm. However, a datagram may span more than one segment, and to support this the userspace library must know how to coalesce segments. To support this the kernel is modified to append the TCP sequence number corresponding to the segment being read. This, combined with the length of the segment, is sufficient to allow segments to be combined; this combination continues until a complete COBS-encoded datagram has been read.

The receiver-side modifications introduce the `recv_dgram()` API call shown in figure 1. The functions takes the socket file descriptor, data buffer, and maximum read length; it writes a datagram to the buffer, and returns the length of the datagram. The datagram is a COBS-decoded segment that is either read from the socket receive buffer, or has been buffered in userspace. Userspace buffering of datagrams can occur where a single segment contained multiple datagrams; only one is delivered with each call to `recv_dgram()`.

3.2 Timelines

The use of timelines is designed to prevent expired segments from reaching the receiver, by having the sender estimate the time at which they will be received. As segments are sent, several steps are taken within the kernel: (i) it is determined if the segment will arrive on time, based on the presentation time set by the application; (ii) if the segment has expired, and a suitable, unexpired segment is found, then this is sent instead.

This subsection describes the design of timelines within *u*TLTCP, and particularly how it answers the questions that the basic outline above presents, including what is meant by a "suitable" replacement segment, and what happens if the replacement is smaller than the original segment. It begins by discussing how a segment is deemed to have expired, followed by the replacement segment policy adopted, and concludes with a description of the sub-stream design.

```
size_t send_dgram(int fd, char *buf, size_t len);
size_t send_dgram(int fd, char *buf, size_t len,
                  uint32_t playoutTime);
size_t send_dgram(int fd, char *buf, size_t len,
                  uint8_t substream);
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 2: uTLTCP API including timeline and sub-stream support

Segment liveness

To estimate if a segment will arrive at the receiver on time to be played out, two values are required: (i) an estimate of the round-trip time between the sender and receiver, and (ii) the playout time of the segment.

The round-trip time (RTT) estimate used is important. If this estimate is too optimistic (i.e., the RTT estimate is too low) then the protocol will not be effective in preventing expired segments from entering the network. However, if the estimate is too high, then useful, unexpired data, will not be sent. The protocol leans towards using an optimistic estimate; not sending useful data is more damaging than sending unexpired data. The estimate used is the smoothed round-trip time (*srtt*) [26][14] value computed as part of the TCP protocol specification. This is a weighted average that moves slowly towards the correct average, and responds slowly to rapid variations in round-trip time.

uTLTCP is designed for multimedia applications, and these applications use a wide number of codecs. To support this, the API is extended as shown in figure 2 to allow applications to specify both a playout timestamp, and the clock rate for these timestamps; `setClockrate()` takes the clock rate expressed in Hz. The timestamp of the initial segment sent, plus half of the round-trip time estimate, is taken as the start of the playout of the timelined segments. For subsequent segments, the deadline is calculated using the offset from this initial time. The offset is calculated using the clockrate, playout time, and round-trip time estimate. An estimate is also made of the current time on the receiver (in terms of the playout of received datagrams). If the segment is estimated to arrive after the current estimated playout time, it has expired, and should not be sent. The next step is to select a suitable replacement to be sent in its place.

Segment replacement

Replacing expired segments is important, as it increases the chance of more unexpired data reaching the receiver. The ideal replacement segment is one that is timelined, unexpired, unsent, and appropriately sized. Expired segments are replaced only with timelined data, because the application has expressed a time bound for these segments. In addition, unsent data is preferred because sent data already has the potential to have been successfully received – if it hasn't, then upon retransmission, it will be replaced if it has expired. A replacement segment is not suitable if it is larger than the segment it is replacing; smaller replacements are suitable, and they will be discussed in the next subsection.

The replacement of expired segments with unexpired and unsent segments is the preferred solution. However, if there is not a sufficient number of segments in the socket send buffer, then these conditions cannot be met. uTLTCP will send a segment that has already been sent if this is all that is available, and similarly, will resend the expired segment if no unexpired segments are available. This behaviour is designed to ensure that TCP flow and congestion algorithms are not affected. This also aids deployability; data needs to be sent to ensure this.

Sub-streams and padding

uTLTCP includes a sub-stream abstraction, where datagrams (i.e., COBS-encoded segments) are given stream types. Primarily, uTLTCP flows consist of three types of stream: one padding stream, one timelined stream, and one or more auxiliary non-timelined streams. Each segment of data written at the application layer has a stream identifier prepended before being encoded using COBS. This identifier is then removed by the userspace datagram library at the receiver, and the datagram processed appropriately.

When a replacement segment is sent that is smaller than the original, a datagram within the padding stream is appended to the replacement. This datagram is sized to ensure that the TCP segment is the same size as the original; this results in better deployability, and allows for receivers without kernel modifications. At the receiver, datagrams within the padding stream are discarded by the userspace datagram library and are not sent to the receiving application.

The sub-stream abstraction differentiates between timelined and non-timelined data, and this allows for these two types of data flows to be multiplexed across the same connection. As shown in figure 2, the API has been augmented to allow for the specification of the substream identifier. This can be any integer, above 2 (1 is reserved for the padding substream, while 2 is reserved for timelined data), chosen by the application.

Example application

To illustrate the timelines API and the motivation for the sub-streams abstraction, consider an application using RTP and RTCP. The RTP packets are timelined according to the codec, which specifies a clock rate based on the sampling interval or playout time. For MPEG-1 video, the clock rate is 90000Hz, and this would be specified to uTLTCP using the userspace library call `setClockrate(90000)`. RTP timestamps would therefore be sent using the `send_dgram(fd, buf, len, presentationTs)` call, where `presentationTs` is the presentation timestamp of the RTP packet. As RTCP packets are not explicitly timelined, they are sent using the `send_dgram(fd, buf, len)` call; by default, these will be sent on stream identifier 3, as this is the first non-timelined sub-stream.

