Deployable Transport Services for Low-Latency Multimedia Applications

Stephen McQuistin

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

Abstract

Low-latency multimedia applications generate a growing and significant majority of all Internet traffic. These applications are characterised by tight bounds on end-to-end latency that typically range from tens to a few hundred milliseconds. Operating within these bounds is challenging, with the best-effort delivery service of the Internet giving rise to unreliable delivery with unpredictable latency. The way in which the upper layers of the protocol stack manage this unreliability and unpredictability can greatly impact the quality-of-experience that applications can provide.

In this thesis, I focus on the services and abstractions that the transport layer provides to applications. The delivery model provided by the transport layer can have a significant impact on the quality-of-experience that can be provided by the application. Reliability and order, for example, introduce delay while packet loss is detected and the lost data retransmitted. This enforces a particular trade-off between latency, loss, and application quality-of-experience, with reliability taking priority. This trade-off is not suitable for low-latency multimedia applications, which prefer predictable and bounded latency to strict reliability and order.

No widely-deployed transport protocol provides a delivery model that fully supports low-latency applications: UDP provides no reliability guarantees, while TCP enforces reliability. Implementing a protocol that *does* support these applications is difficult: ossification restricts protocols to appearing as UDP or TCP on-the-wire.

To meet both challenges – of better supporting low-latency multimedia applications, and of deploying a new protocol within an ossified transport layer – I propose TCP Hollywood, a protocol that maintains wire compatibility with TCP, while exposing the trade-off between reliability and delay such that applications can improve their quality-of-experience. I show that TCP Hollywood is deployable on the public Internet, and that it achieves its goal of improving support for low-latency multimedia applications. I conclude by evaluating the API changes that are required to support TCP Hollywood, distilling the protocol into the set of transport services that it provides.

Acknowledgements

First and foremost, my thanks go to my supervisor, Colin Perkins, whose encouragement, guidance, and patience has been much appreciated. In addition, I would like to thank Marwan Fayed and Dimitrios Pezaros for their guidance and support.

I am grateful to my examiners, Lewis Mackenzie (University of Glasgow) and Nicholas Race (University of Lancaster), for their feedback on this thesis.

I would like to acknowledge the University of Glasgow, both for providing me with a scholarship, and for additional funding for travel throughout my studies. In addition, I am grateful for travel grants that I received from the ACM.

I had the opportunity to undertake an internship at Verizon Digital Media Services during my PhD. I would like to thank Marcel Flores, Sree Priyanka Uppu, and the Research team at VDMS for this opportunity, and for making the experience enjoyable and fulfilling.

I have been fortunate to share an office with many friendly and encouraging people throughout my studies. In particular, I would like to thank Ibrahim Alghamdi, Blair Archibald, Vivian Band, Magnus Morton, Alex Pancheva, and Mihail Yanev for making the PhD experience more enjoyable than it would have been otherwise.

Finally, I am especially grateful for the support and encouragement from my family and friends.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Low-latency multimedia applications, including telephony [70], video conferencing [42], and video streaming services [9], are rapidly increasing in popularity, and the traffic that they generate now comprises the majority of Internet traffic. In this thesis, I define low-latency applications as those that have tight constraints on end-to-end latency, which when exceeded, cause the quality-of-service provided by the application to degrade. The actual bounds on latency vary, depending on the application, from tens to a few hundred milliseconds. Given the unreliable delivery and unpredictable latency of most network environments, operating within these bounds is difficult, requiring accommodations across the protocol stack.

In this thesis, I focus on the transport layer. The delivery model provided by a given transport protocol has significant implications on the latency that is introduced, and on how quality-of-experience can be managed at the application layer. Reliability and order, both features of TCP, introduce delay while loss is detected and the missing data is resent. Low-latency multimedia applications prefer predictable and bounded latency to strict reliability and order. As I will discuss in Chapter 2, this delivery model is not provided by the transport layer protocols that are widely deployed on the Internet.

In principle, it should be possible to implement a transport layer protocol that *does* provide a suitable delivery model. Notionally, the layered architecture of the Internet means that routers and other network infrastructure components are concerned with the IP layer, and are agnostic to the transport protocol used. In practice, however, many middleboxes in the network inspect transport headers and payloads. This has resulted in *ossification* around UDP and TCP, requiring their use as substrates in the development of new protocols. I further explore the implications of ossification in Chapter 3.

With UDP providing few additional features beyond those of IP, it is the obvious base upon which to develop new protocols. However, UDP lacks connection-oriented primitives, which combined with its lack of flow and congestion control mechanisms, make it an attack vector. As a result, some UDP-based services are abused, causing it to be blocked by a non-trivial

number of enterprise firewalls. Therefore, it is worthwhile to explore a TCP-based solution: in Chapter 4, I detail the design of TCP Hollywood, a protocol that maintains wire compatibility with TCP, while supporting the requirements of low-latency multimedia applications.

TCP Hollywood changes TCP's ordered, strictly reliable byte-stream abstraction into an unordered, partially reliable message-oriented delivery model, allowing applications to control the trade-off between reliability and latency, improving the quality-of-experience that they can provide. The mechanism that provides partial reliability – *inconsistent retransmissions* – fundamentally changes the semantics of TCP on-the-wire. Given that this change is visible to middleboxes on the path, it may impact TCP Hollywood's deployability. In Chapter 5, I evaluate both the performance and deployability of TCP Hollywood.

While largely unsuitable for latency-sensitive applications, TCP has become the de-facto protocol for video delivery, with many applications using HTTP adaptive streaming protocols such as MPEG-DASH. TCP Hollywood and HTTP/2 allow HTTP adaptive streaming protocols to migrate from their pull-based architectures, which, combined with TCP's in-order, reliable delivery model, introduce significant delay. In Chapter 6, I evaluate the performance of a simulated, modified MPEG-DASH application. This demonstrates the performance benefits that TCP Hollywood can provide, and highlights the ease with which applications that use TCP can be migrated to TCP Hollywood without loss of reachability.

With TCP Hollywood demonstrating that TCP can be used as a substrate in the development of new protocols, and the same evidence for UDP provided by other protocols (for example, in the development of QUIC), it is worthwhile to consider how the transport layer API might be reshaped more broadly. In particular, given the opportunities for domain-specific protocols that a substrate-based approach allows, detaching the delivery model from the protocol that provides it would allow for flexibility, and ease application development. To further explore this, in Chapter 7, I adopt the terminology used by the IETF's TAPS working group to formalise a description of the *transport services* required by low-latency multimedia applications.

1.1 Thesis Statement

I assert that low-latency multimedia applications are poorly served by the existing, widely deployed transport protocols, and that the performance of these applications can be improved with appropriate transport layer support. To test this hypothesis, I will design, implement, and empirically evaluate a protocol that allows applications to define an appropriate trade-off between reliability and delay. Exposing this trade-off leads to improvements in performance in low-latency multimedia applications, where sufficient delay has the same impact on quality-of-experience as loss.

1.2 Contributions

The contributions of this thesis are:

- A description of the transport layer requirements of low-latency multimedia applications;
- TCP Hollywood, a TCP-based transport protocol that accommodates low-latency multimedia applications;
- An analysis of TCP Hollywood that explores its lower latency delivery model;
- An empirical evaluation of TCP Hollywood's performance and deployability;
- An evaluation of a simulated, modified MPEG-DASH application that uses TCP Hollywood, further evaluating its performance benefits, and better characterising its delivery model; and
- A formal description of the domain-specific *transport services* required by low-latency multimedia applications.

1.3 Publications

The work described in this thesis has been published in the following papers:

- **"Reinterpreting the Transport Protocol Stack to Embrace Ossification"**. Stephen McQuistin and Colin Perkins. In *Proceedings of the IAB Workshop on Stack Evolution in a Middlebox Internet*, January 2015.
- **"TCP Goes to Hollywood"**. Stephen McQuistin, Colin Perkins, and Marwan Fayed. In *Proceedings of the 26th ACM SIGMM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2016)*, May 2016.
- **"TCP Hollywood: An Unordered, Time-Lined, TCP for Networked Multimedia Applications"**. Stephen McQuistin, Colin Perkins, and Marwan Fayed. In *Proceedings of the IFIP Networking Conference*, May 2016.
- **"Implementing Real-Time Transport Services Over an Ossified Network"**. Stephen McQuistin, Colin Perkins, and Marwan Fayed. In *Proceedings of the ACM, IRTF and ISOC Applied Networking Research Workshop*, July 2016.
- **"DASHing Towards Hollywood"**. Saba Ahsan, Stephen McQuistin, Colin Perkins, and Jörg Ott. In *Proceedings of the ACM Multimedia Systems Conference*, June 2018.

1.4 Outline

The remainder of this thesis is structured as follows:

Chapter 2 discusses the status quo of multimedia delivery over the Internet, describing the end-to-end delay that packets experience, and presenting two case studies (RTP and MPEG-DASH) that show the applications that result from two fundamentally different design goals;

Chapter 3 studies the evolution of the transport layer, and the impact that ossification has in limiting and shaping novel transport protocols;

Chapter 4 describes the design and architecture of TCP Hollywood, a TCP-based transport protocol for low-latency applications;

Chapter 5 gives a preliminary evaluation of the performance impact of TCP Hollywood, and discusses and evaluates the impact on deployability of its changes vs. standard TCP;

Chapter 6 studies how applications must rearchitected to adopt TCP Hollywood, and the performance benefits that are available, within the context of a simulated, modified MPEG-DASH application;

Chapter 7 sets TCP Hollywood, and the transport services it provides, within a broader rethinking of the transport layer API; and

Chapter 8 discusses potential directions for future work, and concludes.

Chapter 2

Multimedia Delivery in the Internet

Multimedia applications can be split into two broad categories based on the time at which the audio or video is captured, relative to when it is sent, and the latency constraints on playout at the receiver. In live applications, the multimedia (e.g., a live sports match) is captured, encoded, and transmitted immediately in real-time. In on-demand applications, the multimedia (e.g., a pre-recorded TV show) is captured, and then encoded and transmitted at a later time, possibly at a rate faster than real-time. At the receiver, the latency requirements of the application determine the size of the playout buffer: smoother playback is achieved with large buffers, while timeliness requires small, sub-second buffers. The focus of this thesis is on live *and* on-demand applications that require low-latency playback: for example, a stream of a live sports match, or an interactive component of an on-demand stream (e.g., to support viewpoint selection).

There are a number of processes that introduce delay in multimedia applications, including encoding, packetisation, and decoding. Further, when multimedia applications transmit their data across the Internet, they are exposed to the unpredictable nature of a best-effort, packet-switched network. This introduces variable levels of packet delay, re-ordering, and loss, the management and concealment of which the application must balance with its latency constraints. In Section 2.1, I detail the end-to-end delay introduced in networked multimedia applications.

Latency and reliability can be concealed at either the application or the transport layers: in this chapter, case studies are provided for two widely-deployed protocols that take alternative approaches. In Section 2.2, I discuss the Real-Time Transport Protocol (RTP), whose design is based on the principle that the application alone has sufficient information to make optimal choices about the trade-off between latency and reliability. By comparison, in Section 2.3, I describe MPEG-DASH, a widely used protocol that operates over HTTP and TCP, which provides a strict in-order, reliable delivery abstraction at the transport layer, introducing delay that the application cannot control.

This chapter is structured as follows:

Section 2.1 gives an overview of the end-to-end delay experienced by multimedia applications running across the Internet;

Section 2.2 provides a case study looking at the design, implementation, and deployment of RTP, a protocol designed to accommodate the timeliness vs. reliability trade-off underpinning real-time multimedia applications;

Section 2.3 discusses MPEG-DASH, describing how its architecture, being based upon HTTP and TCP, introduces significant latency, making it a poor choice for low-latency applications; and

Section 2.4 summarises the chapter.

2.1 Delay in Networked Multimedia Applications

Sources of delay in networked multimedia applications can be split into two broad categories: the fundamental processes of the application, and the impact of running these applications over the Internet. Figure 2.1 illustrates the sender-side of a typical multimedia application. The sender is responsible for capturing, encoding, and packetising the multimedia. These packets are then sent across the Internet, through the transport layer. When the packet is in the Internet, it is subject to serialisation, propagation, and queueing delays at each hop, as illustrated in Figure 2.2. Once the packet arrives at the receiver, shown in Figure 2.3, it is passed through the transport layer, and placed into a de-jitter buffer. Once the packet is at the head of this buffer, it is then decoded and played out.

In this section, I set aside the delays that might be introduced by the transport layer, and focus on those – outlined above – that are introduced by either the application or as data traverses the Internet. I describe each in turn in the sections that follow.

2.1.1 Encoding Delay

After capturing the multimedia, the sender must encode and compress it. The time taken to encode the media is dependent on the encoding scheme that is used. For frame-based encoding schemes, the encoding delay is the sum of the frame size, the look-ahead or algorithmic delay, and the processing delay. For voice-over-IP (VoIP) applications, encoding delay ranges from around 15ms using G.726 [37] (with 10ms frames, no look-ahead delay, and low processing delay) to around 77.5ms with G.723.1 [39] (with 30ms frames, 7.5ms look-ahead delay, and less than 30ms processing delay [46]).

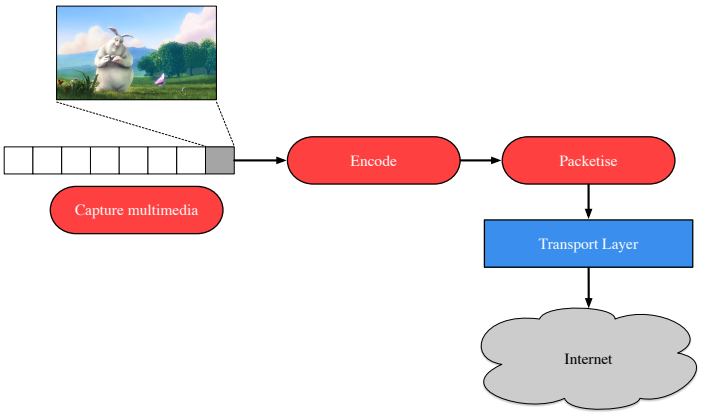


Figure 2.1: Sources of delay at the sender-side of multimedia applications

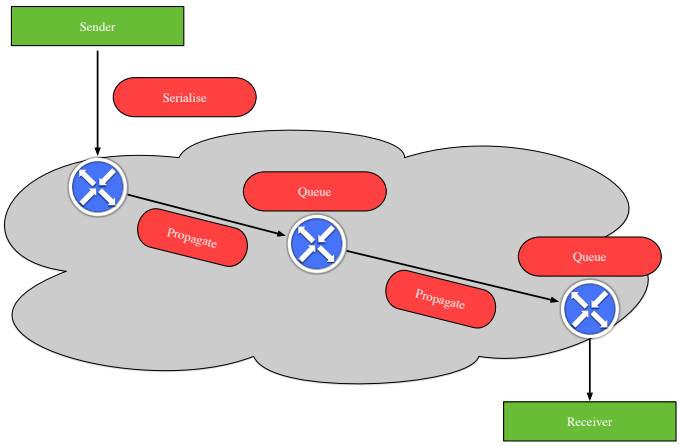


Figure 2.2: Sources of delay within the Internet

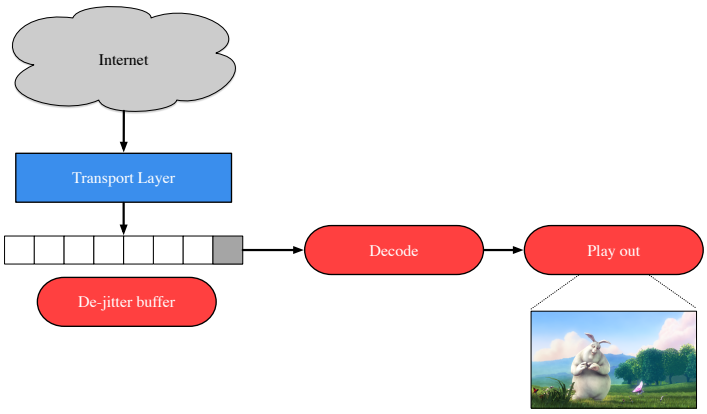


Figure 2.3: Sources of delay at the receiver-side of multimedia applications

2.1.2 Packetisation Delay

Packetisation delay is the time taken to fill a packet with encoded media. The size of a packet is dictated by the application, and the underlying protocols at the transport and network layers. For example, VoIP applications typically send packets with 20ms payloads [72]: this is the packetisation delay.

Encoding and packetisation can usually be pipelined, such that the overall delay that these processes introduce is less than their sum. As one frame is being packetised, the next frame in the packet is being encoded.

2.1.3 Serialisation Delay

Serialisation delay is the time taken to send the packet on-the-wire. This is determined by the size of the packet, and the bandwidth of the link it is being transmitted over:

$$\text{serialisation delay} = \frac{\text{packet size}}{\text{link rate}} \tag{2.1}$$

Larger packets have a higher serialisation delay over a given link, while links with higher bandwidth will result in lower serialisation delays for a given packet size. While high bandwidth links have rendered serialisation delay largely negligible, especially when compared to other sources of delay, it can be significant for large packets and slower access links. For example, a 1500 byte packet on a 10Mbps link will experience a serialisation delay of 1.2ms, while a 9000 byte datagram – the largest allowed in an Ethernet jumbo frame – would experience a delay of 7.2ms.

While serialisation delay does not account for the length of the link, it is important to note that it is experienced for each hop – that is, per middlebox (e.g., router or switch) – that the packet passes through. As noted above, this is likely to be negligible in the core of the network, where links have sufficient bandwidth. Serialisation delay is more significant across slower access links.

2.1.4 Propagation Delay

The propagation delay for each packet is the time taken for the bits to travel between the sender and the receiver. This is determined by the length (i.e., physical distance) of the link, and the speed at which the bits travel. This varies by the type of physical link: fiber optic links, for example, have lower propagation delays (over the same distance) than copper wire links. The end-to-end propagation delay for a packet is the sum of the per-hop propagation

delays as the packet passes through the network. Propagation delay, at around 4-6 nanoseconds per metre (depending on the type of physical link), is typically small in intracontinental links, but may be significant for long-distance, intercontinental links.

2.1.5 Queueing and Processing Delay

Given both serialisation and propagation delays, for routers to fully utilise a link, they must have a buffer in which to store packets that need to wait before being transmitted. The delay introduced to the packet by this buffer is the queueing delay; the end-to-end queueing delay of a packet is the sum of all the queueing delays it encounters on its path through the network.

Queueing delay is bounded by the size of the router's buffer. As a result, the sizing of the buffer is an important consideration in the end-to-end delay of a path. If a router's buffer is too large, and its upstream link is a bottleneck, then high, variable latency may result from *bufferbloat* [25]. Bufferbloat results in high delay because packets spend longer in the buffer at the router. Variability is likely to result from the interaction between the large buffer, and the congestion and flow control algorithms at the transport layer. For example, TCP is designed to fill the buffer of the bottleneck link, and use packet losses (that result from drops of packets that do not fit in the buffer) as a signal to moderate its sending rate. Once packet loss occurs, latency is already increased due to the buffer; as a result, the sender will slow down, and latency will also drop. This has a significant impact on the performance of real-time applications, which require both low and predictable latency. Queueing delay is the most significant component of end-to-end latency, with some queues adding tens to hundreds of milliseconds of delay [16].

Active queue management techniques [58, 47] have been proposed as a way of reducing the latency associated with bufferbloat. Instead of drop-tail queueing, routers drop or mark packets probabilistically *before* the buffer fills, allowing for earlier congestion avoidance.

In addition to queueing delay, there may be a processing delay for each packet. This is the time taken to process the IP header of each packet, and perform basic checksum validation. This process is typically performed at line rate: that is, it is masked by the serialisation delay described above. However, some middleboxes perform deep packet inspection, requiring per-packet or per-flow processing beyond the basic header processing. While this is typically negligible, it may contribute to overall delay.

The end-to-end latency observed above the IP layer, incorporating serialisation, propagation, and queueing and processing delays, within the UK, is around 15 milliseconds (for high-speed, fibre broadband) to 25 milliseconds (for ADSL) [60]. However, latency can reach the hundreds of milliseconds, with some providers, and over longer, intercontinental links [16].

2.1.6 De-jitter Buffering

To reduce the impact of the variable delays that are introduced as the packet traverses the network, it is placed into a de-jitter buffer at the receiver. The application decodes and plays out the packet at the head of the de-jitter buffer at regular intervals. Sizing the de-jitter buffer correctly requires care: too much buffering inflates end-to-end delay, while too little will cause sporadic gaps in playback.

Given that the purpose of the de-jitter buffer is to absorb the impact of delay variation, it is optimally sized at the total end-to-end delay variation. However, in the Internet, this value is not known ahead of time.

2.1.7 Summary

As this section has illustrated, there are a number of areas where delay and loss can occur as the application captures and encodes the media into packets, as those packets traverse the Internet, and as they are buffered for decoding and playback at the receiver. The sources of delay discussed in this section are fundamental to networked multimedia applications, and are not the focus of this thesis.

This thesis is concerned with the impact of packet loss and reordering in the network, and how this is handled by the application or transport layers, such that the impact on quality-of-experience is minimised. Given that the one-way delay bound for typical low-latency applications ranges from 150 milliseconds for voice over IP [38], to around 500ms for changing live TV streams [40], to a few seconds for streaming video, accommodations for these applications are necessary across the protocol stack.

The combination of potentially high delay at the network layer, and the relatively low bounds imposed by latency-sensitive applications, means that care must be taken to minimise the latency introduced by the transport and application layers. In particular, it is important to consider the inherent trade-off between reliability and delay. For many applications, it is desirable for loss to be concealed, even at the expense of delay: for example, when downloading a large file, it is necessary that all of the bits arrive, even if the download takes more time. However, for the low-latency multimedia applications that this thesis is concerned with, sufficient delay is equivalent to a loss. In a live video stream, for example, there is a playout time for each frame; if the frame does not arrive by its playout time, then it cannot be played out – it is effectively lost.

An important consideration in this discussion is where in the protocol stack packet loss is concealed. In the sections that follow, two alternative approaches will be described: in RTP (Section 2.2), the application is responsible for interpreting and correcting loss, while

MPEG-DASH (Section 2.3) relies upon TCP's strictly reliable delivery model. These design principles have significant implications on end-to-end delay, and therefore, the classes of applications that can use each protocol.

2.2 Case Study: RTP

The Real-Time Transport Protocol (RTP) [73] and its related control protocol, the RTP Control Protocol (RTCP), are designed to enable the robust delivery of real-time multimedia over the Internet. While many of the services that RTP provides are out-of-scope of the transport layer focus of this thesis, the protocol provides an important case study when considering the boundary between the application and transport layers.

Section 2.2.1 discusses the implications of RTP's design principles, and in particular, its adoption of the *application-level framing* architectural principle. In Sections 2.2.2 and 2.2.3, I describe the architecture of RTP senders and receivers respectively. Finally, in Section 2.2.4, I summarise this section, and discuss the implications of using UDP at the transport layer as is typical, though not mandatory, for RTP applications.

2.2.1 Design Principles

A central design principle of RTP is the application-level framing architecture described by Clark and Tennenhouse [13]. This concept was motivated by how loss and order are handled in TCP, which masks losses using retransmissions and strict ordering. Clark and Tennenhouse argue that this is sub-optimal for two reasons: firstly, TCP does not expose that loss has occured, and secondly, even if it did, TCP's byte-stream abstraction means that its segments typically have no relation to the application data units (ADUs) that are meaningful to the application. As a result, Clark and Tennenhouse propose an application-level framing architecture, where the application and transport layers become closer, with ADU boundaries respected across the layers.