If there is an additional stream of non-timelined control data, such as that from the Real Time Streaming Protocol (RTSP), then the application must explicitly specify the stream identifier it wishes to use. For example, when sending RTSP data it may use the `send_dgram(fd, buf, len, substream)` call; then, the receiving application can use `recv_dgram(fd, buf, len, substream)` to receive both the datagram, and the sub-stream identifier. A more up-to-date protocol that would benefit from transport-layer support of a sub-stream abstraction is WebRTC [1], which includes sup-

port from multiple independent data channels. These data channels could be mapped to uTLTCP sub-streams.

Figure 3 shows how application data units written at the application layer are modified before being passed to the transport layer. First, the single byte sub-stream identifier is prepended to the ADU. This is then COBS-encoded, and a zero is appended and prepended to the result. It is this zero delimited block that is then passed to the transport layer. The worst-case overhead of the COBS encoding algorithm is 1 byte in every 254 bytes, or 0.4%, as shown in figure 3. While this is a low overhead, this combined with disabling the Nagle algorithm means that uTLTCP is more efficient with moderately sized datagrams, in terms of being able to amortise the overhead of both COBS and TCP headers.

3.3 Dependencies

While it is possible for many datagrams to be processed independently by the receiving application, for most multimedia applications there exists an interdependency between datagrams. To motivate this, figure 4 shows the interdependencies between different frame types in MPEG-1 video. In MPEG-1 [28], video is encoded using three frame types: I (independent), P (predictive) and B (bi-directionally predictive) frames. In terms of dependencies, P frames are dependent on earlier I and P frames, while B frames are dependent on both the previous I or P frame, and the next I or P frame. The timeline modification may lead to an expired frame not being sent; if this is the case, then any P or B frames that depend on it will not be useful to the receiver. Therefore, the segments containing these frames should also be treated as if they have expired.

To enable the kernel to use dependencies to check if a packet should be sent, the dependencies need to be expressed by the sending application. The modifications required to the API are described in the next section.

Expressing dependencies

To allow applications to express dependencies between datagrams, the API has been extended to include the specification of a sequence number for each datagram. In addition, the sequence number of the datagram upon which the datagram being sent depends can also be specified. Figure 5 shows the extended API used to support this.

Segment replacement

If dependencies are expressed by the application, then they are used as an additional factor in the selective retransmission algorithm introduced by timelines. If a given datagram expires (i.e., it is not sent successfully before reaching the end of its timeline), then any datagrams that are dependent on it also expire. This means that if the datagram is to be sent (either for the first time, or as a retransmission) then, where possible, it will be replaced by a suitable alternative.

Only the sending API calls have been modified, as dependencies are only used at the sender side and are not transmitted with the datagram. Where the API function does not include the sequence number of the datagram on which the datagram being sent depends, it is assumed that it is independent. This translates to being dependent on itself, in terms of the kernel level modifications made to support dependencies.

Example application

To motivate these changes, consider an application streaming real-time video using MPEG-1. The first 9 frames may be as shown in figure 4, and these may have sequence numbers from 0 to 9. The first datagram (sequence number 0) contains an I-frame; to express that this is independent, it sets its dependency to itself, 0. This can be set explicitly, or will be assumed if no sequence number is given in the call to send the datagram. The next datagram (1) contains a B-frame that is dependent on both datagram 0, and on datagram 3. Given that datagram 3 is that the API limits its dependencies to a single datagram, datagram 1 sets its dependency to datagram 0. This follows until the next independent frame (shown in figure 4 as a frame without any arrows pointing into it) is sent.

This discussion assumes that a frame of MPEG-1 video corresponds to a single datagram; of course, typically, this will not be the case. A single frame may be spread across several datagrams, depending on the type of frame (I-frames are larger than P-frames, which are larger than B-frames). To express this using the API, the datagrams that comprise a single frame will have different sequence numbers, the same timestamp, and be dependent on each other. This means that if one datagram containing a part of a frame is lost, then it will only be resent if it is both on time, and if the other parts of the frame have not expired.

4. EVALUATION

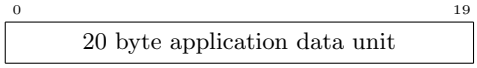
The motivation for uTLTCP is to design a transport layer protocol that is more performant than existing protocols, whilst also being deployable. Therefore, to evaluate the extent to which the design of uTLTCP has met these goals, the evaluation must focus on both the performance of the protocol against other protocols, and also on its deployability.

The first part of this section presents performance tests carried out on a testbed that measures how uTLTCP performs again TCP and UDP, with performance being measured using a number of metrics. The second part discusses the work by Honda et al. [13] that is used to justify the claim that uTLTCP is deployable, and gives details of an evaluation to add to this justification. The section concludes by discussing and critically reflecting on the evaluation.

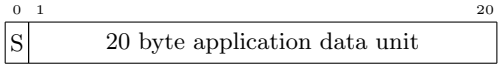
4.1 Performance

The main purpose of designing any new protocol should be that it is in some way "better" than what has gone before. In motivating the protocol it has been shown that the API provided by uTLTCP is better for application developers in that it allows them to be more expressive about the traffic their applications are generating. However, it remains to be shown that uTLTCP is better in terms of its performance.

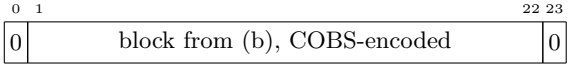
This section begins by describing the metrics that will be measured as part of the evaluation, and outlining the design of the testbed upon which it will be carried out. The remainder of the section provides the results of the evaluation, empirically motivating each change as it builds on the previous change, beginning with the introduction of the datagram abstraction.



(a) 20 byte `write()` at application layer



(b) S (sub-stream identifier) prepended before COBS encoding



(c) COBS-encoded datagram to be passed to transport layer

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

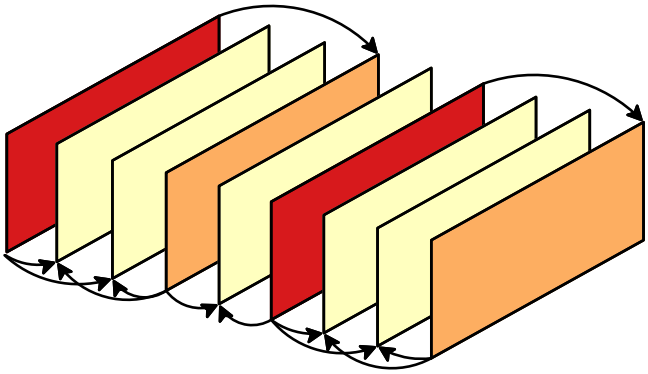


Figure 4: MPEG-1 video frame prediction between I-frames (red), P-frames (orange) and B-frames (yellow)