In addition, there are other fundamental constraints on application data units. ADUs must be able to be processed out-of-order. This means that each ADU should be independently useful to the receiving application. For example, in a video application, a single frame is an ADU if it can be independently decoded. It follows that the length of ADUs becomes an important consideration: if some portion of an ADU is lost, then the entire ADU cannot be processed by the receiving application. This places an upper bound on ADU size; if an ADU is too large, then the probability that some portion of it will be lost increases. Finally, the application-level framing principle means that the application must be responsible for retransmitting lost ADUs.

In embodying the application-level framing architecture, RTP and RTCP are designed to provide *applications* with the information about loss and delay that is required to make choices about the data that is sent, rather than relying on the transport protocol to conceal losses. As a result, these protocols function best over the best-effort, datagram delivery service that IP provides: most RTP applications use UDP at the transport layer. Importantly, the protocols are not prescriptive about what applications should do in response to loss or delay; the central tenet of the application-level framing principle is that this should be decided by the application.

The flexibility that results from application-level framing is also present in other elements of the protocols. RTP and RTCP are designed as a framework for real-time multimedia delivery, and the protocols are agnostic about the particular application use case. For example, RTP and RTCP are useful for both unicast voice-over-IP applications, and broadcast IPTV applications. This flexibility is achieved using *profiles*, which take the framework provided by RTP and RTCP, and produce a complete protocol for a specific use case.

RTP and RTCP are designed to support bidirectional, multiparty applications. As a result, a given host is typically both a sender and a receiver. RTP is responsible for carrying the multimedia data, while RTCP is used for session management and quality-of-service (i.e., generating periodic reports that contain statistics about loss and delay).

2.2.2 Sender Architecture

The first responsibility of an RTP sender is to capture and encode the raw multimedia data to be sent, as described in Section 2.1. This process typically generates frames of data that are not independent of each other. For example, in MPEG-2 [36] there are three encoded frame types: I-frames (that are independently decodeable), P-frames (that rely on data from earlier frames), and B-frames (that rely on data from both earlier and later frames). These interdependencies may be important to the transmission order of the frames.

Next, the RTP sender will packetise the encoded frames into RTP packets. RTP packets are the application data units, as discussed earlier: they can only be processed by the application once they have been received in full. Therefore, while multiple encoded frames can be sent in a single RTP packet, the overall size of the RTP packet should be considered [31]. The RTP header, amongst other fields, carries a sequence number, timestamp, and source identifier; these are used by RTCP to detect and report loss and delay. Depending on the particular profile used, RTP packets may be reordered before they are sent: frame interdependencies create a decoding order that differs from the order in which the frames were captured and encoded. Once the frames have been sent, the RTP profile may require that they be buffered to allow for retransmission later.

Under RTCP, the sender is responsible for periodically generating sender reports. These reports contain timestamp synchronisation data and packet transmission statistics. These reports are important for controlling jitter (i.e., packet delay variation) and for allowing the receiver to detect and report loss.

2.2.3 Receiver Architecture

An RTP receiver first buffers RTP packets as they arrive from the network. As they are buffered, packets are reordered based on their timestamp. The purpose of this buffering is to limit the impact of jitter: while packet interarrival times may vary due to the delays introduced by the network, playback can remain smooth (and introduce minimal latency) if the playout buffer is sized correctly.

Each RTP packet contains a playout time. When this is reached, the data that the packet contains is decoded, possibly with other data from other packets. Finally, the decoded multimedia is played out. As with RTP senders, much of the behaviour of the RTP receiver is determined by the particular profile and payload format used.

Under RTCP, the receiver is responsible for periodically generating receiver reports. These reports contain packet loss and delay statistics, as well as data to allow for timestamp synchronisation. Synchronising timestamps, and correcting for clock skew, is an important function of the receiver. Given that the sender and receiver use independent clocks, these are likely to drift over time: correcting this is important for maintaining smooth playback.

2.2.4 Summary

The architecture of both RTP senders and receivers demonstrates the flexibility required by the application-level framing principle. Given that the application is responsible for making decisions about the data that is sent, RTP and RTCP themselves cannot be prescriptive about how loss and delay are handled. However, these decisions do need to be made: loss and delay are inevitable over packet-switched networks, and how they are handled is critical to the quality-of-service provided by the application. As a result, while RTP and RTCP provide a framework for real-time applications, they are not complete without a profile and payload format specification that describes the unspecified components of the protocols.

By pushing decisions about how and when data is sent up to the application layer, the application-level framing principle means that more prescriptive protocols, such as TCP, are largely incompatible with RTP and RTCP. TCP's delivery model, in masking loss, takes control away from the application layer. This means that RTP applications typically use UDP

(there are other protocols, like SCTP and DCCP, that also respect the application-level framing principle, but they do not see wide deployment). This has two broad implications. Firstly, the complexity of loss detection and correction, as well as connection management and flow and congestion control, are pushed to the application layer, where they are less reuseable. Secondly, UDP sees less deployability (vs. TCP) on the Internet: its lack of connection-oriented primitives makes it an attack vector that is often blocked by enterprise firewalls. As a result, UDP streams are often blocked or heavily rate limited. Initial measurements [78] of the reachability of the QUIC [41] protocol, which is carried over UDP, shows that 5-10% of connections fallback to TCP because UDP is unusable. This is likely to be a best-case, conservative estimate of UDP blocking: it is likely to be higher for peer-to-peer applications.

In summary, RTP provides application developers with a trade-off: they are provided with the tools to control latency and delay, but at the expense of complexity and deployability. If deployability needs to be maximised, then the design of MPEG-DASH, discussed in the next section, shows that this requires using TCP at the transport layer. This introduces significantly more delay, as the application loses control over how delay and loss are managed.

2.3 Case Study: MPEG-DASH

Dynamic Adaptive Streaming over HTTP (MPEG-DASH) [76] is an HTTP adaptive streaming protocol, designed for the delivery of video across the existing HTTP infrastructure. In targeting widespread deployability, and therefore choosing HTTP and TCP, MPEG-DASH makes for an interesting contrast to the design goals of RTP.

Section 2.3.1 describes the design principles of MPEG-DASH, while Sections 2.3.2 and 2.3.3 detail the roles played by MPEG-DASH senders and receivers respectively. Section 2.3.4 summarises the chapter by discussing the implications of MPEG-DASH's use of TCP.

2.3.1 Design Principles

The central design goal of MPEG-DASH is to maximise the use of the existing HTTP infrastructure, such as content distribution networks and HTTP caches. HTTP, and TCP which underpins it, simplifies middlebox (such as NATs and firewalls) traversal, making MPEG-DASH easy to deploy at scale. Much follows from this desire to reuse the existing infrastructure: MPEG-DASH is designed such that most of the application logic is performed at the client, allowing for regular, commodity HTTP servers to be used.

As discussed in the previous section, in using TCP the application cedes control over how loss and delay are handled. TCP provides a strictly reliable, in-order bytestream abstraction. When a TCP segment is lost, the TCP sender uses acknowledgements from the receiver to

determine which segment was lost, and then retransmits it. This takes place without any input from the application: indeed, aside from the increased latency, segment loss is not apparent at the application layer. However, as was the motivation for the application-level framing principle, these retransmissions may not be beneficial to the overall quality-of-service of the application: latency is introduced retransmitting data that might not be useful to the application.

With the increased latency introduced by TCP, MPEG-DASH is largely unsuitable for low-latency applications. Given that segment loss introduces at least an additional round-trip time of latency (to tell the sender that loss occurred, and for the retransmission to arrive at the receiver), meeting the latency bounds of interactive applications (such as voice-over-IP, with its one-way delay budget of around 150ms) is not possible if the end-to-end delay is sufficient. Further, loss, and the latency it introduces, is unpredictable, requiring a significantly larger playout buffer than needed under RTP over UDP.

MPEG-DASH uses a chunk-based architecture, splitting multimedia into chunks that have equal durations. While much of the latency in MPEG-DASH applications comes from TCP and the transport layer, how the application fetches chunks using HTTP also introduces latency, imposing a high lower bound on the size of the playout buffer. I describe the architecture of both the sender and receiver in the sections that follow.

2.3.2 Sender Architecture

MPEG-DASH senders begin by capturing and encoding the multimedia in much the same way as RTP senders do, and MPEG-DASH is similarly agnostic about the particular codec that is used to encode media frames. However, MPEG-DASH encodes the multimedia into chunks, or groups of pictures, that are independently decodable. These chunks are typically multiple seconds in duration, and all chunks within a particular stream have the same duration. Given that each chunk is independently decodable, they are typically made available at multiple different bitrates. This allows for chunks to be requested at a bitrate that is suitable for the available bandwidth.

Once the multimedia has been encoded into chunks – possibly at different rates – a manifest file is generated. The manifest, amongst other metadata, contains information about the bitrates that chunks are available in, and the URLs of the chunks. The MPEG-DASH sender then makes the manifest file, and all of the chunks, available over HTTP.

An important principle of the MPEG-DASH architecture is that senders are “dumb” HTTP servers: they are responsible only for making the manifest and chunks available. All decision making is pushed to the receiver. This means that the existing HTTP infrastructure can be reused without modification.

2.3.3 Receiver Architecture

MPEG-DASH receivers first request the manifest file; this provides them with a listing of the available bitrates, and the URLs of the chunks. Each chunk is requested in turn, and decoded and played out. As with RTP receivers, MPEG-DASH receivers also have a playout buffer. This playout buffer must be at least one chunk in duration if the chunk is to be fully downloaded before being played out. However, the playout buffer will typically be larger to accommodate jitter.

This means that the duration of each chunk is an important factor in overall performance: if, for example, chunks are 5 seconds in duration, then the playout buffer must be at least as long. Intuitively, reducing chunk durations may be a first step in reducing the overall latency introduced in MPEG-DASH applications. Indeed, this is the basis of recent attempts to improve MPEG-DASH's support for low-latency applications. MPEG's Common Media Application Format (CMAF) [34], for example, allows for sub-chunk application data units: rather than requiring an entire chunk (which can be in the order of seconds) be encoded and decoded as one (incurring significant latency) smaller groups of frames can be delivered, reducing latency [19, 3]. However, the pull-based architecture of MPEG-DASH – where an HTTP request is sent for every chunk – presents a trade-off: if chunks are too small, then the network will become overloaded with requests. I will further discuss approaches to reducing latency in MPEG-DASH in Chapter 6.

Given that the MPEG-DASH sender does not contain any logic about playback or rate adaptation, this must be handled by the receiver. The receiver requests each chunk when it estimates that it can hold it within its playout buffer. Additionally, the receiver requests each chunk at a suitable bitrate, using estimates of the bandwidth between it and the server, statistics about its playout buffer occupancy, and other metrics [49].

2.3.4 Summary

The design of MPEG-DASH provides an interesting contrast to that of RTP. The goal of maximising reachability and ease-of-deployment essentially means that TCP must be used at the transport layer. However, as discussed in the previous section, this conflicts with the application-level framing principle. In the case of MPEG-DASH, decisions about delay and loss are made without information from the application. This limits MPEG-DASH's applicability for low-latency applications.

2.4 Summary

The two case studies presented in this chapter expose the trade-off that applications are presented with. RTP over UDP allows for latency requirements to be met, but is unavailable on a non-trivial number of paths. MPEG-DASH over HTTP and TCP maximises deployability, but at the expense of not being useful for latency sensitive applications. The goal, then, is to develop a protocol that allows for the application-level framing principle to be held (i.e., the application needs to be able to control retransmissions), while maintaining the deployability of TCP.

In principle, the layered architecture of the Internet should make this straightforward: as long as the interface with the network layer is unchanged, then changes at the transport layer should be deployable. However, in practice, this is not the case. I will explore the impact that ossification has on protocol development in the next chapter.

Chapter 3

Innovating within an Ossified Transport Layer

In the previous section, I illustrated the trade-off that the existing, widely deployed transport protocols present to application developers. UDP provides flexibility, but at the cost of increased complexity and reduced reachability, while TCP maximises reachability, but removes the application's control over the response to loss and delay. For the low-latency applications that this thesis focuses on, it is desirable to control latency at the expense of delay *and* to maximise reachability. Deploying a new transport protocol that provides this delivery model is theoretically possible: given the layered architecture of the Internet, swapping out the transport layer should be possible so long as the network layer remains unchanged.

However, in practice, there are middleboxes in the network that look beyond IP headers, and enforce behaviour based upon the headers and payloads of both transport and application packets. This effectively prevents the deployment of entirely new protocols: the rational security response to unfamiliar protocols is to block them, ossifying [29, 62] the transport layer around those protocols that are already widely deployed. This limits new protocols to looking like UDP or TCP on-the-wire, but does not prevent transport layer innovation. In this chapter, I will describe how TCP and UDP can be used as substrates for protocols that have reachability *and* support for low-latency multimedia applications.

This chapter is structured as follows:

Section 3.1 describes how the evolution of the Internet – from an academic research project into a critical infrastructure component – has resulted in ossification;

Section 3.2 discusses how the two widely deployed transport protocols – UDP and TCP – can be used as substrates in the design of novel protocols; and

Section 3.3 summarises the chapter.

3.1 Ossification

As highlighted in Chapter 2, UDP and TCP present application developers with two different delivery models, with UDP providing flexibility, but with limited additional functionality beyond that of IP, and TCP giving a strictly reliable, ordered bytestream abstraction. These two delivery models serve the needs of different classes of application, but they do not serve the needs of *all* applications, including the low-latency multimedia applications that are the focus of this thesis.

Recent efforts to standardise novel transport protocols include SCTP [75] and DCCP [48]. SCTP combines the delivery models of UDP and TCP, providing a reliable, ordered, and congestion-controlled abstraction. DCCP is similarly congestion-controlled and message-oriented, but is not ordered or reliable. These protocols embody the spirit of the layered architecture, and the purpose of the transport layer: they support novel delivery models, and do so below the application layer, reducing complexity and enforcing network stability. In theory, these protocols should be deployable on the Internet, with IP still being used at the network layer. The end-to-end principle [71] should apply to protocols at the transport layer and above: while routers should inspect the source addresses of packets to perform ingress filtering [20], and use the destination addresses to route packets towards the correct destination, they should *not* inspect their contents.

However, this is not how the network operates in reality. There are security and performance benefits that can be attained by performing deep packet inspection within the network. For example, a firewall can better protect the network if it can detect anomalies within transport and application payloads. A rational security response by a middlebox that encounters traffic that is unfamiliar is to block it. As a result, protocols like SCTP and DCCP are often blocked by middleboxes, and do not see wide deployment in the Internet [32].

In terms of supporting the Internet's role as a component of critical infrastructure, ossification is largely beneficial. Careful consideration *should* be required before making changes to a network that supports services that are vital to the functioning of society [52]. Rapid change brings instability, and would have limited the adoption of the Internet in critical systems.

Ossification does not prevent innovation at the transport layer. However, it does mean that any innovation must produce protocols that look like UDP or TCP on the wire: middleboxes should not be able to differentiate between a new protocol and either UDP or TCP. This has important implications on how UDP and TCP can be used as the building blocks for new protocols. As the next section will discuss, middleboxes often interact with traffic in non-obvious and undocumented ways: maximising reachability requires understanding such behaviour.

3.2 Substrate-based Protocol Design

In this section, I discuss the relative merits of using either UDP or TCP as building blocks, or substrates, in the development of new transport protocols.

3.2.1 UDP

UDP is the obvious base upon which to build new protocols: it provides minimal services beyond those of IP, allowing it to introduce minimal latency, and resulting in it being the recommended protocol for real-time applications [65]. While middleboxes extensively profile UDP-based protocols, such as RTP and QUIC, there are few behaviours associated with UDP flows themselves that can be fingerprinted by middleboxes, and used to enforce behaviour. Further, the overhead of using UDP as a substrate, when compared to running natively over IP, is only an additional 8 byte header. DNS and RTP, as described in Section 2.2, are perhaps the most widely deployed protocols that use UDP as a substrate. Other examples include WebRTC data channels [43], which tunnel peer-to-peer SCTP associations over DTLS and UDP, and QUIC [41], which provides a modern alternative to TCP implemented over UDP.

While UDP's flexible semantics can be advantageous, they also present a limitation in its role as a substrate. UDP traffic is often blocked by enterprise firewalls, and some parts of the operations community strongly distrust UDP-based protocols and applications [8]. In part, this is due to a lack of familiarity with UDP: outside of specific uses, UDP has not been widely used in enterprise environments. As with ossification more broadly, the rational security response is therefore to block it. Further, UDP, like TCP, has often been used as an attack vector in distributed denial of service (DDoS) attacks. However, unlike TCP, UDP lacks connection-oriented primitives, making blanket banning UDP traffic more viable than the targeted blocking that might be appropriate for malicious TCP traffic.

In addition, while UDP itself presents few semantics that can be fingerprinted by middleboxes, protocols that use it as a substrate are often targeted. Middleboxes that otherwise block UDP more broadly, may allow DNS, RTP, QUIC or other UDP-based protocols to pass specifically. This poses a challenge for new UDP-based protocols: their reachability is likely to be limited until they reach a sufficient critical mass, after which their reachability is likely to improve. Google reports that 90-95% of endpoints use QUIC running over UDP [78], while Edeline et al. [18] report that up to 3.66% of RIPE Atlas probes have no UDP connectivity. It is not clear that either dataset is wholly representative – Google's study only used Chrome and connections from its well-connected servers, and Edeline et al. used UDP traceroute – and the percentage of flows where UDP is not available is likely to be higher. Again, QUIC and UDP traceroute are specific applications, whose reachability is unlikely to be representative of novel applications. While QUIC is the most recent widely-deployed

UDP protocol, its development process has been atypical, with significant deployment in a comparatively short time, with development by Google, who control a non-trivial number of end-to-end Internet connections.

In summary, UDP reachability is likely to be notably lower than that of TCP. Where UDP is blocked or rate limited, applications are likely to fallback to TCP. For low-latency applications, the difference in performance between UDP and TCP can be significant. Brosh et al. [7] show that TCP's in-order, reliable, bytestream abstraction is detrimental to low-latency applications, when network latency and loss rates are relatively high. In addition, as seen in the architecture of MPEG-DASH discussed in Section 2.3, and in application development more broadly [66], it is often desirable to use TCP to make use of the benefits (e.g., the existing CDN infrastructure) of HTTP at the application layer. As a result, we must consider how TCP can be used as a substrate in the development of low-latency protocols.

3.2.2 TCP

TCP is more difficult to use as a substrate when compared with UDP. It is a far more sophisticated protocol, with a relatively complex header format, and an intricate state machine. Further, its state machine is well understood by middleboxes in the network [33] making it difficult to deviate from. However, this does not mean that TCP cannot evolve, or form the basis for transport protocols that provide radically different delivery models. Rather, it means that innovation and evolution must be carefully considered, with attention paid to backwards compatibility.

There are three main components of TCP that must be explored if it is to be considered as a substrate for protocols that deviate from its delivery model: its congestion control algorithm, how data is segmented, and its reliability guarantee.

Given that TCP's **congestion control algorithm** is executed only at the endpoints, it can be modified with relative freedom if no new header-level signalling is required. While CUBIC [28] – the most widely deployed TCP congestion control algorithm – follows the goal of maximising throughput at the expense of latency and jitter, this is not required by the protocol. For example, TCP Vegas [6] is a well-known delay-based approach to TCP congestion control that, while being less aggressive than CUBIC, reduces latency. FAST TCP [44] is a more modern delay-based algorithm that competes better with CUBIC in many environments, and is seeing deployment.

The development of new TCP congestion control algorithms shows that while there may be issues in developing protocols that use TCP as a substrate that are fair and compete well with standard TCP, the network does not prevent their deployment. In terms of the low-latency multimedia applications that are the target of this thesis, it would be possible to

implement alternative congestion control algorithms that seek low, predictable latency, and that are compatible with a particular multimedia codec, rather than being limited to seeking to be "TCP friendly". To do this effectively might require changes to the transport layer API. For example, video applications generate data periodically; it might help the congestion control algorithm to know the on-off period so that data sending can be paced correctly. As video traffic is less elastic than typical TCP bulk flows, it might be beneficial for the application to provide the congestion control algorithm with upper and lower rate bounds. In the other direction, the congestion control algorithm could inform the application of its RTT estimate and congestion window size, allowing it to better schedule traffic to match available capacity. Novel interactions between multimedia codecs and congestion control have been proposed for RTP over UDP [83] and could be applied to TCP. There have been a number of approaches to TCP congestion control that consider its impact on the performance of multimedia applications [12, 35, 17].

Given that the congestion control algorithm can be changed relatively easily, it is out-of-scope for the remainder of this thesis. All performance evaluations, unless stated, use the Linux implementation of the CUBIC algorithm. The changes to TCP that are proposed can be combined with existing proposals to reduce delay related to congestion control, including active queue management [58, 47], or the delay-based algorithms [6, 44] described above.

As discussed in Section 2.2, low-latency applications benefit from controlling the boundaries between their application data units: this is a departure from **how TCP segments data**. As a result, framing ADUs within TCP's byte stream is desirable. It is necessary to consider the possibility that TCP segments may be resegmented by middleboxes on the path: Honda et al. [33] found that this occured on 7% of the paths they evaluated. There are a number of mechanisms, such as COBS [11], as used in TCP Minion [59], that allow for data segmentation within TCP's bytestream and that are resilient to resegmentation.

The application-level framing principle states that application data units should be independently decodeable. As a result, beyond simply framing data units correctly, it is also advantageous to receive these out-of-order. As noted in the discussion of congestion control above, the API that is exposed to applications that use TCP is invisible to the network, and can be changed. For example, TCP Fast Open [10] has been implemented by overloading the connectionless `sendto()` call of the Berkeley Sockets API to trigger an implicit `connect()` when used on an unconnected TCP socket. Relaxing the API to enable out-of-order delivery of segments is conceptually straightforward: segments are delivered to the application in the order that they arrive, with their TCP sequence number 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()`.

While segmenting data within TCP's bytestream and delivering data out-of-order go some

way to enabling its use as a substrate in the development of low-latency protocols, TCP's strict **reliability guarantee** introduces significant latency. When a TCP segment is lost, it is retransmitted: this is a delay of at least one round-trip time, as the sender becomes aware of the loss, and sends the missing data. This retransmitted data may not arrive in time to be useful to the application, but it will be sent by TCP regardless.

The intuitive solution to this problem – to not send the retransmission at all – would create gaps in the sequence space as observed by middleboxes. Honda et al. [33] note that this may be problematic: errors occurred in up to 15% of the paths evaluated. Given that sequence space gaps are not widely deployable, the alternative is to maintain a complete sequence space by transmitting a segment with the sequence number of the missing data, but not necessarily with the same payload. This is known as an *inconsistent retransmission*. Honda et al. show that this is largely deployable, with either the original payload or the inconsistent retransmission making it through the network in the majority of paths tested. As a result, hosts should be resilient to the reception of either the original payload, or the inconsistent retransmission. While this would not be possible under TCP's bytestream abstraction, if a message-oriented abstraction is used, this can be achieved by taking care with how and when inconsistent retransmissions are used. In Chapter 4, I will detail how inconsistent retransmissions can be used such that middlebox compatibility is maximised. Further, in Chapter 5, I will detail deployability measurements that indicate that, with a careful approach, inconsistent retransmissions are deployable on all major consumer networks within the UK.

There are a number of other components of TCP that allow flexibility in its use as a substrate, or allow for its evolution as a protocol. For example, it has an options space that, ostensibly, allows for the end-to-end signalling of new protocol features. Honda et al. [33] discuss the deployability of various TCP extensibility points, including TCP's option space.

3.3 Summary

In this chapter, I have discussed the extent to which UDP and TCP can be used as substrates in the design of new protocols for low-latency multimedia applications. While UDP appears to be the natural choice for the development of low-latency protocols, given the simplicity of its delivery model, limitations in its reachability may be problematic. When UDP is not available on a given path, it is likely that applications will fallback to TCP, to maintain connectivity, as is seen in DNS [15] and QUIC [79] deployments. In many respects, this fallback may be seamless: TCP's order and reliability semantics must already be supported by UDP-based applications. However, there are two areas in which falling back to TCP may be challenging: its bytestream abstraction, and the latency introduced in providing order and reliability. Applications may be reliant upon UDP's datagram abstraction to provide data

segmentation; TCP's bytestream does not maintain message boundaries. Finally, the difference in transport-induced latency between UDP and TCP is large: TCP's order and reliability mechanisms can introduce significant delay. For the low-latency applications that are the focus of this thesis, falling back from UDP to TCP would introduce a gulf in performance that could impact the quality-of-service that the application provides. As a result, it is necessary to develop a fallback from UDP that minimises the transport layer delay, and maintains levels of reachability that are comparable to TCP.

In Chapter 7, I will describe a fundamental rethinking of the transport layer API that shifts it away from considering transport protocols as providing a monolithic service model. In this architecture, applications specify the *transport services* that they require, rather than the transport protocol that they want to use [63]. Commonality in the transport layer API may give rise to expectations around performance: at present, an unordered message delivery service has very different performance properties across UDP vs. TCP.

In the next chapter, I will describe the design of TCP Hollywood, a protocol that maintains wire compatibility with TCP (and so maintains the same levels of reachability), but, using the extensibility points described in the previous section, provides support for low-latency applications. This allows for a better fallback for UDP-based applications, and for the provision of transport services that offer better latency performance.

Chapter 4

TCP Hollywood: Design and Architecture

So far, I have discussed the end-to-end latency requirements of low-latency multimedia applications, and shown that these are not well supported by the existing widely-deployed transport protocols. In addition, I have shown that deployment of an entirely new transport protocol (i.e., with a novel wire format) that *does* support these applications is not possible: ossification limits new protocols to looking like UDP or TCP on the wire. Finally, UDP's reachability limitations, combined with the desire for a reasonable fallback protocol, mean that it is desirable to develop a protocol for low-latency applications that makes use of TCP.

In this chapter, I will describe the design and implementation of TCP Hollywood, an unordered, time-lined TCP variant designed to support low-latency applications while being widely deployable. The design of TCP Hollywood makes use of the TCP extensibility points discussed in the previous chapter, and has been architected with a view towards partial deployments where only the sender or the receiver has been updated.

This chapter is structured as follows:

Section 4.1 provides an overview of the broad architecture of TCP Hollywood, detailing the split of functionality across the stack;

Section 4.2 details the sender architecture and implementation;

Section 4.3 details the receiver architecture and implementation;

Section 4.4 discusses how TCP Hollywood's architecture supports partial deployments;

Section 4.5 outlines how TCP Hollywood maintains wire-format compatibility with TCP; and

Section 4.6 summarises the chapter.

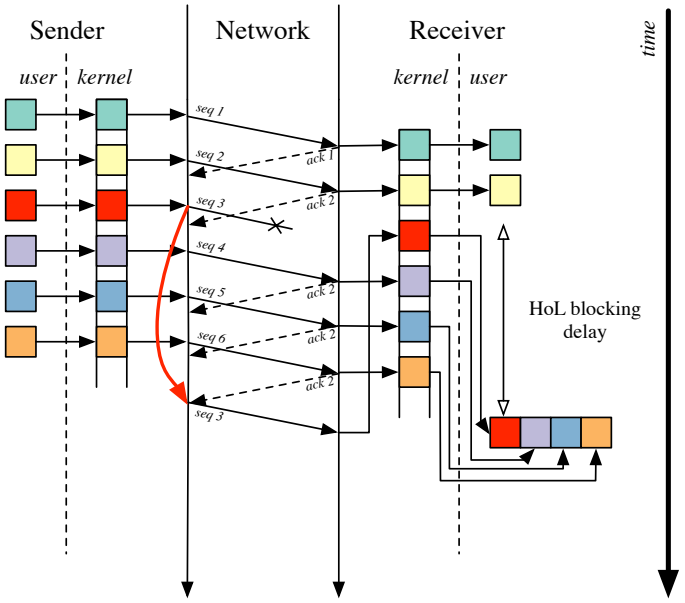


Figure 4.1: The latency impact of order and reliability in TCP: multiple segments are head-of-line blocked waiting on a retransmitted segment, and potentially delivered too late to be useful

4.1 Overview

The two primary design goals of TCP Hollywood are to: (i) improve standard TCP's performance for low-latency multimedia applications, by minimising the transport-induced latency that is passed on to applications; and (ii) maintain deployability on the scale of standard TCP. The need to consider middlebox interaction when creating new protocols, or proposing modifications to existing ones, is underscored by the successful design of Multi-Path TCP [67]. With every wire-visible modification that TCP Hollywood makes to standard TCP, it is important to consider how those changes might interact with middleboxes, and impact on deployability. As discussed in the previous chapter, TCP introduces latency in part due to the nature of its congestion control dynamics, and in part by providing an ordered, reliable delivery model, using head-of-line blocking and retransmissions. The former – TCP's congestion control dynamics – can be addressed using the active queue management or delay-based congestion control techniques outlined in Chapter 3. The latter issue, of the delay introduced by order and reliability, is more applicable for low-latency multimedia applications, and is the focus of TCP Hollywood.

Figure 4.1 illustrates the impact of the interaction between TCP's requirements for order and reliability. In the diagram, when the third segment is lost and not delivered to the receiver, subsequent segments are buffered at the receiver: they cannot be delivered to the application, given TCP's ordering requirement. This is *head-of-line blocking*: only when the lost segment has been retransmitted and received can TCP deliver the buffer of in-order segments to the

application. As shown, TCP's reliability mechanism is based upon the sender receiving three duplicate acknowledgements for the segment received before the missing segment, and then retransmitting the lost data. This introduces a delay of at least 1 round-trip time, not only to the missing segment, but to the segments that have arrived in the intervening period. For low-latency applications, where the play out buffer means that there is a small window in which data is useful to the application, this can lead to TCP delivering data that is not useful. Effectively, such segments are lost to the application, despite, in the case of those segments that were head-of-line blocked, being delivered to the host on time. It is these *late losses* that TCP Hollywood seeks to minimise.

Two requirements follow from these design goals. First, segments must be delivered as they arrive, to eliminate head-of-line blocking; to do so requires framing within TCP's bytestream, as outlined in Section 3.2, providing a message-oriented abstraction. Second, data should only be retransmitted if it is likely to be useful to the application. Enabling this requires changes to the API to allow the application to provide timing information, for this to be evaluated as data is transmitted, and by using the inconsistent retransmissions technique discussed in Section 3.2. Given that messages may not be delivered, and that interdependencies between messages may exist, it follows that these interdependencies should be exposed to the transport layer, as part of the logic that decides whether to retransmit missing data.

TCP Hollywood has been designed to support partial deployments where only either the sender or the receiver has been modified to support it. This is achieved by splitting TCP Hollywood's functionality between a user-space intermediary "shim" layer, and a set of extensions to the kernel TCP stack. The intermediary layer provides a message-oriented abstraction within TCP's bytestream, and can operate over either standard TCP, or with the TCP Hollywood kernel extensions. These kernel extensions provide out-of-order delivery of TCP segments at the receiver, and enable inconsistent retransmissions at the sender. In further discussing the design and architecture of TCP Hollywood, it is useful to consider the sender (in Section 4.2) and receiver (in Section 4.3) separately, and within each, to distinguish between the functionality provided by the intermediary layer and the kernel extensions. Finally, in Section 4.5, I will reflect upon how TCP Hollywood as a complete protocol maintains wire-format compatibility with standard TCP, achieving the goal of TCP-levels of deployability.

4.2 Sender Architecture

Figure 4.2 shows the architecture of a TCP Hollywood sender. The sender is responsible for supporting a message-oriented abstraction, and, by sending inconsistent retransmissions, time-lined messaging.

The message-oriented abstraction is provided by the intermediary layer. It accepts a se-

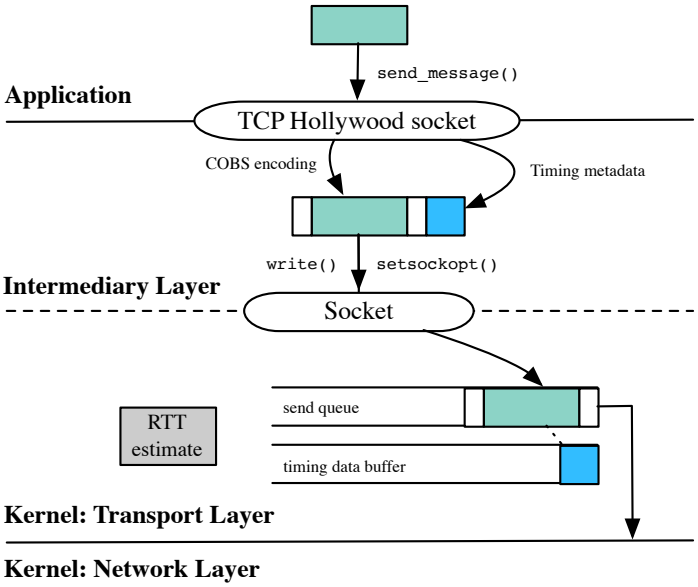


Figure 4.2: TCP Hollywood sender architecture

quence of messages (i.e., datagrams, rather than TCP's bytestream) from the application to be delivered to the destination, alongside optional timeliness and dependency metadata that is used only by the sender. The intermediary layer supports a sub-stream abstraction, allowing messages from multiple flows to be multiplexed on a single transport connection. This is similar to how multiple streams can be sent within a single SCTP association [74] or QUIC connection [41]. This mechanism can be used to cleanly multiplex audio and video flows onto a single connection, or to distinguish multiple layers of a stream encoded using scalable video coding [61] using H.264/SVC, for example. The intermediary layer appends a sub-stream identifier to messages before they are encoded, framed, and passed to the kernel TCP sender, with a default sub-stream being reserved for messages where no sub-stream is specified. The application can provide timing or dependency metadata via the intermediary layer API. This metadata consists of a lifetime, specified in milliseconds, indicating how long (from when the message is queued for sending) the message is useful for, and the 32-bit identifier of a message upon which the message being sent depends. The intermediary layer API returns the identifier of the message that was queued for sending; the application is responsible for tracking these, and using them for dependency management. The timing and dependency metadata is passed to the kernel alongside the encoded message, and used to determine whether inconsistent retransmissions should be triggered – unlike the message's sub-stream identifier, it is not sent across the network.

As outlined in Section 3.2, to support a message-oriented abstraction over TCP's bytestream, TCP Hollywood messages must be resilient to re-segmentation or segment coalescing by middleboxes in the network. Message integrity must be maintained: messages received *must* be the same as those sent, and only complete messages can be delivered. This is en-

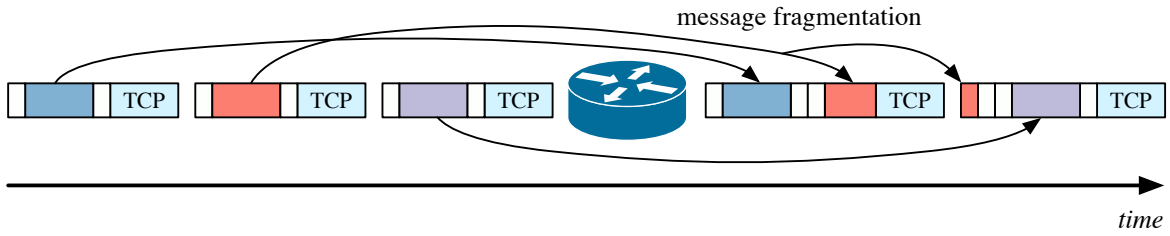


Figure 4.3: Message integrity can be maintained by encoding and framing messages with leading and trailing markers to protect against middlebox re-segmentation and coalescing

sured by the intermediary layer, which frames messages using a leading and trailing marker. Figure 4.3 illustrates how middleboxes can resegment or coalesce TCP segments, and how framing can be used to delineate messages in spite of this. As shown, given the bytestream abstraction that TCP provides, middleboxes can split up or combine TCP segments. Given that TCP segment boundaries are not necessarily static end-to-end, framing is required to maintain message boundaries. The intermediary layer encodes messages using consistent overhead byte stuffing (COBS) [11]. This efficiently encodes the message to escape all zero bytes, allowing for their use as framing markers, while still providing a transparent channel that can carry messages with any content.

The TCP sender implementation in the kernel is modified to perform consistent segmentation, and to manage the use of inconsistent retransmissions by tracking message timing, lifetime expiration, and dependencies. Consistent segmentation ensures that a single `write()` call made by the intermediary layer (and as the result of a single message being sent by the application) will generate a single TCP segment, provided that the size of the segment does not exceed the MTU. This ensures that each message is sent in a separate TCP segment, allowing the receiver to process it independently of other messages, reducing latency. This implies that Nagle's algorithm [57] is disabled (i.e., that the `TCP_NODELAY` socket option is set) to avoid unnecessary buffering. Nagle's algorithm would not provide a significant benefit to the low-latency applications that this thesis is concerned with, where messages are relatively large compared with their headers.

While TCP retransmissions ensure reliability, as discussed, they do so at the expense of latency that might cause late losses. A TCP Hollywood sender has a notion of *message expiry*. The application provides metadata for each message that includes a relative deadline for the message in milliseconds. This is the time between the message being queued, and it being required for playback at the receiver. This relative deadline is combined with an estimate of the one-way network delay, and used to determine if a retransmission will arrive before its deadline has passed. The one-way network delay is estimated as $\frac{1}{2}RTT$, where the RTT estimate is the smoothed RTT estimate maintained by TCP [64]. Using this estimate of RTT provides accuracy [77], while lessening the impact of short-lived latency spikes. The use of

$\frac{1}{2}RTT$ as a proxy for true one-way delay results from the difficulty in calculating one-way delay between unsynchronised hosts in the Internet; mechanisms to accurately determine one-way delay [14] could be used instead in environments where this is feasible.

With the metadata provided by the application, and that maintained by TCP Hollywood at the sender, it is possible to estimate if a message will arrive on time to be played out at the receiver. When the message is retransmitted, the estimate of one-way delay is used to determine if the message will arrive within its lifetime, or if it will expire beforehand. If the message is likely to expire, TCP Hollywood will send a new message using the same TCP sequence number as the previously sent message, re-writing the remaining bytes in the TCP send buffer with new, unexpired content, if available. The replacement data must be the same size or smaller than the original, and will be padded as necessary: I describe the selection of the replacement message later in this section. Importantly, this metadata is never transmitted on the wire, and is held locally for each message for as long the message is buffered (i.e., until all acknowledgements associated with the message have been received). The TCP Hollywood kernel extensions maintain a separate buffer to hold per-message metadata.

There are four instances where a message may be sent, even when the message's deadline may be missed. First, the logic to determine message liveness is triggered when the standard TCP retransmission logic would be triggered by a triple duplicate ACK or timeout. Therefore, with a sufficiently short relative deadline, it is possible for the message's lifetime to have passed before its first transmission. Second, dependency information is used to override the deadline, and send a message if there are later messages that depend on it. A message will always be retransmitted if there are unexpired messages that depend on, or if a new message could be queued that depends on it. To bound this, messages can only indicate dependencies on a single message, and in non-decreasing order: if a message expresses dependency on message with sequence number x, then the next message can only express dependency on a message with sequence number x or higher, subject to the wrap-around of the 32-bit message sequence number space.

Third, given that TCP Hollywood's message-oriented abstraction is layered on top of TCP's bytestream, TCP Hollywood messages may be comprised of multiple fragments, split across multiple TCP segments. To preserve message integrity at the receiver, fragments that complete a partially received message are always retransmitted. That is, inconsistent retransmissions are not used to replace part of a message. Only if no part of a message was received can it be replaced with a new message when its containing TCP segment is retransmitted. Finally, there may be situations where there is no suitable replacement message for the expired message. To ensure that TCP Hollywood maintains compatibility with standard TCP, the expired message will be retransmitted.

When inconsistent retransmissions *are* triggered – because the message has expired and it

has no live dependencies – an important consideration in the performance of the mechanism is the choice of replacement message. TCP Hollywood considers all complete messages (i.e., excluding those that have been partially acknowledged) in the sending queue as candidates to replace the expired message. Two properties of the candidate replacement are considered: whether or not it has expired, and how close it is to expiry. Candidate replacements are priority ordered, with live messages that are closest to expiry given highest priority, and expired messages farthest from expiry given lowest priority.

Finally, replacements may be smaller (but not larger) than the message they are replacing. If this is the case, TCP Hollywood will insert a padding message to ensure that the inconsistent retransmission has the same size as it would under standard TCP. This is important for middlebox compatibility, as demonstrated by Honda et al. [33]. These padding messages are sent on a reserved substream, and are processed by the intermediary layer at the receiver: they are not delivered to the receiving application.

Once a message has been used to replace an expired message, it is marked such that when its original transmission time (as determined by standard TCP) is reached, it is treated as if it has expired. This reduces the likelihood that it is sent twice (i.e., as an inconsistent retransmission *and* at its original transmission time), but cannot guarantee that this will be the case. Again, if there is no suitable replacement for the message, then it will still be sent.

TCP Hollywood's timeline architecture builds on a number of protocols, and tweaks to TCP, that alone are unlikely to be deployable. TL-TCP [56] makes use of time-lined data. However, the underlying mechanism works by injecting gaps into TCP's sequence space. Liang and Cheriton [51] describe TCP-RTM, which similarly allows receivers to read data out-of-order, and sends acknowledgements for unreceived data. PRTP-ECN [26] modifies TCP's response to loss in a similar way, allowing receivers to acknowledge lost packets, while using ECN to ensure that the sender's congestion control remains responsive. As observed by Honda et al. [33], sequence space gaps cause widespread deployability problems: on around 30% of paths tested, acknowledging data that a middlebox had not seen resulted in middlebox interference. The inconsistent retransmission mechanism described in this section ensures that there are no gaps in the sequence space. Finally, Deadline-aware TCP [81] is a modified TCP specifically designed for datacenters, and for flows with soft time constraints. The approach that deadline-aware TCP takes requires ECN support in the network, alongside a modified TCP sender. Requiring ECN support effectively prevents deployment outside of datacenters.

4.2.1 Implementation

TCP Hollywood has been implemented within the Linux 3.18.34 kernel. The kernel modifications (including both sender- and receiver-side changes) comprise around 650 new or

modified lines of code, while the userspace intermediary layer is comprised of around 800 lines of C code.

At the sender, the kernel implementation maintains metadata for each message. This metadata includes the message's sequence number, length, lifetime, and sub-stream identifier. This metadata is maintained in a linked list structure, and is paired with the data already stored by the TCP implementation. As the TCP segments that comprise a message are acknowledged, the metadata associated with that message is deleted. Other metadata that is required for processing each message (i.e., estimating whether a message will arrive on time) is already maintained by the TCP implementation.

The intermediary layer exposes an API to the application that is broadly the same as that discussed in Section 7.2. Metadata is communicated between the intermediary layer and the kernel either alongside the data (e.g., appended to the data as it is written to the socket) or using socket options. As will be discussed in Section 7.2, neither of these is entirely suitable, and new API architectures should be considered.

The processing overhead of TCP Hollywood at the sender is comprised of COBS encoding at the intermediary layer and the maintenance of the kernel metadata described above. COBS encoding requires a copy of the message to be made, but this could be eliminated by performing the byte stuffing as the message is being generated, as part of the multimedia encoding, or by overlapping the copy needed by the COBS encoding with that used during encryption, if being applied. Beyond this copy, COBS has been found to be computationally cheap [11].

4.3 Receiver Architecture

Figure 4.4 illustrates the architecture of a TCP Hollywood receiver. Like the sender, it is comprised of a user-space intermediary layer, and kernel modifications to TCP's receive path. The receiver supports message-oriented delivery, and is additionally responsible for eliminating head-of-line blocking by delivering incoming messages out-of-order where necessary. Importantly, the use of inconsistent retransmissions is invisible to the receiver; this is achieved by only inconsistently retransmitting whole messages at the sender.

The kernel initially processes incoming segments as TCP would. It generates the appropriate acknowledgements (e.g., duplicate ACKs for out-of-order or lost segments), and places segments into the reassembly buffer as usual. The on-the-wire response to the receipt of each segment is *identical* to that of TCP: ACKs (and SACK blocks, or other extensions, if negotiated) are generated in exactly the same way as standard TCP. As a result, TCP Hollywood's response to congestion is unchanged vs. standard TCP.

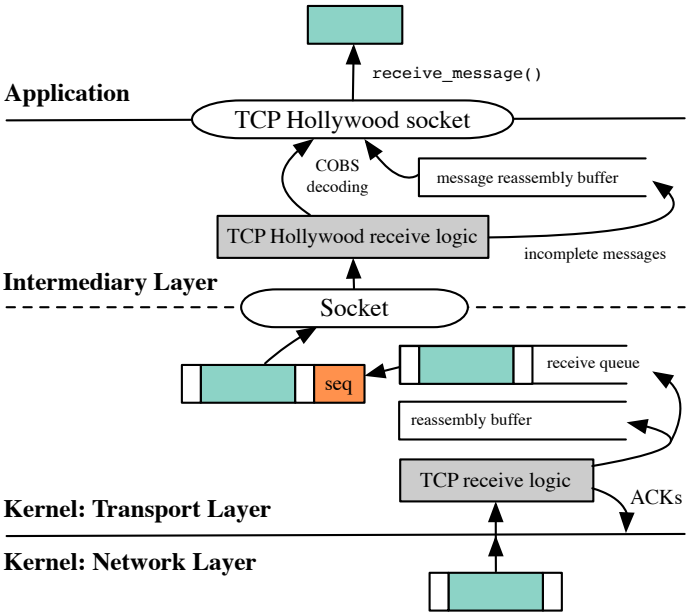


Figure 4.4: TCP Hollywood receiver architecture

Where a TCP Hollywood receiver differs from standard TCP is that all segments, including those received out-of-order, are delivered to the intermediary later in the order that they are received, with no head-of-line blocking or reordering. As each segment arrives, a metadata structure is created to store its TCP sequence number. This sequence number is then appended to the segment as it is read by the intermediary layer. Sequence numbers are used by the intermediary layer to reconstruct messages that are encoded across multiple segments. Making segments available to the intermediary layer as they arrive is the only change needed to support TCP Hollywood within TCP's kernel code at the receiver.

The intermediary layer scans incoming segments for complete messages, delineated by the COBS framing markers. If consistent segmentation was used, and segments were not fragmented or coalesced in the network, then messages will correspond to TCP segments. Otherwise, incomplete message fragments are buffered in the fragment reassembly buffer, awaiting missing fragments. The relative ordering of the bytes in message fragments is maintained using the TCP sequence number that is provided by the kernel with received segments. As shown in Figure 4.4, complete messages are decoded and queued for delivery to the application. The API between the intermediary layer and the application is message-oriented; this simplifies receiver processing, when compared to using TCP's bytestream API.

The kernel-level change to deliver incoming segments as they arrive results in multiple copies of the same segment being delivered to the intermediary layer: if a segment is out-of-order, then it will be delivered as an out-of-order segment, and then again once it becomes an in-order segment as a result of the earlier missing data being delivered. The intermediary layer passes duplicate messages to the receiving application, which must be robust to the delivery of these duplicates. This implies an architecture similar to UDP, where message delivery

semantics are best-effort.

4.3.1 Implementation

At the receiver, the kernel implementation maintains metadata for each incoming segment. This metadata includes the sequence number and length of the segment (data that would have been otherwise lost, given that it is irrelevant for the maintenance of a bytestream abstraction). For incoming segments that are out-of-order, or arrive while there are segments in TCP's reassembly queue, an additional copy (i.e., beyond any made by the standard TCP implementation) of the segment's payload is made, and stored with that segment's metadata. This additional copy is not inherently required by the design of TCP Hollywood, but is present in the prototype implementation. This greatly reduces the complexity of the code, given that the TCP implementation processes and buffers incoming segments in different, non-obvious ways, but it could be eliminated with further optimisation. Each per-segment metadata structure is maintained in a linked list. A segment's metadata is maintained while there is data from that segment that has not been read by the intermediary layer.