```
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: *u*TLTCP 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 *u*TLTCP 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 *u*TLTCP sender and a TCP receiver is evaluated because this configuration is more deployable than having a *u*TLTCP 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

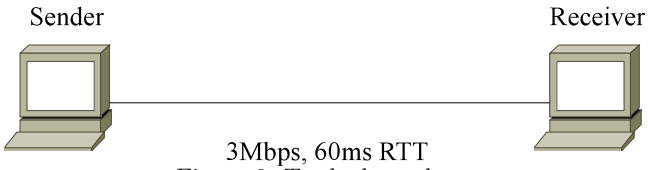


Figure 6: Testbed topology

Label	Sender	Receiver
A	TCP	TCP
B	UDP	UDP
C	*u*TLTCP	TCP
D	*u*TLTCP	*u*TLTCP

Table 1: Protocols under evaluation

20ms between each packet; where timelines are being tested, the clockrate is 8000Hz. The size of datagrams will alternate 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 clock rate and packet sizes have been selected to imitate that of audio transmission. However, in such applications, packet sizes are usually constant. They vary here only to 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

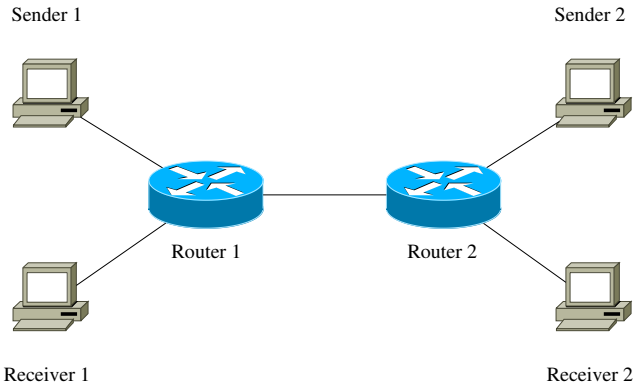


Figure 7: Testbed topology for TCP fairness measurements

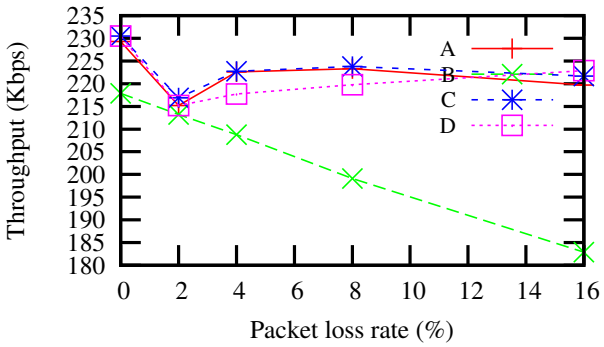


Figure 8: Throughput vs packet loss for datagram modifications

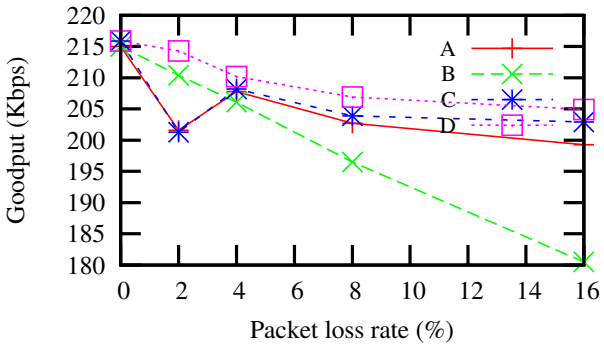


Figure 9: Goodput vs packet loss for datagram modifications

packets delivered to the application. As with goodput, packets that are too late (i.e., arrive after their playout time) are not counted. This is defined in RFC 3550 [30].