The COBS decoding process is similar to that of the sender, incurring an additional copy at the intermediary layer.

4.4 Partial Deployments

By design, the TCP Hollywood intermediary layer is a user-space library that can run over either a standard TCP implementation, using the Berkeley Sockets API, or on a modified TCP stack, using the extensions to the kernel that have been described. If both the sender and receiver support the kernel extensions, then the full benefits of out-of-order delivery and partial reliability, as described in the previous sections, can be realised. However, the TCP Hollywood intermediary layer can also be deployed as part of an application, irrespective of the state of deployment of the kernel TCP extensions.

If only the receiver supports the TCP Hollywood kernel extensions, with a standard TCP sender, then the intermediary layer and application will benefit from the elimination of head-of-line blocking, as provided by out-of-order delivery. However, inconsistent retransmissions will not be supported. Message-oriented delivery will be provided, given that COBS framing is implemented at the intermediary layer at both the sender and receiver. However, without support for consistent segmentation at the sender, message decoding may be less efficient: message boundaries are less likely to be aligned with segment boundaries.

If only the sender supports the TCP Hollywood kernel extensions, then it will generate inconsistent retransmissions, and perform consistent segmentation as described, as these processes

are both invisible to the TCP layer of the receiver. This will allow latency to be improved, and increase the efficiency of message decoding, at the receiver, irrespective of whether the receiver has the TCP Hollywood kernel extensions implemented.

If neither the sender or receiver support the TCP Hollywood kernel extensions, then the intermediary layer will fallback to communicating over a standard TCP connection. In this instance, the message-oriented abstraction is maintained, and the application can use the intermediary layer to exchange messages rather than using TCP's bytestream abstraction. However, messages will be delivered strictly in-order and reliably, preventing the application from realising the latency benefits that might be achieved by eliminating head-of-line blocking and relaxing TCP's reliability guarantees.

4.5 Wire-Format Compatibility with TCP

As discussed in Chapter 3, deviating too far from the wire format of standard TCP compromises deployability. Middleboxes in the network perform functions that make assumptions about the behaviour of TCP hosts, and reset connections when unexpected behaviour is observed. As an example, TL-TCP [56] provides a timelined delivery abstraction similar to that of TCP Hollywood, but does so by creating holes in TCP's sequence space. When a middlebox observes an incomplete sequence space, it assumes that the sender is misbehaving, and closes the connection. It is important to consider middlebox interaction, and to limit wire-visible modifications if deployability is a priority.

The only wire-visible difference between standard TCP and TCP Hollywood is the use of inconsistent retransmissions, where the payload of a retransmitted segment differs from that of the original. Everything else remains the same: the checksum is recalculated, and padding used to ensure that size is consistent. Therefore, TCP Hollywood is only visible to middleboxes that perform payload inspection. Broadly, there are two reasons for a middlebox to perform this inspection: to enhance performance through caching, and to improve security through anomaly detection. Split-connection TCP caches are widely deployed where round-trip times are high and non-congestive packet loss is common; in Chapter 5, I will describe observations that indicate that the majority of UK mobile providers cache TCP segments in this way. These caches deliver the original segment rather than the retransmission, removing the performance benefit of TCP Hollywood, but without disrupting the connection. Firewalls that perform deep packet inspection (DPI) are designed to detect anomalies, including protocols that behave unexpectedly. These firewalls are slow and computationally expensive at scale, and so are generally limited to enterprise networks where the trade-off is worthwhile; no such middleboxes were observed as part of the deployability evaluations of TCP Hollywood. A firewall that detects inconsistent retransmissions would likely reset the connection;

this could be detected, and used to reconnect with inconsistent retransmissions disabled.

Inconsistent retransmissions can interact negatively with middleboxes that cache and resegment TCP streams. This may result in messages being corrupted in a way that cannot be detected by TCP Hollywood alone: messages may be formed from some combination of the original message and an inconsistent retransmission, given the reuse of TCP sequence numbers. 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 [68], running over TCP Hollywood.

In summary, caching middleboxes alone result in a safe failure mode for TCP Hollywood: while the receiver will not benefit from inconsistent retransmissions, connections are undisrupted. However, other middleboxes that detect anomalies, or that resegment data in combination with caching middleboxes, may impact deployability. Mechanisms to protect against such interactions do not form part of TCP Hollywood. Deployability is considered further in Chapter 5, where the results of initial deployability measurements are discussed.

4.6 Summary

In this chapter, I have described the design and architecture of TCP Hollywood, which provides an unordered, partially reliable delivery abstraction, while maintaining wire compatibility with standard TCP. Together, these modifications, by eliminating head-of-line blocking and relaxing TCP's reliability guarantees, should reduce application layer latency. However, it remains to analyse the impact of these modifications, and to identify the network and application conditions in which they provide benefit. In Chapter 5, I will perform such analysis, and validate it using evaluations of a TCP Hollywood implementation.

Chapter 5

TCP Hollywood: Performance and Deployability

TCP Hollywood, as described in the previous chapter, reduces transport-induced latency (vs. standard TCP) through the use of inconsistent retransmissions and by eliminating head-of-line blocking. To quantify the benefits of these transport-layer changes, in this chapter I will analytically model the interaction between each component and the latency characteristics of both the network and the application. In addition, I will validate this analysis using a simulated low-latency application.

Finally, as considered in Chapter 4, the inconsistent retransmission mechanism that TCP Hollywood uses is visible to middleboxes in the network. To demonstrate that TCP Hollywood's use of this mechanism is deployable, I will present results from measurements carried out across all of the UK's major fixed-line and cellular network providers.

This chapter is structured as follows:

Section 5.1 analyses the performance implications of inconsistent retransmissions, identifying the application and network conditions where they are effective, and describes empirical evaluations that validate this analysis;

Section 5.2 extends the analysis and evaluations to include the impact of removing head-of-line blocking;

Section 5.3 describes real-world evaluations of TCP Hollywood's deployability; and

Section 5.4 summarises the chapter.

5.1 Inconsistent Retransmissions

To demonstrate when inconsistent retransmissions are useful (beyond standard TCP retransmissions), I first provide analysis in Section 5.1.1. To validate this analysis, I will perform a set of empirical evaluations, described in Section 5.1.2. Finally, I will present the results of these evaluations in Section 5.1.3.

5.1.1 Analysis

To quantify the performance benefits, by way of reduced latency, that TCP Hollywood provides, I begin by modelling one-way transport delay, T_{owd}, as:

$$T_{owd} = T_{sender} + T_{playout} + \frac{T_{rtt}}{2} \tag{5.1}$$

where T_{sender} is the time taken for the sender to capture, encode and transmit a frame of media, incorporating the encoding and packetisation delays described in Chapter 3. $T_{playout}$ is the sum of the de-jitter buffering delay, and the time taken to decode and render a frame at the receiving application. Finally, T_{rtt} is the network round-trip time. As discussed in Chapter 4, obtaining a measurement of one-way network delay between two hosts with different clocks is non-trivial. As a result, with no loss of generality, I assume broadly symmetric network delays in this analysis.

The inter-frame interval (that is, the duration of media in each frame), is denoted as $T_{framing}$. Since a frame cannot be sent before it has been captured, it follows that $T_{sender} \geq T_{framing}$. Similarly, at the receiver, if the media is to be decoded and rendered without gaps, then $T_{playout} \geq T_{framing}$. The time needed to encode and decode media is generally negligible in comparison to the framing interval, making $T_{sender} \approx T_{playout}$ a reasonable approximation in the absence of jitter. At the receiver, however, while media decoding and rendering time is comparatively small, the duration of the de-jitter buffer can be significant, and a similar approximation cannot be made.

The one-way transport delay contributes to an application's acceptable delay bound, T_{max}, such that for the application to be deployable, it is required that $T_{owd} \leq T_{max}$. For interactive voice applications, the delay bound is around 150ms [38], whereas streaming applications can support higher delay bounds, from around 0.5 seconds (to support channel switching) to tens of seconds for on-demand video delivery.

TCP senders interpret the receipt of three duplicate acknowledgements as an indication that a segment has been lost, and that it should be retransmitted. It follows that the time needed

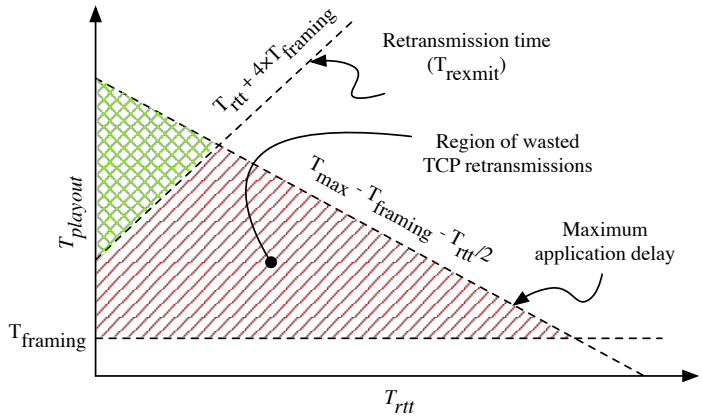


Figure 5.1: Utility of TCP retransmissions: retransmissions arrive too late to be used, as in the red, lined region

by a sender to identify packet loss following a transmission has a lower bound of:

$$T_{rexmit} = T_{rtt} + 3 \times T_{framing} \tag{5.2}$$

This assumes that a frame fits into a single TCP segment; T_{rexmit} will be lower if frames consist of multiple segments. At the receiver, there is an additional framing interval to allow for decoding and gap-free playout. Assuming that media decoding and rendering take negligible time, a retransmitted packet will arrive in time to be received and rendered at the application, provided:

$$T_{playout} \geq T_{rexmit} + T_{framing} \tag{5.3}$$

When $T_{playout} < T_{rexmit} + T_{framing}$, retransmissions of the original message will arrive after the data was scheduled to be rendered, and will be discarded by the application. This gives a lower bound on $T_{playout}$ above which standard TCP retransmissions are useful. The corresponding upper bound is the maximum delay acceptable to the application, T_{max}. Assuming that media encoding delay is negligible, T_{sender} approximates $T_{framing}$ ($T_{sender} \approx T_{framing}$). By combining these bounds, the range of playout delays for which standard TCP retransmissions will arrive in time to be rendered to the application is:

$$T_{rtt} + (3+1) \times T_{framing} \leq T_{playout} \leq T_{max} - T_{framing} - \frac{T_{rtt}}{2} \tag{5.4}$$

These bounds are illustrated in Figure 5.1. The unshaded regions in Figure 5.1 fall outside the feasible operating regime of the application, and may be ignored: selecting a playout delay in these regions will result in stalls in playout, or violations of the maximum application delay bound. The feasible operating parameters are represented by the shaded regions that separate useful and wasteful retransmissions. The green cross-hatched area highlights the region where standard TCP retransmissions arrive in time to be useful: the playout delay is

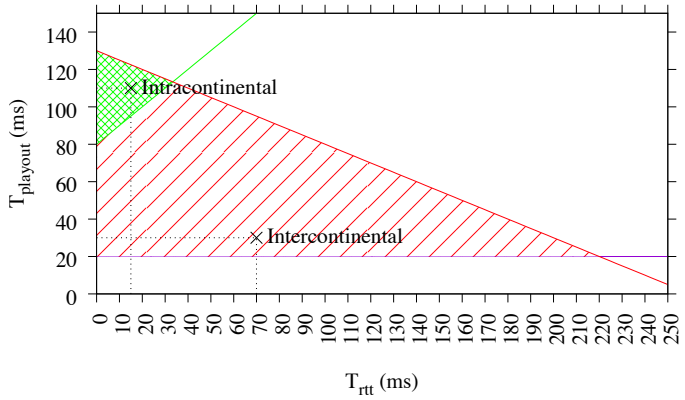


Figure 5.2: Utility of TCP retransmissions for VoIP applications, with $T_{framing} = 20$ms and $T_{max} = 150$ms: standard TCP retransmissions arrive too late to be used, as in the red, lined region

large enough to accommodate the retransmission, but low enough to meet the application's overall delay requirements.

Wasteful TCP retransmissions are marked by the red lined region in Figure 5.1. When the media playout delay is less than the retransmission time (i.e., $T_{playout} < T_{rexmit} + T_{framing}$) and satisfies the overall delay bound ($T_{playout} \leq T_{max} - T_{framing} - \frac{T_{rtt}}{2}$), and is greater than the framing interval ($T_{playout} \geq T_{framing}$), then standard TCP retransmissions will arrive too late to be played out. This is where inconsistent retransmissions are useful: when a standard TCP retransmission will arrive too late to replace the original lost segment in this region. By contrast, an inconsistent retransmission can use that slot to transmit usable data. The lost segment is never recovered, and wouldn't be useful if it was, but its sequence number is reused to send data that will be useful when it arrives.

The benefits of inconsistent retransmissions can be quantified by substituting parameters for different low latency applications into Equation 5.4. First, I consider interactive voice telephony. As discussed in Chapter 2, widely deployed speech codecs typically use $T_{framing} = 20$ms, with a delay bound of $T_{max} = 150$ms [38]. Assuming that media encoding delays are negligible, so that $T_{sender} = T_{framing}$, the feasible region where standard TCP retransmissions arrive in time to be useful can be derived from Equation 5.4 as:

$$130\text{ms} - \frac{T_{rtt}}{2} \geq T_{playout} \geq T_{rtt} + 80\text{ms} \tag{5.5}$$

which has valid solutions for $T_{playout}$ provided that $T_{rtt} \leq 33.3$ms, as illustrated in Figure 5.2. This round-trip bound is low for wide-area networks. For example, TCP retransmissions would be useful for calls within Europe, but are likely to be wasteful during intercontinental calls. Figure 5.2 shows that TCP Hollywood's inconsistent retransmissions provide valid solutions for $T_{playout}$ when $T_{rtt} \leq 220$ms (i.e., one way delay ≤ 110ms) providing utility for

Application	T_{max}	RTT Bound (ms)		Utility	
		Standard	Hollywood	Standard	Hollywood
Voice telephony	150	33.3	220	Within a continent	Intercontinental
On-demand video	30000	13333.3	52000	Intercontinental	Intercontinental
IPTV	1000	0.0	1200	None	Intercontinental

Table 5.1: Sample standard TCP and TCP Hollywood RTT bounds required to meet application bounds

calls over intercontinental links (i.e., links where T_{rtt} exceeds 70ms).

For on-demand video streaming using the MPEG-DASH [76] protocol, the framing interval and delay bounds are typically much larger. A typical deployment today might use an encoding segment size of $T_{framing} = 2$s, and an overall delay bound of $T_{max} = 30$s. Assuming that $T_{sender} = T_{framing}$, and substituting into Equation 5.4, TCP retransmissions are useful provided that $T_{rtt} \leq 13.3$s, giving no benefit from inconsistent retransmissions.

These two applications represent extremes in terms of latency bounds: voice telephony has tight bounds, while those of on-demand video streaming are relaxed. I analyse a third application: IPTV delivery using MPEG-DASH. IPTV applications seek to minimise *zap time* (i.e., the total time taken between a viewer selecting a channel, and content from that channel being displayed). Bouzakaria et al. [5] show that end-to-end latencies – the time between encoding and decoding of a frame – of less than 240ms can be achieved over MPEG-DASH. Using their techniques, segments are fragmented into 200ms chunks for delivery, giving $T_{sender} = T_{framing} = 200$ms. An overall delay bound of $T_{max} = 1$s allows for channel surfing to be supported. Substituting these values into Equation 5.4 indicates that TCP retransmissions are wasteful for all RTT values. In contrast, inconsistent retransmissions in TCP Hollywood can be used when $T_{rtt} \leq 1$s.

Table 5.1 summarises the three applications considered. In summary, the utility of inconsistent retransmissions is largely dependent on the latency bounds of the application. Interactive applications, where the overall latency requirements are tight, can strongly benefit from the ability to send data in place of a retransmission, but those applications with relaxed latency bounds find less benefit.

5.1.2 Experimental Setup

To demonstrate that the analysis presented holds true, I will perform a series of preliminary evaluations. These evaluations are not designed to fully exercise the features of TCP Hollywood, nor will they demonstrate the broader performance implications of the protocol. The focus here is narrowed to empirically validating the analysis described, showing

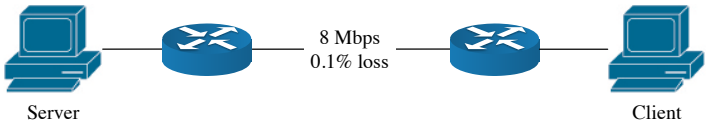


Figure 5.3: Topology used in VoIP performance evaluations

that inconsistent retransmissions and the elimination head-of-line blocking are beneficial in the regions identified. To go beyond this, I will describe the application and transport layer changes required to fully utilise TCP Hollywood, and evaluate its overall performance, in Chapter 6.

The evaluations in this chapter are carried out using the implementation of TCP Hollywood described in Sections 4.2.1 and 4.3.1. Beyond validating the analysis, as described above, these evaluations will also validate this implementation, demonstrating that inconsistent retransmissions and out-of-order delivery function as described in Chapter 4.

For the evaluations here, I use the Mininet [1] network emulator, configured with the topology shown in Figure 5.3. The bottleneck link of the dumbbell simulates an ADSL connection, with a download speed of 8Mbps, and an upload speed of 1Mbps. ADSL is simulated here because its use remains widespread in large parts of the world. Two round-trip times will be simulated, reflecting two classes of paths: an intracontinental link, with a round-trip time of 15ms, and an intercontinental link, with a round-trip time of 70ms.

The application used is a simulated voice-over-IP application, that sends small, 160-byte messages every 20ms, as is typical (e.g., with the G.711 codec). The application will run for 1 minute, sending 3,000 messages. Two play-out delays will be simulated, derived from Figure 5.2: 110ms in the intracontinental scenario, and 30ms in the intercontinental scenario. These numbers have been selected to show that both the regions of Figure 5.2 exist: in the intracontinental scenario, with the play-out delay selected, TCP Hollywood offers no benefit beyond standard TCP. However, in the intercontinental scenario, where standard TCP is no longer useful, given that retransmitted messages will exceed their delay bounds, TCP Hollywood should offer some benefit. It is worth noting the high bounds on play-out delay that TCP places on applications: in the intercontinental scenario, play-out delays lower than the 110ms selected would fall into the region where TCP retransmissions would not be useful.

Finally, random packet loss is introduced at a rate of 0.1%. Random packet loss is used here, instead of a more realistic loss model or congestive loss induced by cross-traffic, to induce discrete, identifiable loss events. While this allows for empirical validation of the analysis, no claims can be made about TCP Hollywood's performance more broadly. I will show what TCP Hollywood does in response to loss in the regions that the analysis identifies.

[1] `http://mininet.org`

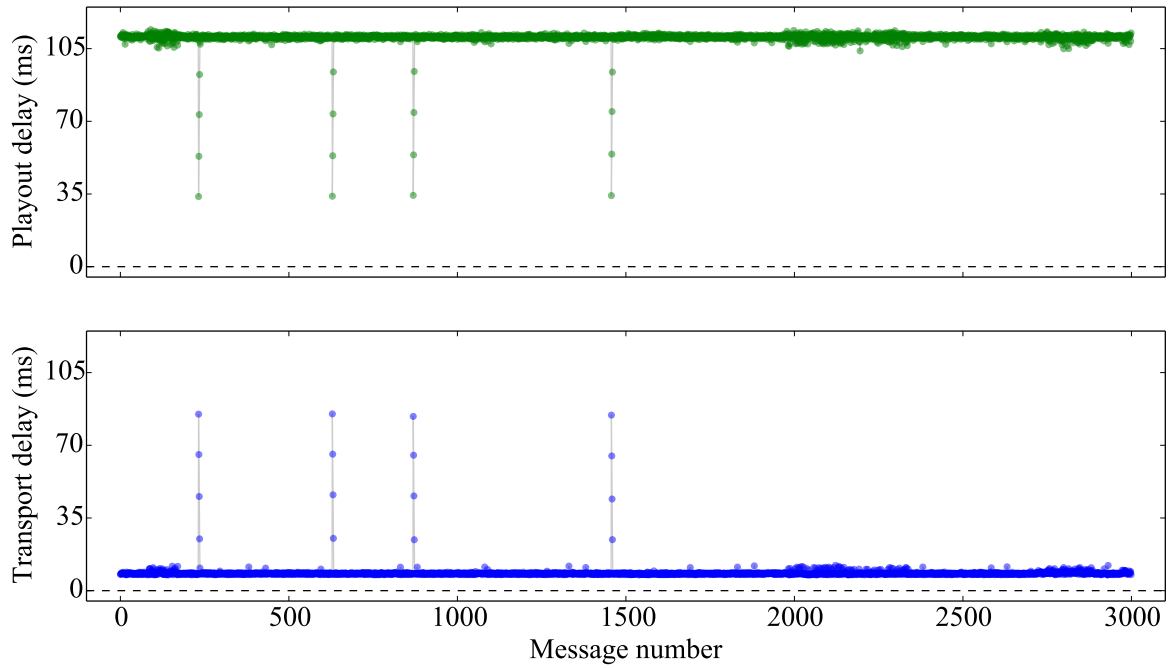


Figure 5.4: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 15ms RTT and 110ms playout delay, using standard TCP

5.1.3 Evaluations

To begin, I evaluate the performance of the VoIP application in the intracontinental scenario, with an 8Mbps link, 15ms RTT, and 110ms playout delay. Figure 5.4 shows the playout delay (i.e., the duration between a message arriving, and its playout time) and transport delay (i.e., the duration between the message being sent and it arriving at the receiver application), for each message sent by the simulated application. This plot shows the impact of loss under TCP: when a TCP segment[2] is lost, it is retransmitted as described earlier, and delivered to the application. This results in the spike in transport delay, and in the message being buffered at the application for less time: this is shown by the dips in time buffered. To highlight this behaviour, Figure 5.5 focusses on a smaller range of messages. In this plot, message 1457 has been lost, and retransmitted: this causes it to be buffered for only 35ms before being played out. As can be seen, there is a tail of subsequent messages that are also buffered for less time: these messages have been head-of-line blocked waiting for the delivery of the retransmission of message 1457. As will be discussed in the next section, they are delivered once this retransmission arrives. The stepping pattern is caused by the 20ms framing interval: all of the head-of-line blocked messages are delivered at the same time (i.e., when message 1457 arrives), but are played out at 20ms intervals.

[2]"TCP segment" and "message" are used interchangeably throughout. A one-to-one mapping largely exists, given that Nagle's algorithm is disabled in all evaluations. However, messages may be coalesced or segmented when they are retransmitted. The impact of this behaviour, for the scenarios used in this chapter, is negligible.