To support these metrics, each packet will contain a sequence number, and two timestamps, in both microseconds and in units of the clock rate. The sequence number will allow the timing offset from the previous packet in the sequence to be calculated (for goodput and interarrival jitter calculations), while the clock rate timestamp will allow interarrival jitter to be calculated. The microsecond timestamp is used to measure latency. The clocks on both the sender and receiver are synchronised to a central time server using the Network Time Protocol (NTP) [21]. Dummynet is used specify the bandwidth and RTT values shown in figure 6, and is also used to modify the packet loss rate. A combination of `tcpdump` and `tcptrace` is used to measure throughput and goodput; other measurements are taken by the application.

The fifth metric that will be measured is **Jain's fairness index** [15]. This is a measure of the equality of the bandwidth allocation between competing flows. For n connections, where x_i is the bandwidth for the ith connection, the fairness index is defined as:

$$fairness\_index = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2} \quad (1)$$

Figure 7 shows the topology of the testbed used to calculate the fairness index. uTLTCP will be used to send data from sender 1 to receiver 2, while TCP will be used between sender 2 and receiver 1. These two flows will be long-lived and started at the same time, the bandwidth allocation that they each receive will be measured at the receivers, and the fairness index calculated using equation 1.

Datagram and framing

Figure 8 shows throughput. Throughput is higher for the TCP-based protocols because TCP is a reliable protocol, and so packets are retransmitted if lost. This means that the variable factor here is the time that it takes to send all of the data, giving a significantly higher throughput than UDP at higher loss rates. There is no significant variation in throughput between the three TCP-based protocols (labelled A, C and D). This is expected because none of the modifications introduced by datagrams change the amount of data being transmitted.

Figure 9 shows goodput. For all protocols, goodput is lower than throughput. While UDP shows goodput falling rapidly, the TCP-based protocols do not. Given the delays introduced by packet loss (i.e., head-of-line blocking in TCP), goodput should have been substantially lower. It may be the case that the evaluation code is incorrectly identifying packets as having arrived on time, and is therefore showing a higher than expected goodput result. Goodput should increase substantially when using a uTLTCP receiver, because such a receiver removes head-of-line blocking. The results given here do not adequately reflect this.

Figure 10 shows latency. For UDP, latency does not vary significantly as the packet loss rate is increased. This is somewhat intuitive; lost packets are not retransmitted, and therefore there is no increased delay to subsequent packets. Latency increases as the packet loss rate increases for the TCP-based protocols. Where head-of-line blocking is removed (i.e., when a uTLTCP receiver is used), latency is significantly lower.

Figure 11 shows interarrival jitter. As for latency, interarrival jitter does not fluctuate significantly when UDP is used, but increases with the packet loss rate for the TCP-based protocols. The uTLTCP receiver lowers interarrival jitter when compared with the TCP receiver, again due to the removal of head-of-line blocking from TCP.

Jain's fairness index for TCP and uTLTCP with datagrams (labeled A) is given in figure 12. uTLTCP is compared with TCP with the `TCP_NODELAY` socket option enabled. This makes the protocols the same in terms of how