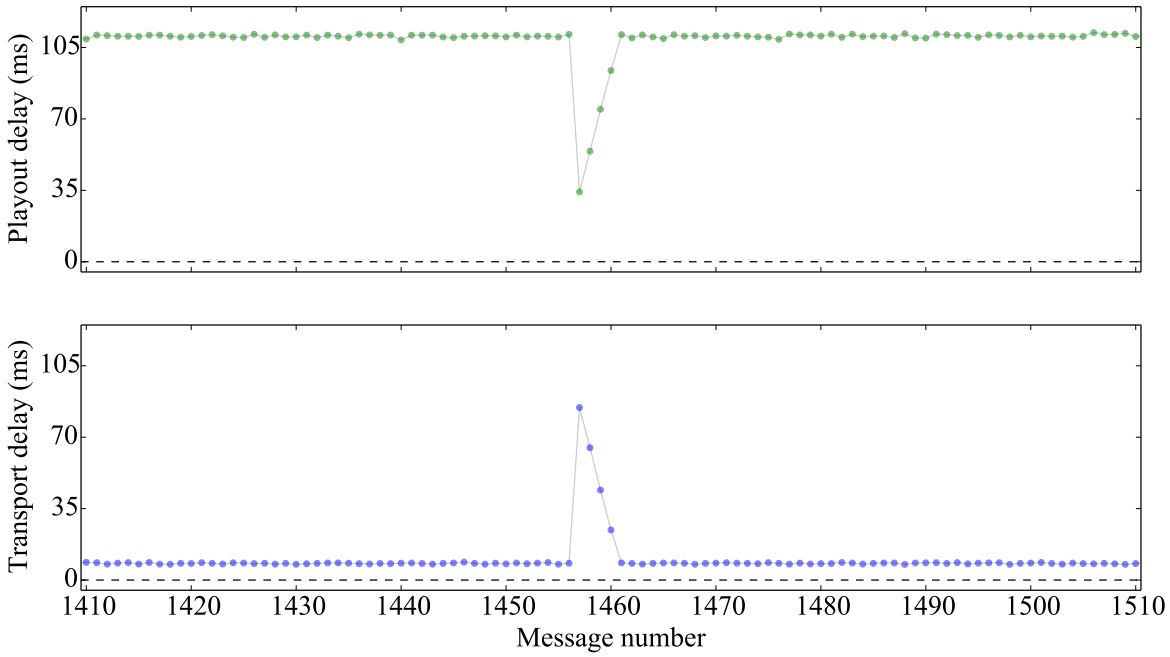


Figure 5.5: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 15ms RTT and 110ms playout delay, using standard TCP (zoomed in on messages 1410 through 1510)

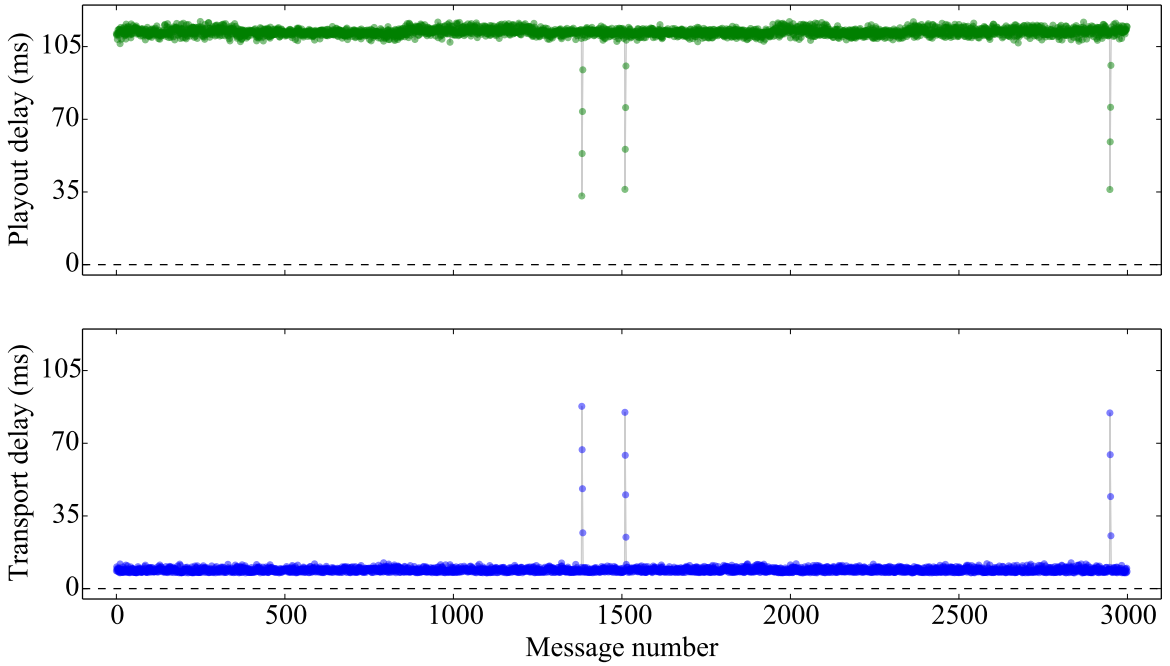


Figure 5.6: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 15ms RTT and 110ms playout delay, using TCP Hollywood with inconsistent retransmissions enabled

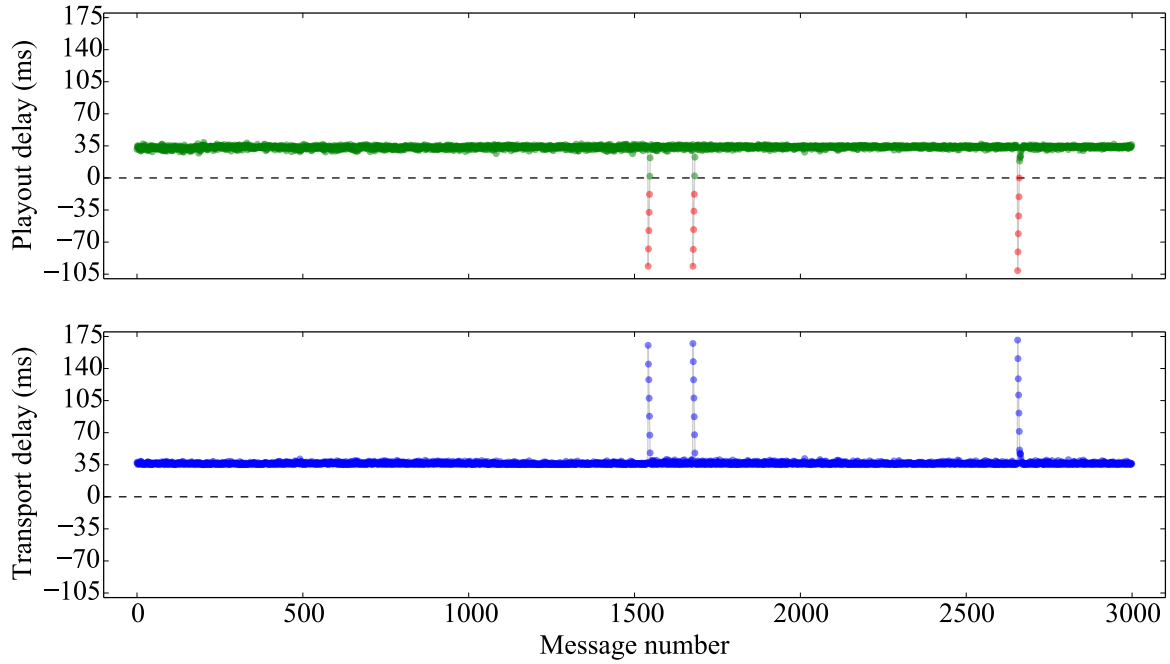


Figure 5.7: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 70ms RTT and 30ms playout delay, using standard TCP

Importantly, Figure 5.4 shows that no messages arrive too late to be played out: these would be shown by red, negative buffered times. As a result, TCP Hollywood is unable to help: standard TCP retransmissions operate well in this scenario, detecting and retransmitting lost packets within the playout delay bounds. It is important that TCP Hollywood does not harm performance: if inconsistent retransmissions were to be triggered incorrectly, then usable data would be lost. Figure 5.6 repeats the experiment shown by Figure 5.4, but over TCP Hollywood with only inconsistent retransmissions enabled. As shown, the behaviour in response to loss is the same: lost segments are buffered for a shorter time, but they do arrive on time to be played out. No messages are lost by TCP Hollywood, because inconsistent retransmissions are not triggered for messages that are estimated to arrive before their deadline.

However, when standard TCP is used for the same application in the intercontinental scenario – that is, an 8Mbps link with 70ms RTT and playout delay set to 30ms – retransmitted messages arrive too late to be played out. Figure 5.7 illustrates this: the red, negative buffered times indicate that the messages arrive after their scheduled playout times, resulting from the spike in transport delay. Figure 5.8 focuses on the lost message 1541. This message is lost and subsequently retransmitted, but the combination of high delay and low playout time (to meet the application's maximum delay bound) mean that retransmissions cannot be accommodated. This impacts on the quality-of-service provided by the application: it cannot playout the missing media at the required time. In the case of the simulated VoIP application, this would result in glitches in the audio being heard by the receiver.

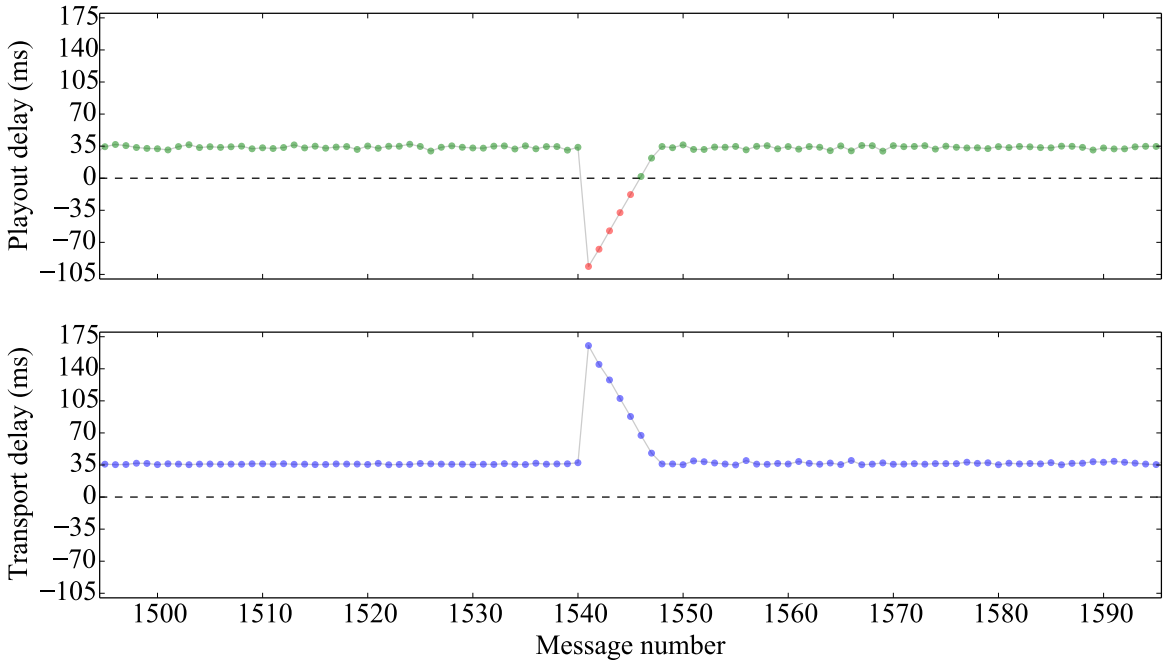


Figure 5.8: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 70ms RTT and 30ms playout delay, using standard TCP (zoomed in on messages 1495 through 1595)

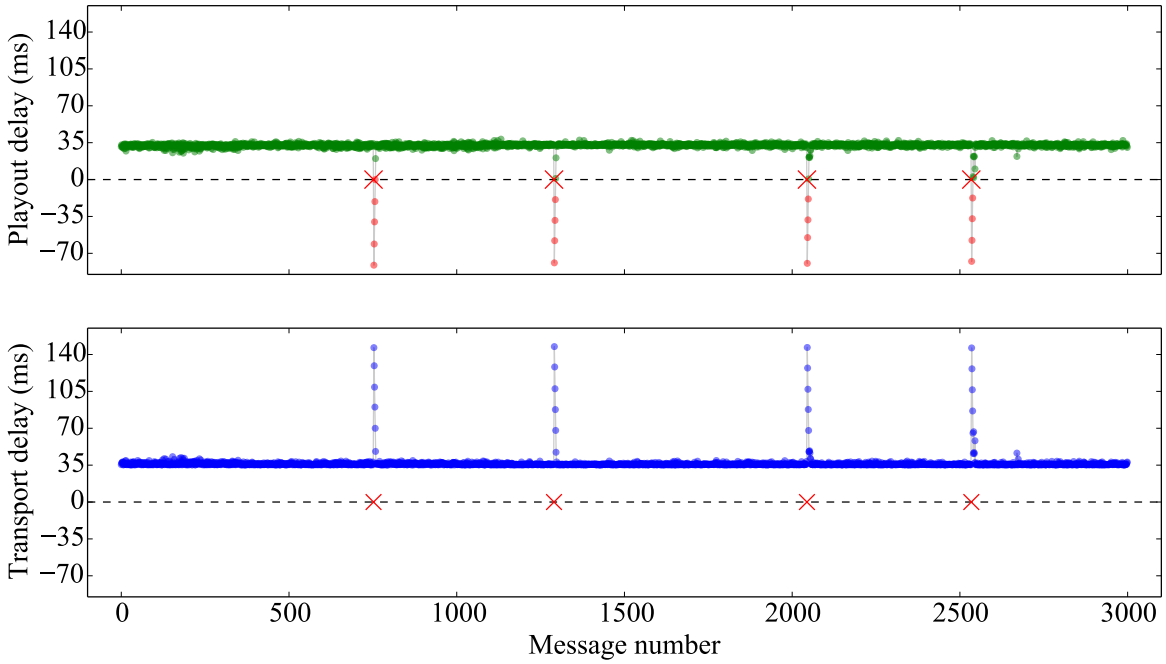


Figure 5.9: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 70ms RTT and 30ms playout delay, using TCP Hollywood with inconsistent retransmissions enabled (red crosses indicate lost messages)

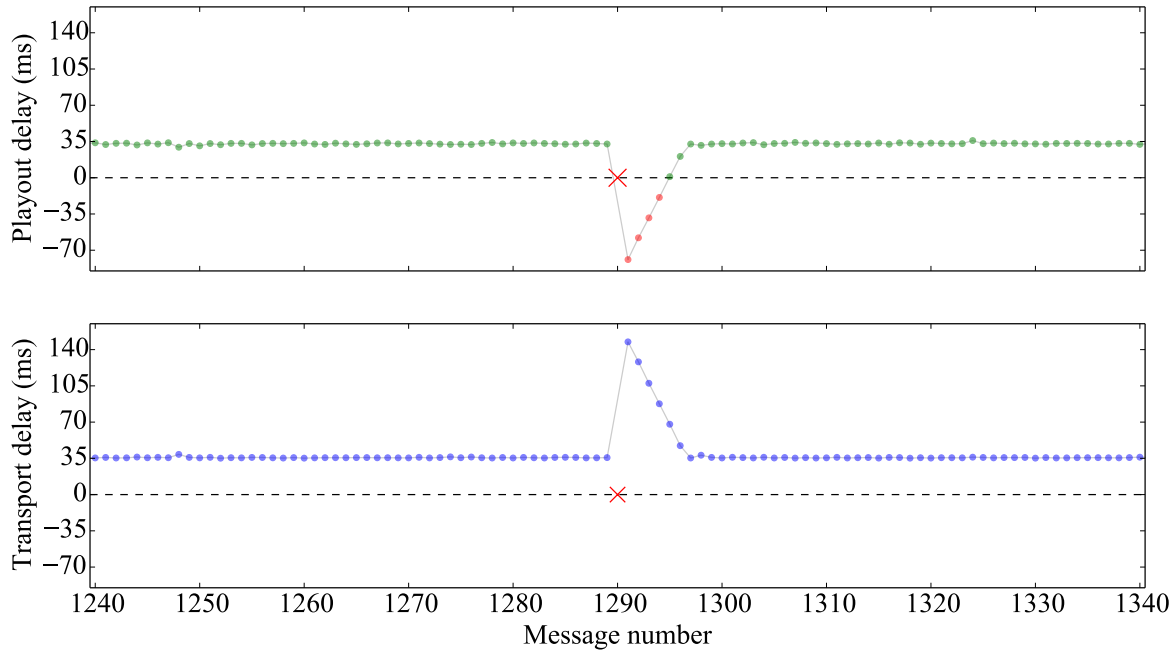


Figure 5.10: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 70ms RTT and 30ms playout delay, using TCP Hollywood with inconsistent retransmissions enabled (red crosses indicate lost messages, zoomed in on messages 1240 through 1340)

However, the analysis presented in the previous section – and summarised for the simulated VoIP application in Figure 5.2 – indicates that this scenario should fall into the region where TCP Hollywood's inconsistent retransmissions help. Figure 5.9 validates this analysis: the red crosses indicate lost messages that result from them being swapped for an alternative message due to the inconsistent retransmission mechanism. Figure 5.10 focuses on lost message 1290. Unlike standard TCP, where message 1290 would have been retransmitted, and would have arrived too late to have been useful, TCP Hollywood inconsistently retransmitted the lost segment with a different message. This means that message 1290 is never delivered to the application. However, head-of-line blocking still causes a tail of messages (1291 through 1294) to miss their playout times. These messages have arrived at the receiving host, and could therefore be played out, but standard TCP's in-order delivery abstraction means that they are not delivered to the application until the retransmission of the segment that originally contained message 1290 has been delivered. In the next section, I will analytically explore the impact of head-of-line blocking.

5.2 Removing Head-of-Line Blocking

In the last section, I showed that inconsistent retransmissions are useful within bounds that are determined by the latency constraints of a given application. In this section, I extend

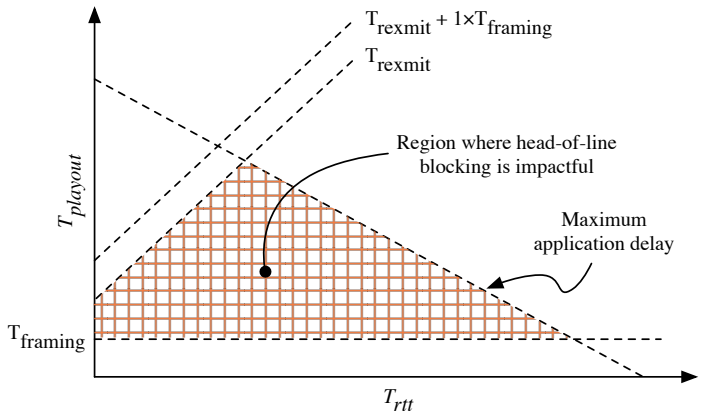


Figure 5.11: Head-of-line blocking in TCP impacts performance within the orange, hatched region

the analysis (Section 5.2.1) and evaluations (Section 5.2.2) given previously to include the impact of eliminating head-of-line blocking.

5.2.1 Analysis

As noted in Section 4.1, if a packet is lost, then a TCP sender will send a retransmission upon receipt of a triple duplicate acknowledgement. If standard TCP is used, then segments that arrive in the intervening period – that is, between the time that the lost segment should have arrived at the receiver, and the time that its retransmission does arrive – will be buffered, and not delivered to the application until the missing segment's retransmission has arrived, potentially causing media playout to stall. This is head-of-line blocking, as described in Chapter 4.

The size of the playout buffer relative to the round-trip time and media framing interval determines whether playout stalls, or whether there is sufficient buffering to cover the retransmission delay. As shown by Equation 5.3, if $T_{playout} \geq T_{rexmit} + T_{framing}$, then the retransmission will arrive in time to be played out, and no head-of-line blocking will occur.

However, if $T_{playout} < T_{rexmit} + T_{framing}$, then the retransmission will *not* arrive in time to be played out. This will cause a 1-frame gap in the media play out, since some data is missing. This occurs with both standard TCP, and with the TCP Hollywood extensions. If standard TCP is used, then the receiver may additionally be impacted by head-of-line blocking, and be unable to access later segments, resulting in a longer gap in play out.

If the retransmission arrives less than one framing interval after it was scheduled to be played out (i.e., if $T_{playout} - T_{rexmit} - T_{framing} \geq T_{framing}$), then it will arrive before the following frame is to be played. In this case, there is no head-of-line blocking: the receipt of the retransmission, though delayed and the cause of a one-frame gap in playback, allows for the

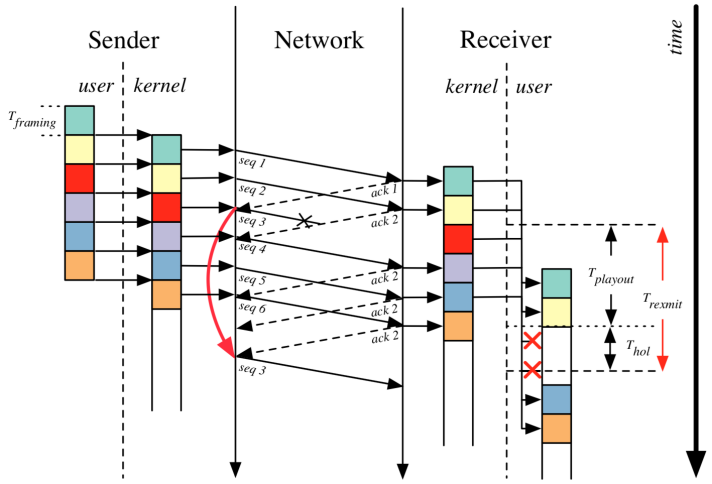


Figure 5.12: The relationship between $T_{playout}$, T_{rexmit}, and T_{hol} in standard TCP

delivery of the subsequent frames in time for their playback. If the retransmission is further delayed then head-of-line blocking will cause one or more later frames to also miss their playout. The region in which head-of-line blocking impacts performance is illustrated in Figure 5.11.

The duration of the impact (i.e., the duration of the discarded frames), can be modelled as T_{hol}:

$$T_{hol} = T_{rexmit} - T_{playout} = T_{rtt} + 3 \times T_{framing} - T_{playout} \tag{5.6}$$

From this, the number of discarded frames, N_{hol}, can be expressed as:

$$N_{hol} = \max\left(\left\lceil \frac{T_{hol}}{T_{framing}} \right\rceil, 0\right) \tag{5.7}$$

Figure 5.12 illustrates head-of-line blocking in standard TCP. In this example, segment 3 is lost, and must be retransmitted. While this retransmission is being triggered and sent, segments 4, 5, and 6 arrive at the receiver, but are not delivered to the application: they are head-of-line blocked. When the retransmission of segment 3 arrives, it is delivered to the application, along with the blocked segments 4, 5, and 6. As shown, $T_{playout}$ is less than T_{rexmit}, and so segment 3 has arrived too late to be used. Additionally, given that T_{hol} is greater than zero, head-of-line blocking has resulted in segment 4 being discarded by the application. Importantly, this is despite its on time arrival at the receiver: under TCP Hollywood, segments 4, 5, and 6 would be delivered to the application in time to be useful.

Using Figure 5.12 to summarise TCP Hollywood (vs. standard TCP) more broadly, there are two behaviours: (i) inconsistent retransmissions will be triggered for the retransmission of segment 3, increasing network utility, decreasing latency, and improving goodput; and (ii) segment 4 would be delivered on time to be played out successfully.