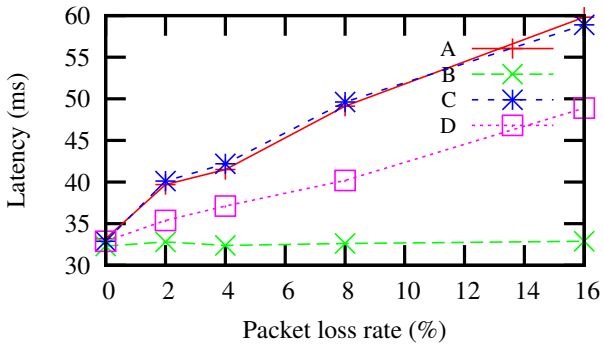


Figure 10: Latency vs packet loss for datagram modifications

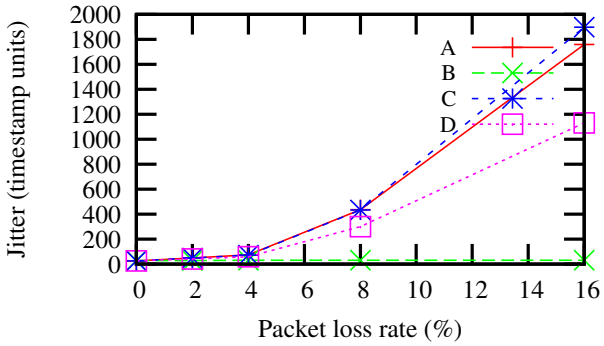


Figure 11: Interarrival jitter vs packet loss for datagram modifications

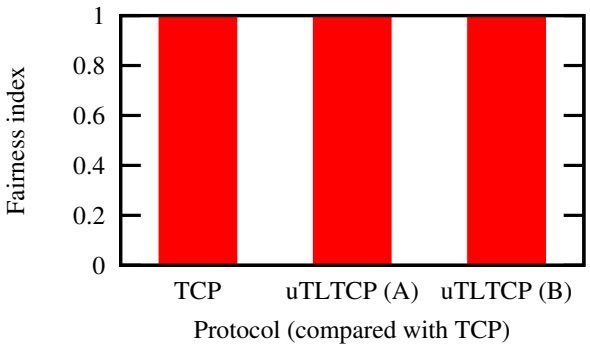


Figure 12: Fairness ratios for all modifications

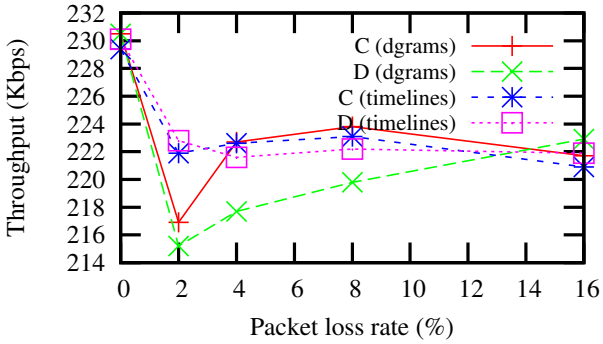


Figure 13: Throughput vs packet loss for timeline modifications

they appear on the wire. Out-of-order delivery does not affect the fairness of the protocol, as no change is made to how the receiver deals with packet loss. From figure 12, it can be concluded that *u*TLTCP with datagram support is fair to competing TCP flows.

Timelines

As shown in figure 13, the introduction of timelines does not cause significant variation in throughput. This is expected, as the amount of data being sent is not changed. In addition, *u*TLTCP does not modify TCP's response to packet loss, and it is this that would most likely affect the throughput measurements.

While figure 14 shows a small increase in goodput when timelines are introduced, this does not appear to be significant. If the selective retransmission algorithm is working optimaly, then the majority expired traffic (i.e., "badput") should be prevented from reaching the receiver. This is not the case here, especially at higher loss rates; future work should focus on when the algorithm works optimally.

Figures 15 and 16 show small declines in latency and interarrival jitter as packet loss rates increase. These are not significant, and this follows from the discussion of the goodput results when timelines are introduced. From these three metrics, it is clear that not enough inconsistent retransmissions are taking place to significantly reduce the amount of expired traffic arriving at the receiver.

Jain's fairness index for TCP and *u*TLTCP with timelines (labeled B) is given in figure 12. *u*TLTCP with timelines is fair to competing TCP flows (with the `TCP_NODELAY` socket option set). This is somewhat intuitive given that while the timeline modifications change what data is sent in expired packets, the amount and timing of the data that is sent is not changed.

Dependencies

Figures 17, 18, 19 and 20 all show small and insignificant changes in the metrics being measured as a result of introducing dependencies. This indicates that only a small number (if any) of additional inconsistent retransmissions are being made, over and above those being made as due to timelines.

This is likely to be as a result of the evaluation design: in testing for dependencies, every fifth packet was expressed as being independent, with other packets being dependent on the previous packet. These are relatively small dependency chains, and this combined with the random packet loss inserted by Dummynet, may combine to mitigate the effects of the dependency modification. As with timelines, further work should be carried out to expand the evaluation, so as to identify the patterns of traffic that are most likely to trigger the modifications present in *u*TLTCP.

No further measurements of Jain's fairness index are required to verify the fairness of the dependency modifications. These modifications only serve to add an additional factor to the selective retransmission algorithm introduced by timelines, and so the fairness index given for *u*TLTCP with timelines applies also to dependencies.

4.2 Deployability

A study performed by Honda et al. [13] evaluates the

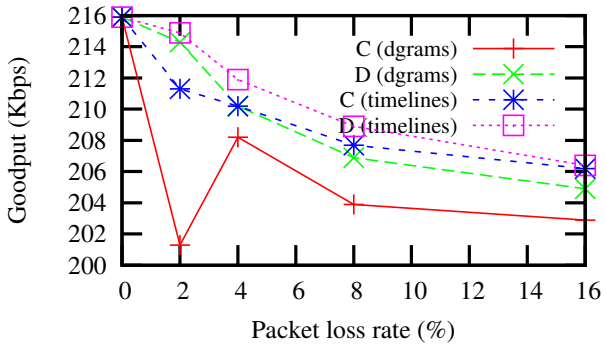


Figure 14: Goodput vs packet loss for timeline modifications

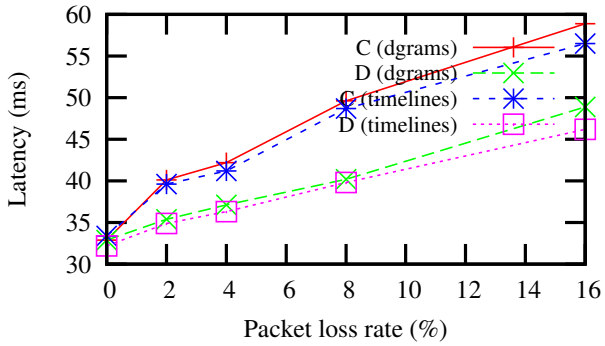


Figure 15: Latency vs packet loss for timeline modifications

behaviour observed when TCP retransmissions are inconsistent (i.e., they do not match the original transmission); it is this work that justifies the claim made here that uTLTCP is deployable. The evaluation carried out by Honda et al. involved sending two segments, with the receiver acknowledging only the first of these. The receiver would then send a duplicate acknowledgement for the first segment, indicating that the second segment has been lost. After this, the sender sends a new segment with the same sequence number as the "lost" segment, but with a different payload. This was repeated for segments that were smaller, the same, or larger than the original transmission. The authors identified four middlebox responses to inconsistent retransmissions:

- The new segment was received successfully;
- The middlebox cached the original, and sent this instead of the retransmission;
- There was no response at all to the duplicate acknowledgement;
- The connection was reset.