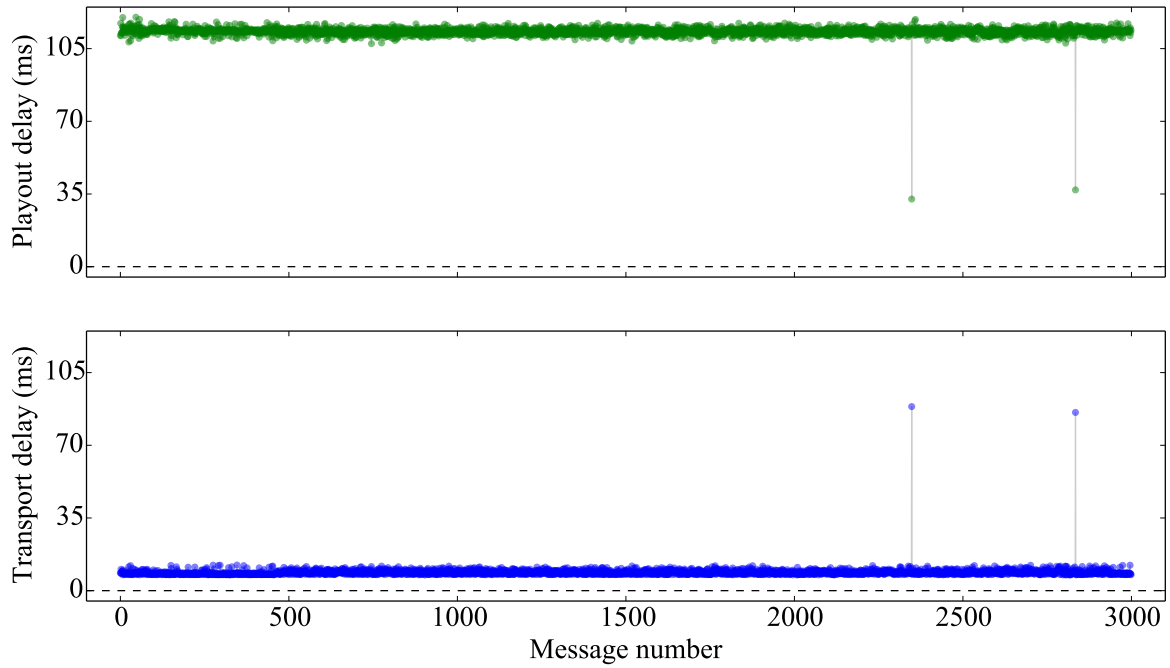


Figure 5.13: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 15ms RTT and 110ms playout delay, using TCP Hollywood with both inconsistent retransmissions and out-of-order delivery enabled

The impact of head-of-line blocking on the application-level loss rate is significant: it amplifies the network loss rate. To ensure that the impact of its removal is maximised, messages sent by applications should be independently useful. This is a central tenet of the application-level framing principle [13] discussed in Chapter 2.

5.2.2 Evaluations

As indicated by the analysis in the previous section, head-of-line blocking only has an impact on performance when standard TCP's retransmissions arrive too late to be used. As a result, eliminating head-of-line blocking should not impact the number of useful messages under the intracontinental scenario, with an RTT of 15ms and a playout delay of 110ms. As shown in Figures 5.4 and 5.5, standard TCP's retransmissions are accommodated by the large playout delay; as a result, head-of-line blocking does not impact on performance. To demonstrate this, I repeat the intracontinental scenario over over TCP Hollywood with both inconsistent retransmissions and out-of-order delivery enabled. The playout delay and transport delay for each message is plotted in Figure 5.13, while Figure 5.14 focuses on lost message 2348. As shown, TCP Hollywood does not inconsistently retransmit the segment containing message 2348: it arrives successfully, in time for playout. However, the impact of eliminating head-of-line blocking can be seen: the tail of delayed messages that would have occurred (and is present in Figure 5.5, which plots the same scenario, but with out-of-order deliv-

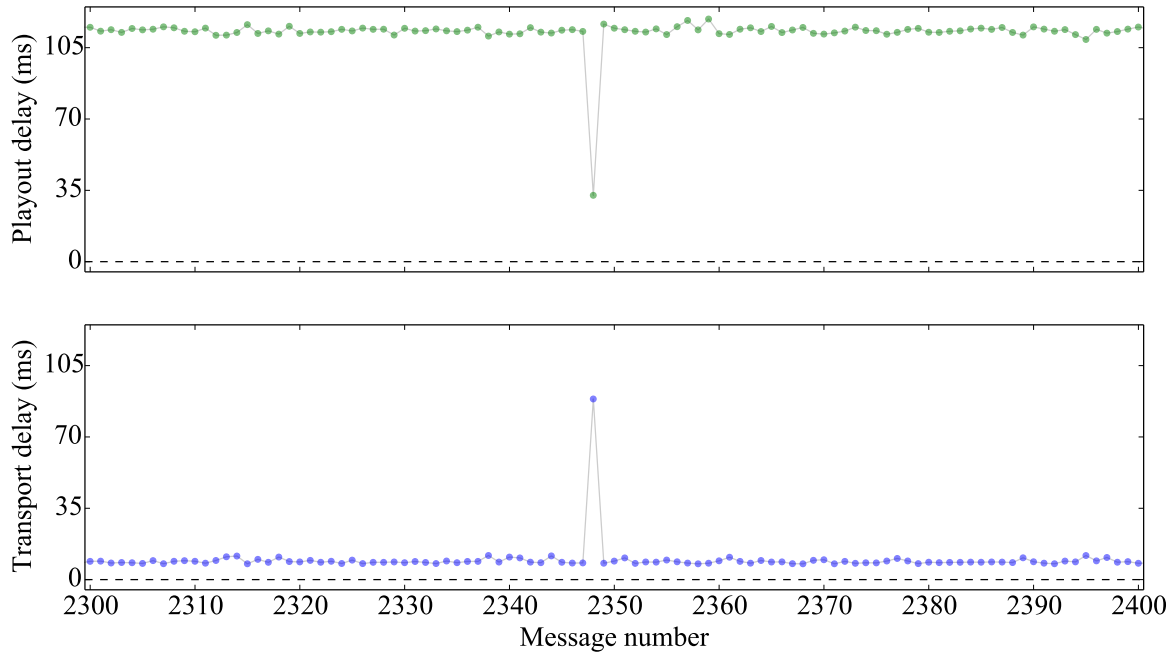


Figure 5.14: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 15ms RTT and 110ms playout delay, using TCP Hollywood with both inconsistent retransmissions and out-of-order delivery enabled (zoomed in on messages 2300 through 2400)

ery disabled) does not appear. Instead, these messages – that is, those that arrive after the original transmission of message 2348, and the arrival of its retransmission – are delivered to the application. Importantly, this does not affect performance in this scenario: as shown in Figure 5.5, these messages would have been useful to the application even if they had been head-of-line blocked. However, this scenario does illustrate how head-of-line blocking works, and that its elimination results in the messages following a loss being delivered to the application earlier.

To illustrate the impact of eliminating head-of-line blocking where it impacts performance, I simulate the VoIP application using the intercontinental scenario, with an 8Mbps link with 70ms RTT and 30ms playout delay. As shown in Figures 5.9 and Figures 5.10, TCP Hollywood's inconsistent retransmissions were triggered under this scenario: standard TCP's retransmissions cannot be accommodated by the playout delay, which is necessarily small, given the round-trip time and maximum delay tolerated by the application. Figure 5.15 shows the total buffered time for each message when this scenario is re-run with both inconsistent retransmissions and out-of-order delivery enabled. Figure 5.16 focuses on lost message 2263. As shown by the red cross, this message is completely lost: TCP Hollywood inconsistently retransmitted the segment that contained it, replacing it with data from another message. However, when compared with Figure 5.10, Figure 5.16 illustrates the additional performance benefit of eliminating head-of-line blocking: there is no longer a

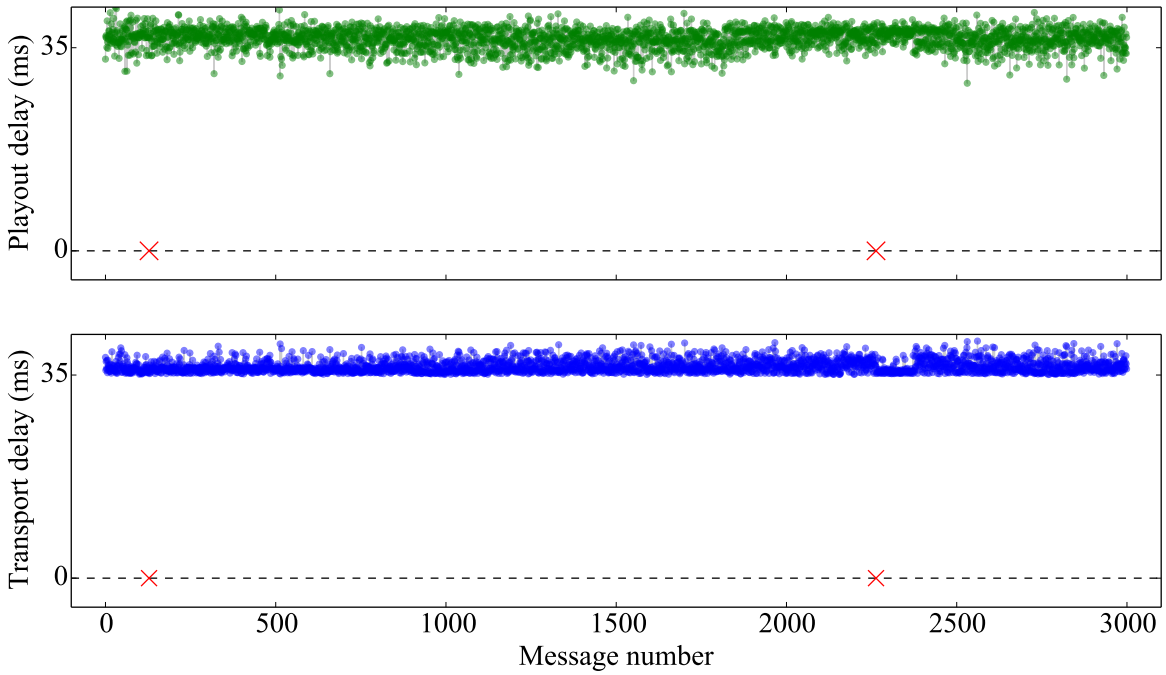


Figure 5.15: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 70ms RTT and 30ms playout delay, using TCP Hollywood with both inconsistent retransmissions and out-of-order delivery enabled (red crosses indicate lost messages)

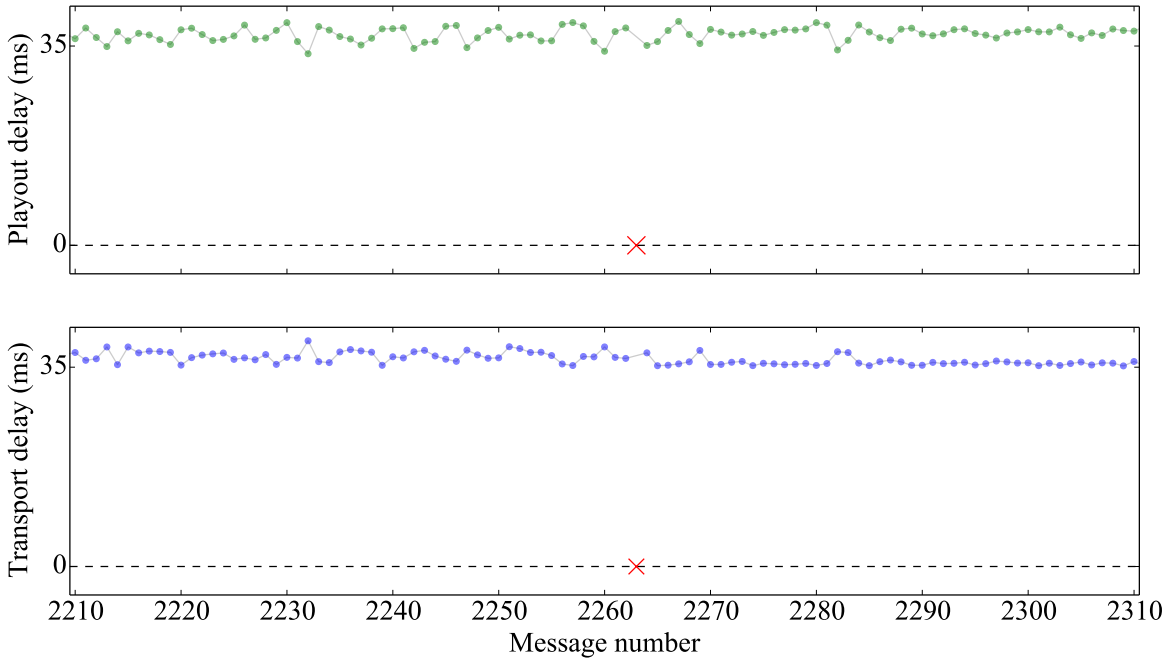


Figure 5.16: Playout delay and transport delay for a simulated VoIP application over an 8Mbps link with 70ms RTT and 30ms playout delay, using TCP Hollywood with both inconsistent retransmissions and out-of-order delivery enabled (red crosses indicate lost messages, zoomed in on messages 2210 through 2310)

tail of messages that miss their playout times. Messages 2264 through 2267 would, under standard TCP, miss their playout time, amplifying the affect of the lost message 2263. With TCP Hollywood, inconsistent retransmissions ensure that message 2263 is not retransmitted unnecessarily – it is not useful to the application – while out-of-order delivery ensures that messages that arrive at the receiving host are delivered to the application in time for playback.

5.3 Deployability

The only wire-visible change introduced by TCP Hollywood, when compared with standard TCP, is its use of inconsistent retransmissions, where the same sequence number is reused in segments that carry different payloads. This modification is invisible to receivers and middleboxes that only process TCP/IP headers. However, it *is* visible to middleboxes that perform deep packet inspection, where the contents of a retransmitted packet are compared with the original. Depending on the configuration of the middlebox, the detection of unusual protocol behaviour may disrupt the connection. For example, a firewall may interpret inconsistent retransmissions as having been injected into the stream as part of a man-on-the-side attack, and opt to reset the connection.

To obtain an initial assessment of whether such middleboxes exist, I present a set of experiments with a live deployment of TCP Hollywood. A TCP Hollywood server was setup on the public Internet, and configured to always send inconsistent retransmissions in lieu of the original data, so that all retransmissions contained a different payload with the same sequence numbers. The server was configured to listen on ports 80, 4001, and 5001. Port 80 is used by web traffic, and can be expected to be affected by HTTP-targeting middleboxes, such as transparent caches and firewalls. We expect ports 4001 and 5001 to be subjected to less interference by middleboxes, given that they are not associated with widely used applications.

Clients were deployed across a number of access networks, operated by different service providers. Each client connected to the server, and received data. All incoming segments to the client were recorded by `tcpdump`, and then filtered by `iptables` to uniformly drop 5% of segments before they reached the sender's TCP stack for processing. While this packet loss rate is high, it is consistent with the goal of testing the deployability of TCP Hollywood, rather than its performance. The primary concern is with the creation of sufficient loss, in order to trigger inconsistent retransmissions. This high *uncorrelated* drop rate enables TCP to maintain connectivity, where it would fail at a similar rate of correlated drops. The high loss rate also reduces throughput, reducing congestive loss in the network. This increases the probability that the only losses are those induced by the experimental setup, making

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

Table 5.2: Deployability evaluation results, measuring the delivery of inconsistent retransmissions. ● indicates that the inconsistent retransmission passed through the network; ▲ indicates that the original data was delivered instead; and ■ indicates that a connection failure occurred. No connection failures were observed.

it more likely that the client will see both the original transmission, and its inconsistent retransmission. Packet loss was induced for traffic on ports 80 and 4001, leaving traffic on port 5001 unaffected.

Each loss induced at the client triggered an inconsistent retransmission from the server. Remaining segments were passed up the stack to the client application as normal. Data received by the client application was recorded, and compared against the `tcpdump` from the server to identify dropped segments. The payload of retransmitted segments was compared with the payload of the original transmission. This allowed for visibility into which segments had been lost, and for confirmation that both the original and inconsistent retransmission traversed the path between the client and server, and whether the inconsistent retransmission was delivered successfully.

These evaluations were conducted using clients in 14 different locations in the UK, connecting to a server located at the University of Glasgow. The results are summarised in Table 5.2. The clients connected via eight different fixed-line residential ISPs (Andrews & Arnold, BT, Demon, EE, Eclipse, Sky, TalkTalk, and Virgin Media), and four mobile operators (EE, O2, Three, and Vodafone). All of the fixed-line residential ISPs successfully delivered the inconsistently retransmitted payloads. In contrast, however, only one out of the four mobile operators delivered inconsistent retransmissions across both ports. The three

remaining mobile operators delivered the original segments instead, while the server did not see any corresponding segment loss and retransmission. This behaviour is consistent with a transparent split-connection TCP performance enhancing proxy cache, which intercepts and responds to ACKs from the client on behalf of the server. On two of the three providers, this caching behaviour was seen on both ports 80 and 4001, while the other provider appeared to operate a cache on port 80 only, likely to target HTTP traffic.

It is important to note that TCP Hollywood continued to operate whether or not middleboxes on the path retransmitted the original payload. At no time did connections suffer a reset, and the use of the TCP Hollywood extensions did not affect connectivity or performance. Middlebox interference, including caching, is designed to be transparent, leaving the client to believe it is interacting with a standard TCP server. As discussed in Chapter 4, TCP Hollywood is designed for partial deployment. These experiments show that TCP Hollywood continues to deliver messages, and eliminate head-of-line blocking, even when inconsistent retransmissions are absent. In the worst case, performance is the same as standard TCP.

The set of networks tested is by no means exhaustive. Further, and larger scale, evaluations are needed to build evidence that inconsistent retransmissions are widely deployable. Previous studies provide optimism, however. Honda et al. [33] investigated deployment of TCP modifications with regards to middlebox interaction, with clients on 142 networks in 24 countries. These measurements included the testing of inconsistent retransmissions across a large number of diverse paths. Their observations mirror those presented here: while in a small number of paths the original data is delivered, in the majority of cases, inconsistent retransmissions are delivered as expected. They observed connection resets on a single path, representing less than 1% of the paths they evaluated.

5.4 Summary

In this chapter, I have analytically identified the combination of application and network conditions in which standard TCP's fully reliable, in-order delivery model poorly serves applications. As described in Section 5.1, this occurs where the application cannot size its playout buffer sufficiently to accommodate the time taken for standard TCP to identify and retransmit lost segments. When this is the case, the retransmitted segment will arrive too late to be useful to the application. Under TCP Hollywood, *inconsistent retransmissions* are used when standard TCP retransmissions would arrive too late. Standard TCP amplifies the impact of a lost segment, delaying segments that have arrived on time to wait for the delivery of the retransmission of the lost segment. Section 5.2 discussed this head-of-line blocking delay, and identified the application and network conditions that result in it impacting application performance. In each case, I performed evaluations using an implementation of TCP

Hollywood in the Linux 3.18.34 kernel, validating both the analysis and the functionality of the implementation.

Finally, by way of real-world experiments, I have shown that TCP Hollywood is widely deployable: in all fixed-line networks in the UK, inconsistent retransmissions pass through the network. In networks where inconsistent retransmissions are not delivered, the preliminary evaluations in this chapter have shown the original message is delivered in its place. This is a safe failure mode for TCP Hollywood, with a delivery model that is partially reliable and unordered.

In the next chapter, I will go beyond the analysis and evaluations presented here by describing how an MPEG-DASH application can migrate from using standard TCP to using TCP Hollywood. Further, I will evaluate the broader performance implications of TCP Hollywood, illustrating its benefits within typical network environments.

Chapter 6

Lowering Latency in MPEG-DASH

In Chapter 4, I described the design and architecture of TCP Hollywood, which makes use of inconsistent retransmissions and eliminates head-of-line blocking to minimise the latency that is introduced at the transport layer. In Chapter 5, I demonstrated that there are application and network conditions under which TCP Hollywood is more suitable for low-latency applications, when compared to standard TCP. However, the performance evaluations presented so far have been limited to validating the design and analysis of TCP Hollywood: they have not shown how low-latency applications that typically use TCP should be modified to use TCP Hollywood, or the performance benefits that such a change might bring.

In this chapter, I will detail a set of changes to an MPEG-DASH application to better support its use for low-latency video delivery. As discussed in Chapter 2, the design of MPEG-DASH was largely driven by the need to use HTTP, and therefore TCP, to maximise deployability. Given that its use of standard TCP has limited its applicability for low-latency applications, MPEG-DASH is a good choice for demonstrating TCP Hollywood's performance benefits, whilst taking advantage of its wire-compatibility with standard TCP.

This chapter is structured as follows:

Section 6.1 describes the architecture of existing MPEG-DASH applications, and how they use HTTP/2. Further, analysis and empirical evaluations are described that benchmark the latency performance of these existing applications;

Section 6.2 discusses the performance improvements that might be realised by switching to a progressive streaming architecture, an application layer change, and presents analysis and empirical evaluations that model such a change;

Section 6.3 details how the simulated application can be further modified to use TCP Hollywood at the transport layer, and evaluates the performance benefits of doing so; and

Section 6.4 summarises the chapter.

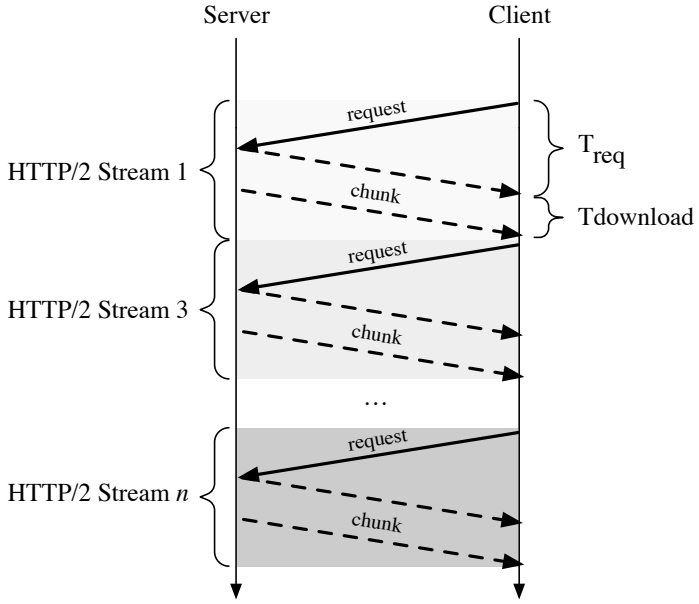


Figure 6.1: MPEG-DASH request-response architecture: chunks are requested and delivered in separate streams; must have fully arrived before being played

6.1 Existing Architecture

In MPEG-DASH, multimedia is split into *chunks* of equal duration, with each chunk made available at multiple bitrates. Clients request each chunk in sequence, using a format specified by the server in a *manifest file*. Clients operate a rate adaptation algorithm to determine the bitrate to request for each chunk. These algorithms can use many signals to determine the best rate to request, including the client's assessment of network conditions, and its buffer occupancy levels.

Many applications using MPEG-DASH do so over HTTP/1.1, but given that, ultimately, my approach to lowering latency requires server push, I opt to use HTTP/2 throughout. The HTTP/2 protocol inherits many of the high-level concepts of HTTP/1.1, such as method types, header fields, and status/error codes; as a result, the baseline application is much the same as if using HTTP/1.1. The basic data transmission unit in HTTP/2 is a frame, with different frame types serving different purposes. For example, a `HEADERS` frame is used to transmit header data, while `DATA` frames contain payload data. HTTP/2 connections can be comprised of multiple concurrent bi-directional streams. Frames *within* streams are ordered, but no ordering is enforced *between* streams. Frames in one stream are not delayed awaiting the delivery of frames in another.

I begin with an MPEG-DASH application that uses HTTP/2 in the same way that HTTP/1.1 is typically used. Figure 6.1 shows how MPEG-DASH can be mapped to HTTP/2's request-response model. This is largely as it would be under HTTP/1.1, with clients requesting each chunk in sequence. To avoid *application* head-of-line blocking between chunks, each chunk

request and response occurs within its own HTTP/2 stream, with all streams carried over a single TCP connection. As a result, head-of-line blocking can still occur at the *transport* layer, as described in Chapter 5. Clients first request the manifest file, and then the chunks that are required to fill the receive-side buffer. Once playback begins (i.e., once the playout buffer has initially filled), the client requests subsequent chunks to maintain its buffer.

The characterisation of the performance of this application architecture takes the form of a set of analyses, working through the theoretical models that the architecture creates, followed by empirical evaluations using a simulator. In the simulator, I stream the same three-minute clip of Big Buck Bunny[1]. Four encoding rates are used, each at 60fps: 7.5Mbps (720p), 12Mbps (1080p), 24Mbps (2K), and 53Mbps (4K). These rates are chosen to reflect those used by YouTube[2]. I simulate different network conditions, including different round-trip times and loss rates, as required by each simulation. In both the analysis and the simulations, I do not operate a rate adaptation algorithm: it is assumed that the rate is fixed (at a specified value) throughout each session. The approach that I present does not include a new rate adaptation algorithm, and the benefits that it provides are independent of the rate adaptation algorithm selected. While the rate adaptation algorithm determines which chunk representation is selected, and when, the approach detailed in this chapter reduces the latency in delivering those chunks to the application layer. By fixing the rate, I am able to better model the impact of the factors that *are* affected by the proposed approach.

In addition, while video is sent from the server to the client, it is *not* decoded by the client. The client assumes that the video is available for playback on its arrival. This reflects the analysis that will be carried out, where decoding delay is not included, due to the process's dependence on the hardware and software used. As a result, in both the analysis and simulations, for all architectures described, the overall end-to-end delay is underestimated by the time taken to decode the video once it arrives at the client. This does not affect the generality of the approach described.

At the application layer, latency is bounded by the receive-side buffer, given that a chunk must have arrived in its entirety before it can be played out. This restricts the length of the receive-side buffer to multiples of the chunk duration, with the minimum buffer duration being that of a single chunk. Given the goal of minimising the receive-side buffer, an intuitive first step is to make chunks shorter. However, there are a number of issues with this approach. Making chunks smaller increases the number of requests received by the server, and reduces encoding efficiency. Further, the pull-based architecture of MPEG-DASH sets a lower bound on chunk size, limiting the impact of this approach. I analyse these issues in Section 6.1.1, and conduct simulations to validate this analysis in Section 6.1.2.

[1] `https://peach.blender.org`
[2] `https://support.google.com/youtube/answer/1722171`

6.1.1 Analysis

In this section, I analyse the impact that reducing chunk duration has on latency. To begin, I first introduce some notation. I define T_{chunk} as the duration of each chunk, $T_{session}$ as the duration of a given playback session (i.e., the duration of the video), T_{buffer} as the size of the receive-side buffer, and T_{fetch} as the sum of the time taken to both request (T_{req}) and download ($T_{download}$) each chunk. Next, I define $R_{encoding}$ and $R_{bottleneck}$ as the video encoding and bottleneck link rates respectively. Finally, I define S_{chunk} as the size of each chunk. I assume that S_{chunk} is constant throughout the session. This is the worst case scenario: under variable bitrate encoding, some chunks would be smaller.

I observe that there are three effects of reducing chunk duration: an increased number of requests, reduced compression efficiency, and a minimum chunk duration due to the network round-trip time.

The **increased number of requests** results from the application's per-chunk request architecture. Under the typical MPEG-DASH architecture, beyond the initial request for the manifest file, there is a single HTTP request for each chunk. Therefore, the number of requests, $N_{requests}$, can be defined as:

$$N_{requests} = \frac{T_{session}}{T_{chunk}} + 1 \tag{6.1}$$

As a result, if $T_{session}$ remains constant, lowering T_{chunk} – that is, making chunks shorter – will result in a greater number of HTTP requests being made to the server. This increases both network and server overheads.

Shortening chunk durations also **reduces compression efficiency**. In MPEG-2, video frames are compressed into three frame types: I, P, and B frames. I frames are independently decodable, and don't rely on data from other frames: this means that they are the largest. P and B frames encode the difference between themselves, and prior or future (in the case of B frames) frames, meaning that while they cannot be independently decoded, they can be significantly smaller than I frames. For example, in an encoding of Big Buck Bunny (3 second chunks encoded at 4300kbps), over the median frame sizes, P and B frames were 20 and 106 (respectively) times smaller than I frames.

MPEG-DASH chunks need to be independently decodable to support rate adaptation at chunk boundaries. This means that they *must* begin with an I frame. As a result, there is at least one I frame per chunk; shortening chunk durations increases the number of chunks (if $T_{session}$ is constant), and so, increases the number of I frames. Given that I frames are the largest frame type, this introduces a trade-off: if the average bitrate is to be maintained, then shorter chunks will increase the total encoding size. If quality is to be maintained, then shorter chunks will increase the total encoding size. This case is illustrated in Figure 6.2, which plots the total encoding size for each of four encodings of the Big Buck Bunny clip,

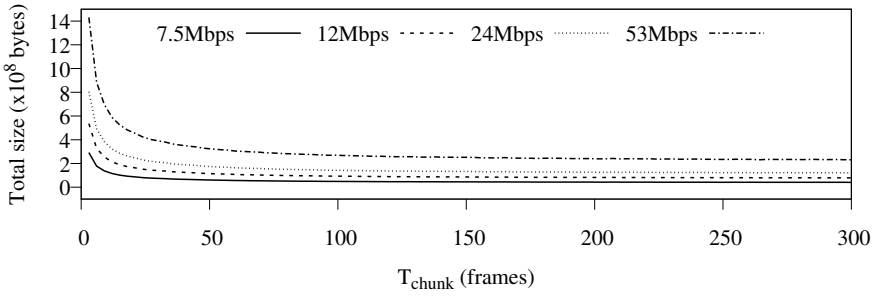


Figure 6.2: Total encoding sizes for various T_{chunk}s, where the original quality is maintained (bitrates refer to those of the original, unsegmented video)

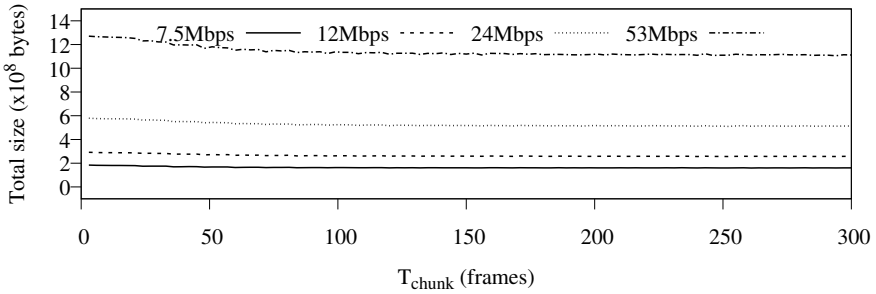


Figure 6.3: Total encoding sizes for various T_{chunk}s, where the target bitrate is maintained at the expense of quality (bitrates refer to those of the original, unsegmented video)

and various T_{chunk}s, where quality is maintained. As shown, there is a significant increase in encoding size as chunks become shorter. Given the bitrates that result from maintaining quality, I opt to reduce quality while maintaining the average bitrate (as illustrated in Figure 6.3). This is the approach taken by most MPEG-DASH applications.

Where playout can only begin on chunk boundaries, the encoding efficiency of using a mix of I, P, and B frames is beneficial. However, if the playout unit is smaller than a chunk, the interdependence between frames introduces latency. When an ordered transport protocol, such as standard TCP, is used, B frames may depend on frames that have not yet been delivered to the application. Unordered transport protocols, such as TCP Hollywood, exacerbate this issue: earlier frames may not have been received, preventing dependent P and B frames from being decoded. Further, the use of B frames increases encoding latency: they cannot be encoded before the next I frame. While encoding each frame as an I frame would make playout units completely independent, this would significantly reduce quality (where average bitrate is maintained). On balance, I opt to encode each chunk using only I and P frames.

Finally, the applicability of shortening chunks is limited by a **lower bound on chunk duration**. In the architecture described, where the client waits for the chunk to have downloaded in its entirety before beginning to play it out, the client must have a buffer that is large enough

to hold at least one chunk. That is:

$$T_{buffer} \geq T_{chunk} \tag{6.2}$$

For playback to be smooth with a buffer that holds a single chunk, the application must be able to accurately estimate the correct time to request the next chunk (i.e., when the current buffer size equals T_{fetch}), such that the chunk arrives just in time for playback. In reality, changes in network queueing latency and available bandwidth introduce variability into the time taken to retrieve chunks, and make this estimate precarious: stalls will result from the application underestimating the time taken to download a chunk. To reduce the risk of stalling, the application requests the next chunk as the playback of each chunk begins. As a result, additional buffering is required:

$$T_{buffer} \geq T_{chunk} + (T_{chunk} - T_{fetch}) \tag{6.3}$$

This accommodates both the chunk being played out and the requested chunk, less the time taken to fetch the requested chunk. With this buffer in place, stalling is avoided when the playout duration of a chunk is greater than the time taken to fetch that chunk:

$$T_{chunk} \geq T_{fetch} \tag{6.4}$$

As was shown in Figure 6.1, there are two components of T_{fetch}, such that $T_{fetch} = T_{req} + T_{download}$, where T_{req} is the time taken to send the request and begin to receive the response (i.e., a round-trip time), and $T_{download}$ is the time taken to transmit the chunk, where:

$$T_{download} = \frac{S_{chunk}}{R_{bottleneck}} \tag{6.5}$$

The size of each chunk (in octets) can be calculated from its encoding rate and duration:

$$S_{chunk} = R_{encoding} \times T_{chunk} \tag{6.6}$$

This assumes the best-case scenario, where encoding sizes are consistent across chunk durations. As described above, in this application, encoding sizes are maintained at the expense of quality, making this a reasonable approximation for the purposes of this analysis. If quality was maintained, while encoding sizes varied, as shown in Figure 6.2, this approximation would not hold.

From Equations 6.5 and 6.6, the time taken to download a chunk can be expressed in terms of its duration:

$$T_{download} = T_{chunk}\left(\frac{R_{encoding}}{R_{bottleneck}}\right) \tag{6.7}$$

$T_{download}$ is the fraction of the chunk duration, T_{chunk}, that is spent downloading the chunk. If $R_{encoding} \leq R_{bottleneck}$, then chunks are being downloaded at a rate equal to or faster than real-time. Otherwise, that is, when $R_{encoding} > R_{bottleneck}$, chunks take longer to download than their playout duration; smooth playback is not possible.

Combining Equation 6.7 with T_{req}, and assuming a request takes a single round-trip time, allows T_{fetch} to be expressed as:

$$\begin{aligned} T_{fetch} &= T_{req} + T_{download} \\ &= T_{rtt} + T_{chunk}\left(\frac{R_{encoding}}{R_{bottleneck}}\right) \end{aligned} \tag{6.8}$$

Taking Equations 6.4 and 6.8, the lower bound (above which playback can be continuous) on T_{chunk} for a given round-trip time, encoding rate, and bottleneck link rate (where $R_{bottleneck} > R_{encoding}$) can be calculated as:

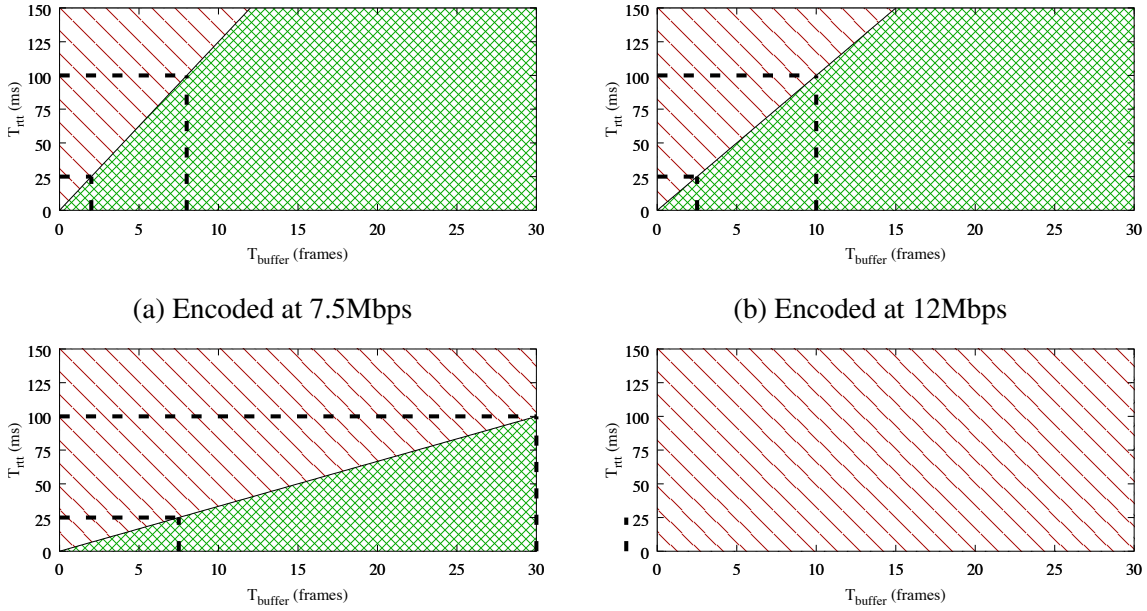
$$\begin{aligned} T_{chunk} &\geq T_{fetch} \\ T_{chunk} &\geq T_{rtt} + T_{chunk}\left(\frac{R_{encoding}}{R_{bottleneck}}\right) \\ T_{chunk}\left(1 - \frac{R_{encoding}}{R_{bottleneck}}\right) &\geq T_{rtt} \\ T_{chunk} &\geq \frac{T_{rtt}}{\left(1 - \frac{R_{encoding}}{R_{bottleneck}}\right)} \end{aligned} \tag{6.9}$$

Equation 6.9 expresses that the duration of the chunk that remains after the time taken to download it has been accounted for must be greater than or equal to the round-trip time, T_{rtt}.

Finally, I combine Equations 6.3 and 6.9 to calculate a lower bound on the size of the playout buffer, T_{buffer}. From Equation 6.3, in the case where the smallest T_{chunk} is selected – that is, where $T_{chunk} = T_{fetch} - T_{buffer}$ must be large enough to hold one chunk:

$$\begin{aligned} T_{buffer} &\geq T_{chunk} \\ T_{buffer} &\geq \frac{T_{rtt}}{\left(1 - \frac{R_{encoding}}{R_{bottleneck}}\right)} \end{aligned} \tag{6.10}$$

Figure 6.4 illustrates this relationship for each of the four encodings of the Big Buck Bunny clip. Here, I set $R_{bottleneck}$ to 30Mbps in all cases, to simulate a typical UK link [60]. Where a combination of T_{buffer} and T_{rtt} falls into the red striped area, smooth playback is not possible: the buffer is not large enough. As shown, with all other variables equal, higher round-trip times or encoding rates require larger buffers. Further, Figure 6.4d illustrates that



(a) Encoded at 7.5Mbps

(b) Encoded at 12Mbps

(c) Encoded at 24Mbps

(d) Encoded at 53Mbps

Figure 6.4: Minimum buffer durations (T_{buffer}), at various encoding rates, required for smooth playback over a 30Mbps link

the case where $R_{encoding} > R_{bottleneck}$: playback cannot be smooth, regardless of the buffer duration.

The analysis presented so far makes two assumptions: (i) that there are no other sources of delay at the sender or receiver, and (ii) that links are lossless. For (i), as described in Chapter 2, there are other application layer delays, such as packetisation at the sender and decoding at the receiver. However, these are difficult to model, and are highly dependent on the libraries used, and the level of optimisation carried out in the application. I do not attempt to model these as part of the analysis, but this does not impact the validity of the broad relationships identified.

For (ii), clearly Internet links are lossy. In Section 6.1.3, I will update the analytical model presented here to include loss, showing that the interactions between standard TCP and loss result in significant latency overhead. Broadly, this significantly increases the minimum values of T_{chunk} and T_{buffer} discussed so far.

6.1.2 Evaluations

In the previous section, I described the minimum chunk durations required to maintain smooth playback. I define smooth playback to be the case where playback is continuous: each chunk is available in the buffer at the time it is to be played out. If a given chunk is *not* in the buffer at its playback time, then the application will *stall*: that is, playback is

paused, and resumes with the delayed chunk upon its arrival. While I will analyse the impact of stalling behaviour in Section 6.3, I note that, when standard TCP is used at the transport layer, stalling delays are cumulative across the entire session: each chunk's playout is delayed by the sum of all of the previous stall durations.

Given this definition of smooth playback, my analysis identifies a boundary between high levels of stalling (i.e., where chunks are too small – the red hatched area in the diagrams in this Chapter), and low or zero levels of stalling (i.e., chunks are large enough – the green hatched areas). For a given round-trip time, there are chunk durations that are too small to be sustained without continuous stalling, punctuated by playback of the chunks, given that it takes longer to fetch a chunk than it does to playout the chunk.

In this section, I describe simulations carried out to validate the analysis presented in Section 6.1.1.

Testbed Setup

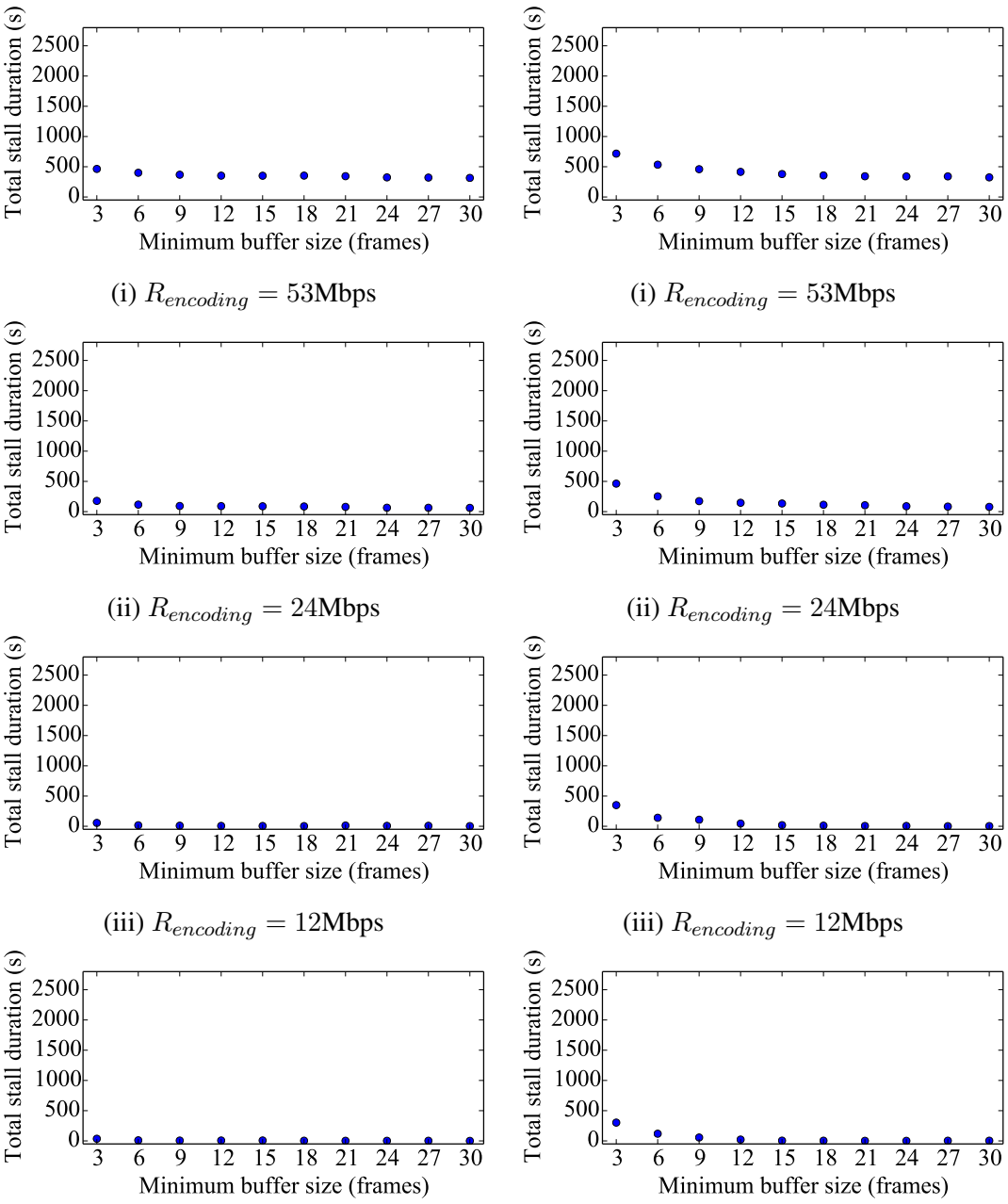
To validate the proposed application-level changes, I use an HTTP/2 simulator. While the simulator exchanges data that mimicks HTTP/2's behaviour (as described in Section 6.1), no HTTP/2 traffic is sent. The simulator's server and client operate the request-response model described, with the client requesting the next chunk as the playback of the current chunk begins. The client requests the manifest file, parses it, and uses this to request video chunks (encoded at the rates described in Section 6.1). As noted earlier, no rate adaptation is applied, and all chunks are encoded at the same rate throughout a given session. In addition, no decoding takes place at the receiver, given that the performance of this process would be highly dependent on the hardware on which the simulator runs. Therefore, the relative trends, rather than the absolute values, are of interest. Further, each simulation is run 5 times with the same parameters; where values are plotted, the mean is given, with error bars showing the standard error across the 5 runs.

The HTTP/2 simulator runs on top of Mininet[3], which emulates the different network conditions described.

Results

Figure 6.5 shows the *total stall durations* for various chunk durations (from 3 frames to 30 frames, in 3 frame intervals) at 25ms and 100ms RTTs. The total session duration is 3 minutes. Total stall duration, as described above, is a measure of the smoothness of playback. These plots validate the analysis presented in Section 6.1.1: increased media encoding rates,

[3] `http://mininet.org`



(i) $R_{encoding} = 53$Mbps

(i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps

(ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps

(iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(iv) $R_{encoding} = 7.5$Mbps

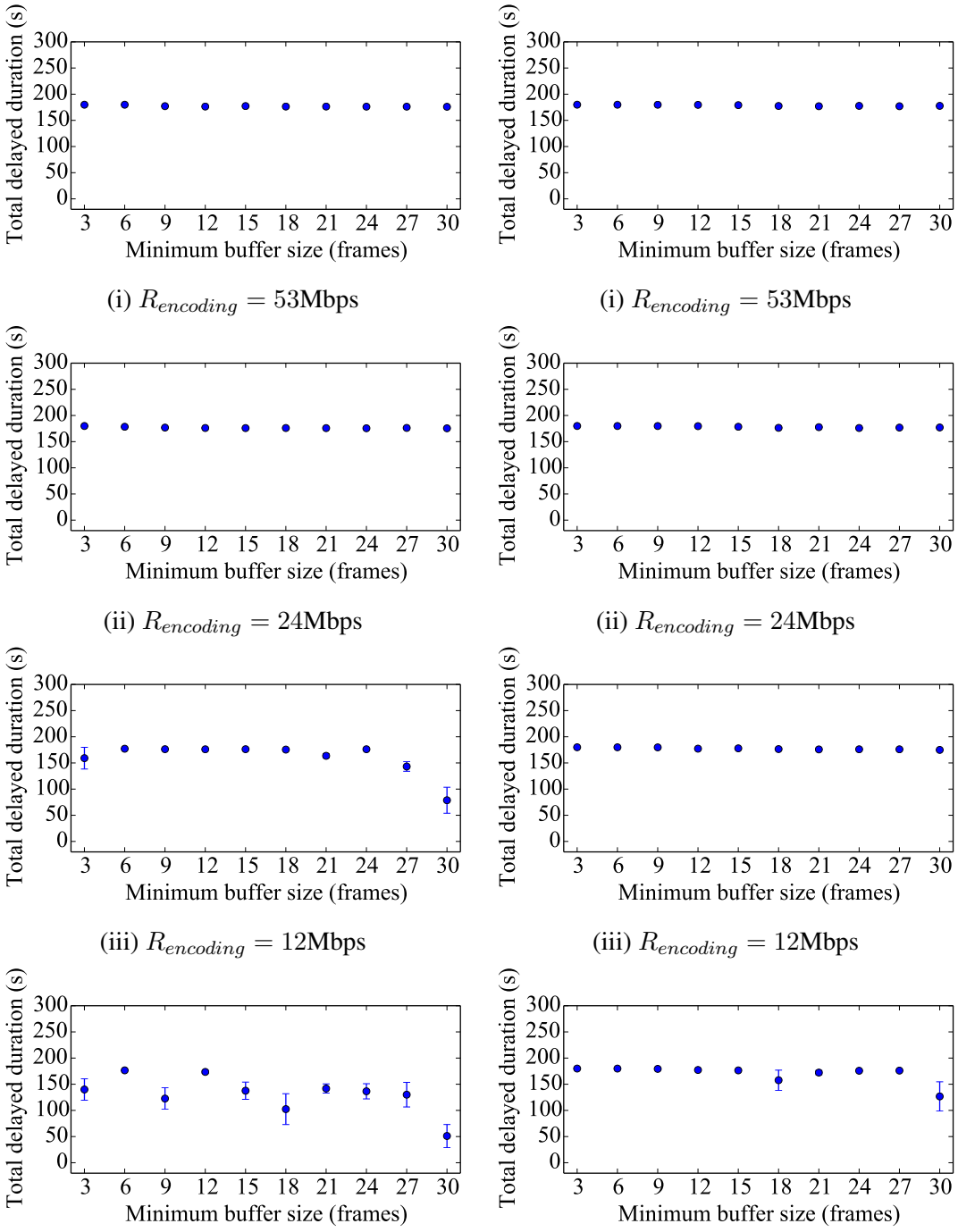
(a) 25ms RTT

(b) 100ms RTT

Figure 6.5: Total stall durations for various chunk durations in a simulated MPEG-DASH application, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossless 30Mbps link with (a) 25ms and (b) 100ms RTT

$R_{encoding}$, with a fixed bottleneck rate, $R_{bottleneck}$, or higher RTTs, result in more stalling at the same chunk durations. While error bars are plotted in Figure 6.5, they are mostly too small to be visible in the plot. The simulations highlight that the boundaries identified by the analysis are not absolute: stalling durations decrease as chunk durations increase.