For the most part, the first two of these was observed. This means that applications should be sufficiently flexible to receive either new data, or the retransmission. As discussed previously, the use of a datagram abstraction means that this is the case for applications using uTLTCP – sufficient information is contained in both the original, and any new segment, to allow the receiver to make use of the data independently from other datagrams.

Given the nature of the Internet, it is not practical to show that every middlebox will interact positively with the

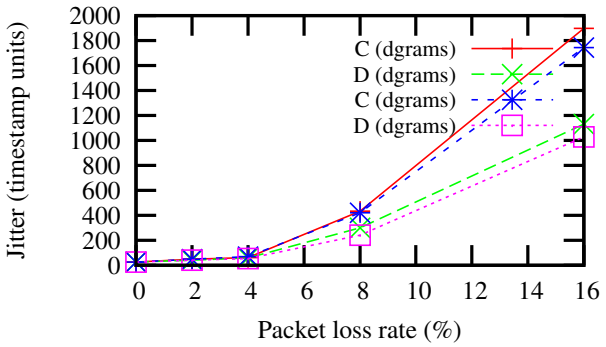


Figure 16: Interarrival jitter vs packet loss for timeline modifications

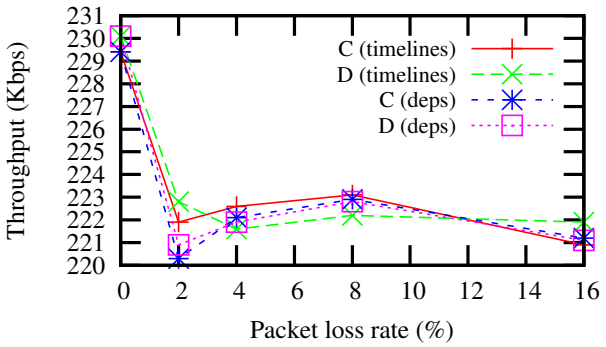


Figure 17: Throughput vs packet loss for dependency modifications

modifications that are part of uTLTCP. Therefore, while Honda et al. go some way towards validating the deployability claims made in the design of uTLTCP, increasing the body of evidence that shows this is important. To do this, an evaluation has been designed to show the deployability of inconsistent retransmissions, which are the basis for both timelines and dependencies.

The evaluation comprises a FreeBSD server on the Internet, modified to change the contents of all TCP retransmissions so that they do not match the original data that has been sent. On the client, `tcpdump` is used to record all incoming packets, and Dummynet [2] is used to drop a small percentage of packets so as to trigger a retransmission. If the packets recorded by tcpdump show two TCP segments with the same sequence number and different payloads (assuming that sequence numbers have not wrapped around), then this means that inconsistent retransmissions are permitted.

Only one evaluation of this type has been carried out; this was completed successfully over a major UK Internet Service Provider (ISP). Clearly, this is not a significant contribution to the claim of deployability. More time would be required to perform a larger number of evaluations of this type. In addition, performing this purely in software limits the number of people who are willing or able to complete the evaluation. Ideally, this evaluation would use a small device (e.g., Raspberry Pi) that has been pre-configured with the task of performing the evaluation. Including this as part of future work is essential to increase confidence in the deployability of uTLTCP.

4.3 Discussion and critical reflection

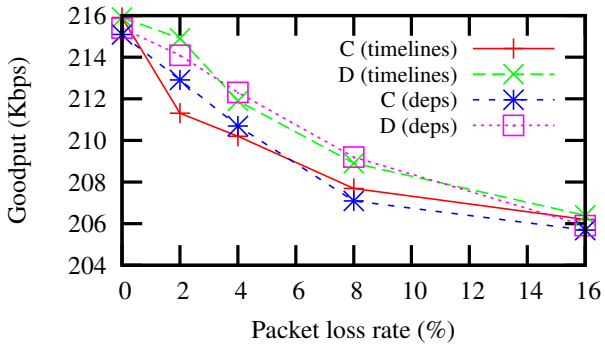


Figure 18: Goodput vs packet loss for dependency modifications

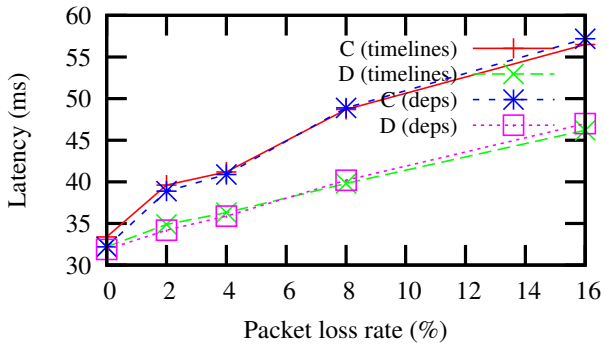


Figure 19: Latency vs packet loss for dependency modifications

The performance evaluation results given in section 4.1 show that there is some benefit, when measuring key metrics, to implementing uTLTCP. These results, however, do not show a *significant* improvement being gained by using timelines and dependencies. This may not be fully representative of the performance benefits of these modifications, either because the proof-of-concept implementation is not performant, or, more likely, because the evaluation testbed does not provide the optimal environment.

Using Dummynet to simulate packet loss does not fully exercise the modifications presented here. When simulated in this way, the packet loss is uniformly distributed throughout the flow, and this does not reflect how packet loss is typically observed in the Internet. Packet loss is generally bursty (i.e., groups of consecutive packets are lost) because of the drop-tail queueing mechanisms in use at routers. It is not clear from these results if the modifications presented here would be more performant when packet loss is more bursty, and further analysis is required to show this.

In addition, a single, constant delay is present in the evaluation testbed. This is also not representative of the network conditions in the Internet. Competing flows and changing links can both contribute to variations in end-to-end delay, but such variations are not characterised in the evaluation. It is not clear if uTLTCP would provide an increased performance benefit if there were larger delay variations, including spikes in latency. These modifications should allow for better recovery of the application by limiting the expired traffic in the network.

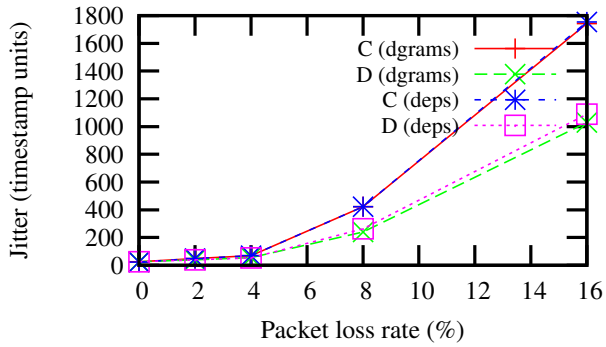


Figure 20: Interarrival jitter vs packet loss for dependency modifications

Given more time, the evaluation would be modified to better analyse the conditions under which uTLTCP delivers the most benefit. The difficulty of implementing the protocol in FreeBSD has not allowed this within the time constraints of the work. Modifying the kernel of FreeBSD and rebuilding it are non-trivial and time consuming tasks, and this limited the extent to which the implementation could be evaluated. The performance of the FreeBSD implementation is also limited by lack of experience in kernel programming.

5. RELATED WORK

uTLTCP primarily builds on Minion, a set of modifications to TCP proposed by Nowlan et al., and Time-lined TCP, proposed by Mukherjee and Brecht. In proposing Minion, Nowlan et al. were clear that it adopts a conservative design in order to guarantee deployability, and to ensure strict wire-compatibility with TCP. This design means that while Minion allows for changes to the sending buffer, no changes are allowed to data that has already been sent – this rules out inconsistent retransmissions. In Time-lined TCP, inconsistent retransmissions are used, but in such a way that leaves gaps in the sequence space and requires a modified TCP implementation at the receiver.

The work by Honda et al. allows the design of uTLTCP to combine both Minion and Time-lined TCP, but in such a way as to minimise the impact on deployability. As discussed in section 4.2, their study found that inconsistent retransmissions did not broadly impact upon deployability. However, the study also showed that leaving gaps in the sequence space led to undesirable behaviour from middleboxes, and that this should be avoided. Taking both of these findings into consideration means that uTLTCP uses inconsistent retransmissions to implement timelines, but uses padding and consistently sized retransmissions to maximise deployability.

There have been a number of other protocols that have been proposed to better support multimedia applications at the transport layer. Dempsey et al. [7] introduce Partially Error Controlled Connection (PECC). PECC adopts a timeline approach, in that its retransmission algorithm is modified to only retransmit packets where the timing requirements of the application are met. Grinnemo and Brunstrom [9] present PRTP-ECN, a TCP-compliant partially reliable transport protocol. It is similar to other protocols, including uTLTCP, in that packets will only be retransmitted if they are thought to be useful to the application. PRTP-ECN does this by modifying the receiver to acknowledge segments that

haven't arrived, preventing their retransmission. However, modifying the receiver may not be the best strategy in terms of deployability. Cen et al. describe the Streaming Control Protocol (SCP) [3], a set of modifications to TCP's flow and congestion control algorithms. These modifications aim to remove the unpredictable latency variations introduced by TCP's algorithms. Finally, as discussed earlier, newer transport protocols such as DCCP and SCTP exist to broaden the support for different applications at the transport-layer. Deployability of these protocols is a concern, given their relatively low deployment at present [12].

The discussion of related work presented here is not exhaustive; there have been many efforts made to improve transport-layer support for multimedia applications. Those protocols and designs highlighted here are sufficiently different approaches. The main difference between other efforts and *u*TLTCP is the focus on deployability. The majority of the related work in the area has not considered the interaction between the protocol and middleboxes.

6. CONCLUSIONS & FUTURE WORK

This paper presented *u*TLTCP, a modified version of TCP that includes datagrams, timelines and dependencies. Providing a novel transport-layer protocol by modifying TCP has allowed *u*TLTCP to be deployable. Ossification has meant that such modifications are essentially the only way to provide new functionality at the transport-layer. Evaluations of *u*TLTCP show that it performs favourably to TCP and UDP, increasing goodput while reducing latency and interarrival jitter, all of which are good measures of the performance of protocols for real-time multimedia applications. Future work should include strengthening the evaluation of the protocol, either using the current proof-of-concept implementation, or be implementing the protocol in a simulation testbed. More work should be done to profile the performance of the protocol, and also to bolster the deployability claim.

This paper, and the design of *u*TLTCP, largely ignores the interaction between real-time multimedia traffic and TCP's congestion control algorithm. TCP uses a number of algorithms to provide congestion and flow control, and these algorithms negatively impact timeliness. A number of proposals, including Google's congestion control algorithm [20], have been made for congestion control algorithms that are more suitable for real-time multimedia applications. The future work that leads from *u*TLTCP should include incorporating one such algorithm and evaluating the resultant performance.

*u*TLTCP is but a single point in a wider design space of deployable domain-specific transport-layer protocols. The decisions taken in its design (e.g., to modify TCP) reflect the particular requirements of the domain in which *u*TLTCP exists. In the future, other domains may benefit from transport-layer support for their applications, such as the power consumption requirements of small sensor devices. The design and use of domain-specific protocols is likely to increase as it becomes clearer that not all applications are ideally suited to existing protocols.

Future work resulting from *u*TLTCP should seek to explore the design space of deployable transport-layer protocols; "greenfield" development of new protocols is no longer pragmatic if deployability is a consideration. The design of other protocols could take on of two approaches: design with more support for specific attributes of the traffic, such as incorporating RTP headers and using this data rather than an expressive API, or opening up the API and allowing applications more flexibility and access to the underlying transport layer. Designs in either of these directions, however, should consider the existing, entrenched transport-layer protocols as essential substrates. Some designs will favour modifying UDP rather than TCP, but either way, UDP and TCP provide the envelope within which deployable protocols are to be developed in future.

Acknowledgments

Thank you to Colin Perkins, both for the initial project idea and for help, support and guidance throughout.

REFERENCES

[1] A. Bergkvist, D. C. Burnett, C. Jennings, and A. Narayanan. WebRTC 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

[2] M. Carbone and L. Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010.

[3] S. Cen, J. Walpole, and C. Pu. Flow and congestion control for internet media streaming applications. In *Photonics West'98 Electronic Imaging*, pages 250–264. International Society for Optics and Photonics, 1997.

[4] S. Cheshire and M. Baker. Consistent overhead byte stuffing. *IEEE/ACM Trans. Netw.*, 7(2):159–172, Apr. 1999.

[5] Cisco. Cisco Visual Networking Index: Forecast and Methodology, 2012-2017. White Paper, May 2013.

[6] D. D. Clark and D. L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, SIGCOMM '90, pages 200–208, Philadelphia, Pennsylvania, USA, 1990. ACM.

[7] B. Dempsey, T. Strayer, and A. Weaver. Adaptive Error Control for Multimedia Data Transfer. In *Proceedings of the IWACA*, volume 92, pages 279–288, 1992.

[8] S. Floyd and K. Fall. Promoting the Use of End-to-end Congestion Control in the Internet. *IEEE/ACM Trans. Netw.*, 7(4):458–472, Aug. 1999.

[9] K.-J. Grinnemo and A. Brunstrom. Evaluation of the QoS Offered by PRTP-ECN-A TCP-Compliant Partially Reliable Transport Protocol. In *Quality of Service - IWQoS 2001*, pages 217–230. Springer, 2001.

[10] M. Handley. Why the Internet only just works. *BT Technology Journal*, 24(3):119–129, 2006.

[11] M. Handley and C. Perkins. Guidelines for Writers of RTP Payload Format Specifications. RFC 2736 (Best Current Practice), Dec. 1999.

[12] S. Hätönen, A. Nyrhinen, L. Eggert, S. Strowes, P. Sarolahti, and M. Kojo. An Experimental Study of Home Gateway Characteristics. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 260–266, Melbourne, Australia, 2010. ACM.

[13] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is It Still Possible to

Extend TCP? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 181–194, Berlin, Germany, 2011. ACM.

[14] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[15] R. Jain, D.-M. Chiu, and W. R. Hawe. *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*. Eastern Research Laboratory, Digital Equipment Corporation, 1984.

[16] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), Mar. 2006. Updated by RFCs 5595, 5596, 6335, 6773.

[17] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion Control Without Reliability. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 27–38, Pisa, Italy, 2006. ACM.

[18] J. Lazzaro. Framing Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP) Packets over Connection-Oriented Transport. RFC 4571 (Proposed Standard). IETF, July 2006.

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

[20] H. Lundin, S. Holmer, and H. T. Alvestrand. A Google Congestion Control Algorithm for Real-Time Communication. IETF Internet Draft – work in progress, Network Working Group, August 2013.

[21] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.

[22] G. Minshall, Y. Saito, J. C. Mogul, and B. Verghese. Application performance pitfalls and TCP's Nagle algorithm. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):36–44, 2000.

[23] B. Mukherjee and T. Brecht. Time-lined TCP for the TCP-friendly delivery of streaming media. In *International Conference on Network Protocols, 2000*, pages 165–176. IEEE, 2000.

[24] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896. IETF, Jan. 1984.

[25] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Fordy. Fitting Pegs Through Round Pipes: Unordered Delivery Wire-compatible with TCP and TLS. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 28–28, San Jose, CA, USA, 2012. USENIX Association.

[26] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988. IETF, Nov. 2000.

[27] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the Internet. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1337–1345. IEEE, 1999.

[28] L. A. Rowe, K. D. Mayer-Patel, B. C. Smith, and K. Liu. MPEG video in software: representation, transmission, and playback, 1994.

[29] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[30] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (INTERNET STANDARD), July 2003. Updated by RFCs 5506, 5761, 6051, 6222, 7022.

[31] K. Sripanidkulchai, B. Maggs, and H. Zhang. An Analysis of Live Streaming Workloads on the Internet. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*, IMC '04, pages 41–54, Taormina, Sicily, Italy, 2004. ACM.

[32] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007. Updated by RFCs 6096, 6335.

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