The slope, peaking at the lowest buffer sizes (and so chunk durations) corresponds with the



(i) $R_{encoding} = 53$Mbps (i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps (ii) $R_{encoding} = 24$Mbps

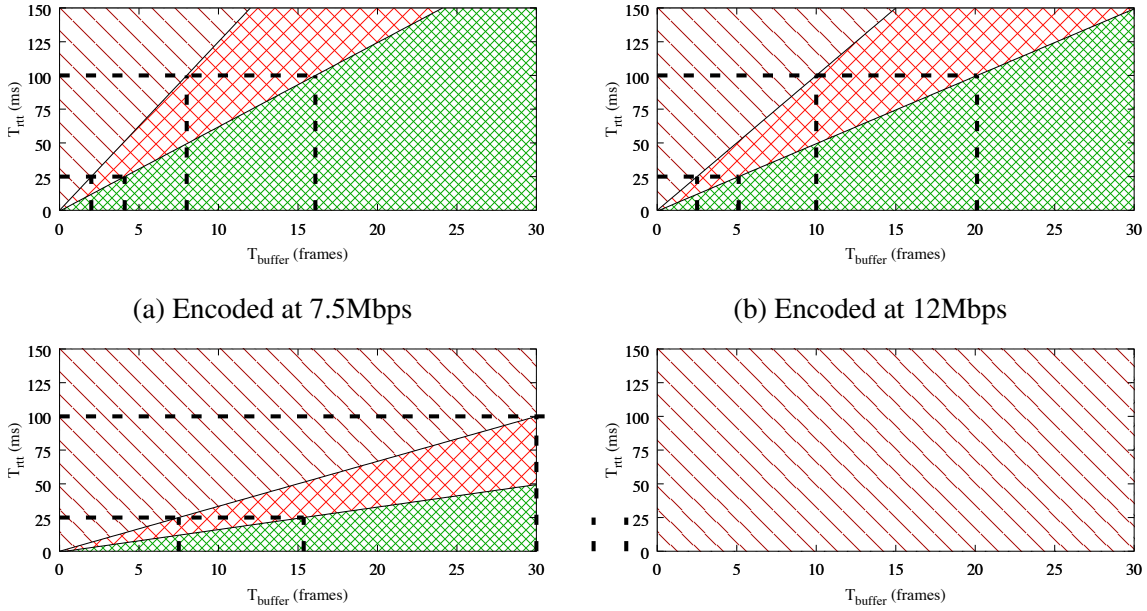
(iii) $R_{encoding} = 12$Mbps (iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps (iv) $R_{encoding} = 7.5$Mbps

(a) 25ms RTT (b) 100ms RTT

Figure 6.6: Total delayed frame durations for various chunk durations in a simulated MPEG-DASH application, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossless 30Mbps link with (a) 25ms and (b) 100ms RTT

same slope shown in Figure 6.3. Encoding very small chunks – in the order of a few frames – results larger total encoding sizes, and in greater variation in the duration of individual chunks.



(a) Encoded at 7.5Mbps

(b) Encoded at 12Mbps

(c) Encoded at 24Mbps

(d) Encoded at 53Mbps

Figure 6.7: Minimum buffer durations (T_{buffer}), at various encoding rates, required for smooth playback over a *lossy* 30Mbps ADSL link

As noted, stalling introduces cumulative delay: after a stall, all subsequent chunks are delayed vs. their original playout times. Figure 6.6 plots the *total duration of the delayed frames*. This illustrates the wider impact of stalling on low-latency applications, where it is important that frames meet their playout times. As shown, at high bitrates and round-trip times, the duration of delayed frames is significant. These plots also highlight that even with the minimal stalling durations seen at lower bitrates and round-trip times, the number of frames that are delayed can be significant. While there is no induced loss in these evaluations, variations in the size of each chunk (as discussed above), combined with small timing variations in the evaluation testbed (which uses Mininet and simulates playback) introduces sufficient variation in when the first stall occurs.

6.1.3 Modelling the impact of packet loss

So far, both the analysis and simulations have assumed that connections are lossless, allowing for the impact of chunk durations to be modelled in isolation. However, on the Internet, congestive loss and packet reordering are not uncommon. As discussed in Chapter 4, standard TCP's reliability and ordering guarantees amplify the impact of segment loss. When a segment is lost, time is required for the sender to detect and retransmit the lost segment. Further, TCP's ordering guarantee results in transport layer head-of-line blocking: segments are delayed waiting for the receipt of earlier delayed or retransmitted segments.

In standard TCP, a sender will classify a segment as lost, and send a retransmission, on receipt of three duplicate acknowledgements. That is, the time taken to retransmit a lost segment is:

$$T_{rexmit} = T_{rtt} + 3 \times T_{mtu} \tag{6.11}$$

where T_{mtu} is the time taken to transmit a single MTU-sized TCP segment. This definition of T_{rexmit} assumes that segments are sent back-to-back, and that retransmissions result from TCP's fast retransmit mechanism. From this, the time taken to fetch a chunk can be revised to include the time taken to retransmit a lost segment:

$$T_{fetch} = T_{req} + T_{download} + T_{rexmit} \tag{6.12}$$

This definition of T_{fetch} assumes that a single loss event (i.e., the loss of one TCP segment or group of consecutive segments) occurs per chunk. With this definition of T_{fetch}, the bound on T_{chunk} given in Equation 6.9 can be revised to:

$$T_{chunk} \geq \frac{T_{rtt} + T_{rexmit}}{\left(1 - \frac{R_{encoding}}{R_{bottleneck}}\right)} \tag{6.13}$$

From this, the lower bound on T_{buffer}, given in Equation 6.10 (for the lossless case), becomes:

$$T_{buffer} \geq \frac{T_{rtt} + T_{rexmit}}{\left(1 - \frac{R_{encoding}}{R_{bottleneck}}\right)} \tag{6.14}$$

This relationship is illustrated in Figure 6.7. In this diagram, the green hatched area represents the valid (i.e., where playback can be smooth) buffer durations in a lossy network. Both the dark and light red regions show the combinations of buffer duration and RTT where playback cannot be smooth; the dark red hatched region highlights those combinations that have become infeasible as a result of including loss (i.e., the difference from Figure 6.4). As shown, the additional T_{rtt} added by T_{rexmit} results in larger buffers being required to accommodate the possibility of a loss and the subsequent retransmission for each chunk.

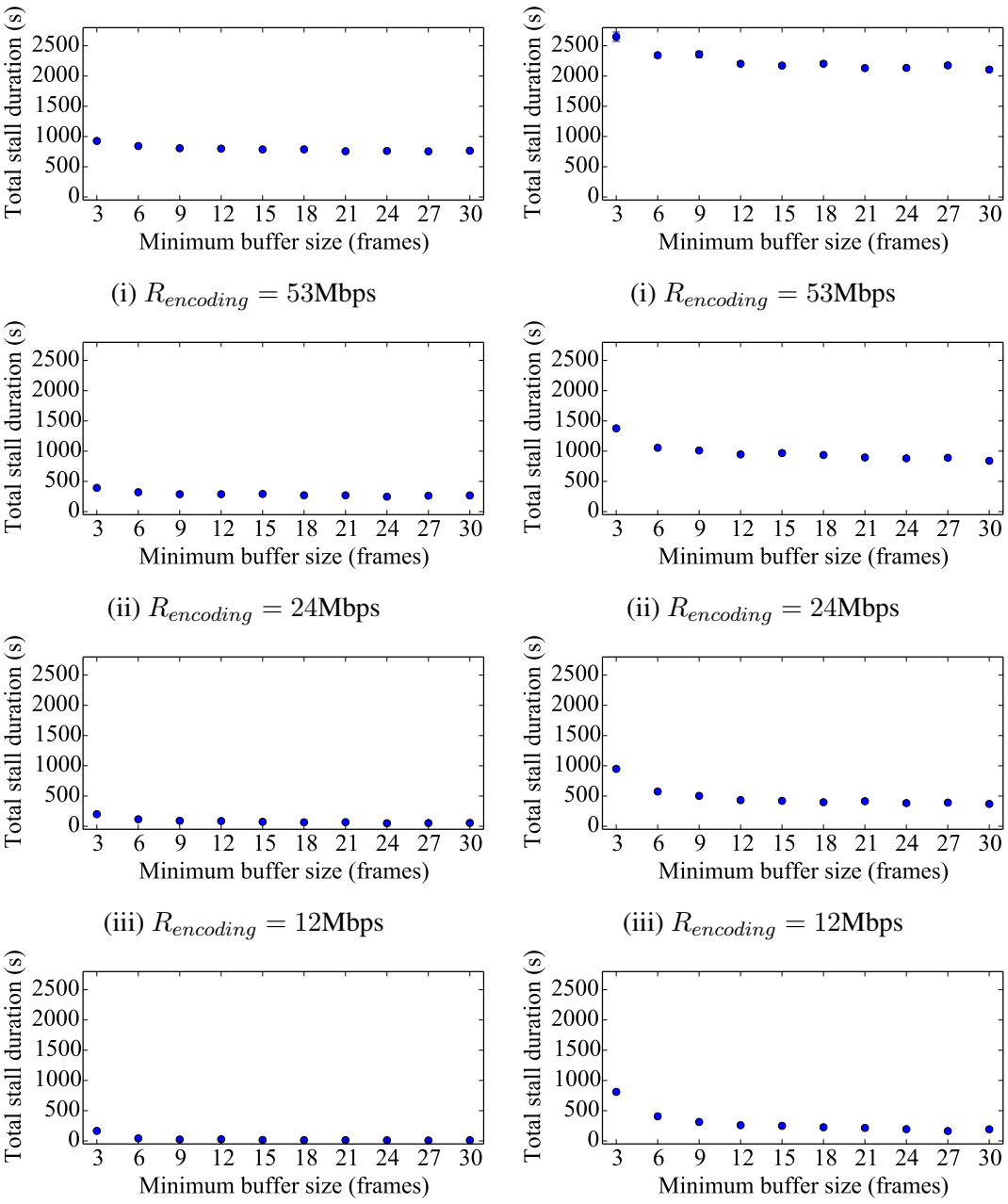
This analysis makes several assumptions that are worth noting. First, I include only a single T_{rexmit}, covering only one loss event (possibly comprised of multiple consecutive segments). Given that the analysis considers relatively small chunk sizes, I expect that this is a reasonable assumption. If the loss rate is higher, then more buffering would be required at the receiver to allow for smooth playback. Next, the analysis assumes that all chunks are comprised of at least three segments. This does not hold for very small chunks encoded at low bitrates, where each frame could be less than three segments in size. However, the analysis

presented in Section 6.1.1 demonstrated that such chunk sizes are too small, even before loss is considered. Additionally, I assume that all of the TCP segments that comprise a chunk are sent back-to-back, without being paced [2]. While pacing may impact the performance of each chunk download (i.e., T_{fetch}), the conclusions of this analysis hold where $T_{chunk} \geq T_{fetch}$: any mechanism that changes T_{fetch} is tangential, so long as this bound is met. Finally, I assume that loss is not at the tail of the series of TCP segments that make up the chunk. Where tail loss occurs, a fast retransmit (i.e., a triple duplicate ACK) is unlikely to be triggered: there aren't three TCP segments to be transmitted to induce duplicate acknowledgements. Instead, an RTO timeout will trigger a retransmission. Typically, this will be substantially slower: RTOs are generally configured in the order of seconds. However, they can be tuned to lower values, to match application behaviour. Tuning the RTO to approximate T_{rexmit} allows the analysis to model these situations. However, low RTOs are likely to have other affects; considering these is outside the scope of this thesis.

In Figure 6.8, I plot the total stall durations that result from repeating the simulations described in Section 6.1.2, but with 0.2% random packet loss on the bottleneck link. As shown, when compared with Figure 6.5, the introduction of packet loss significantly increases the total stall duration for a given buffer duration. Figure 6.9 plots the total delayed frame durations for the same experiment; as shown, most frames are played out at some delay, due to stalls occurring early in the stream. These plots must be interpreted alongside those for stall durations: even small stall durations, as those seen at lower bitrates and round-trip times, delay later frames.

6.2 Application-Layer Improvements

As shown by Equations 6.3 and 6.13, the receive-side buffer has a potentially significant lower bound, given that each chunk must be fully downloaded before playback can begin. To reduce the receive-side buffer below this bound, I evaluate a *progressive streaming* architecture: clients request each chunk as usual (i.e., as playback of the previous chunk in the sequence begins), but the server will deliver each video frame within a separate HTTP/2 stream, and the client will begin to playback the chunk as soon as the first frame has arrived. This allows the client to have a buffer that is sized in multiples of T_{frame}, rather than T_{chunk}. This architecture is illustrated in Figure 6.10. As shown, this mechanism uses HTTP/2 server push; this allows a single client request to be fulfilled by multiple streams. Server push involves sending a push promise – essentially mimicking the request that the client would have sent for the data that will be pushed – followed by the HTTP/2 frames corresponding to the push promise. Under this architecture, the push promise is sent on the stream opened by the client by the request for the chunk, with the data for each video frame pushed on a separate



(i) $R_{encoding} = 53$Mbps

(i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps

(ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps

(iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(iv) $R_{encoding} = 7.5$Mbps

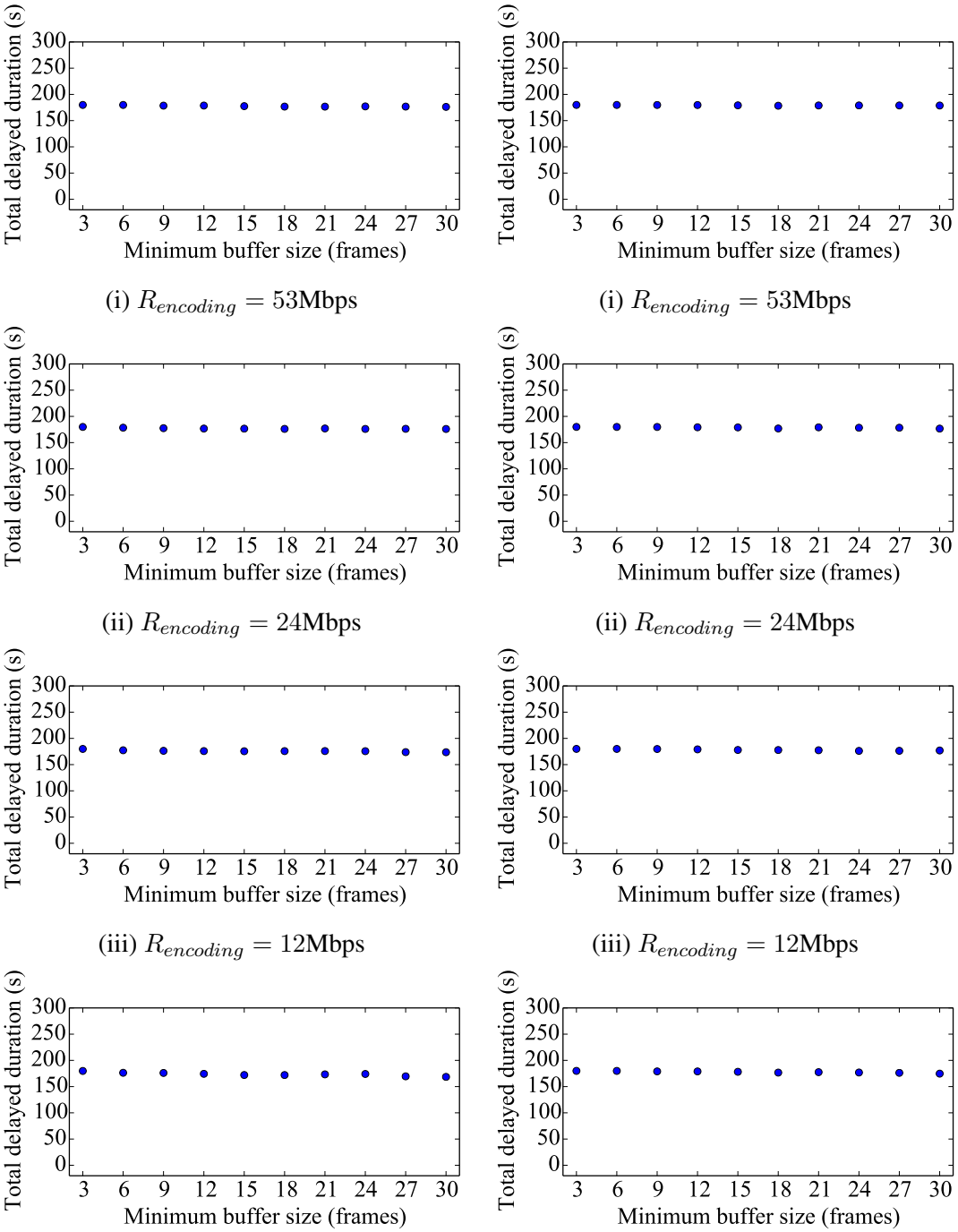
(a) 25ms RTT

(b) 100ms RTT

Figure 6.8: Total stall durations for various chunk durations in a simulated MPEG-DASH application, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossy 30Mbps link with (a) 25ms and (b) 100ms RTT

stream.

In Section 6.2.1, I update the analysis presented in the previous section to reflect the progressive streaming architecture. Section 6.2.2 validates this analysis using simulations. Throughout, I assume (as discussed in Section 6.1.3) that TCP segment loss is possible, and factor this in to both the analysis and simulations.



(i) $R_{encoding} = 53$Mbps

(i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps

(ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps

(iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(a) 25ms RTT

(b) 100ms RTT

Figure 6.9: Total delayed frame durations for various chunk durations in a simulated MPEG-DASH application, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossy 30Mbps link with (a) 25ms and (b) 100ms RTT

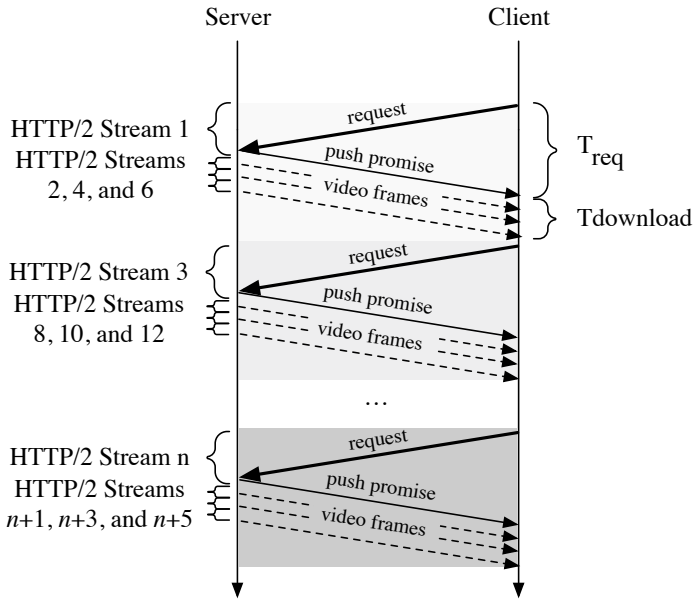


Figure 6.10: MPEG-DASH progressive streaming architecture: chunks are requested as normal; separate video frames delivered for immediate playback

6.2.1 Analysis

Under the progressive streaming architecture, the time taken to request and download a chunk, T_{fetch} (defined in Equation 6.12), stays the same. However, the time to download the chunk, $T_{download}$, can be split into two parts: the time taken to download first frame (an I frame), and that taken to download the remaining P frames.

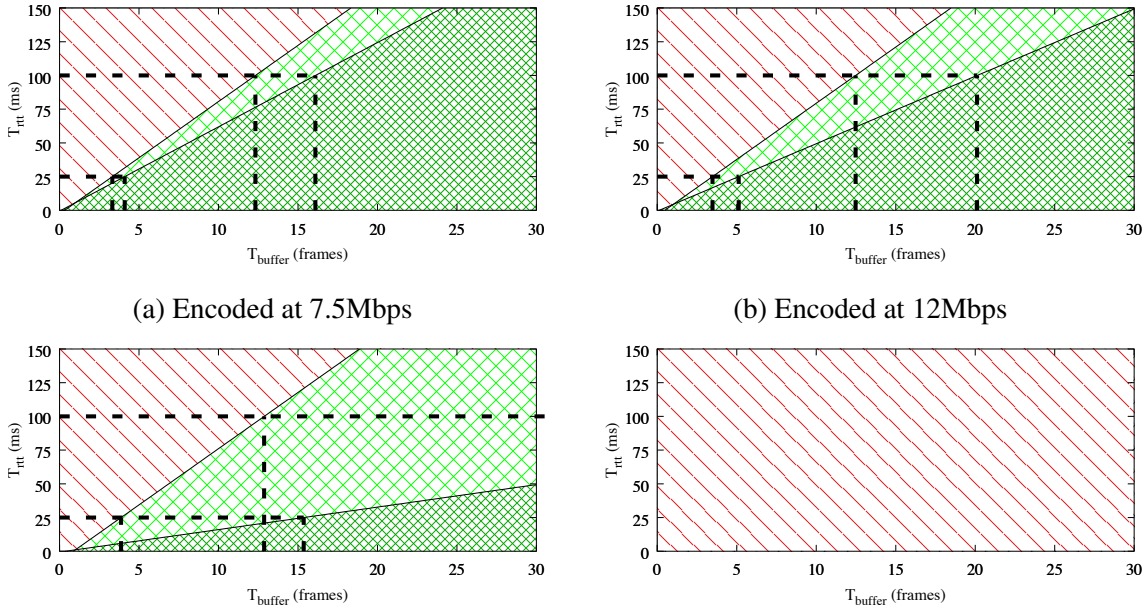
To model this, I define S_{iframe} and S_{pframe} as the size (in octets) of I and P frames, respectively, and F_{chunk} as the number of frames in each chunk. To simplify the analysis, I assume that each chunk is comprised of a single I frame, followed by $F_{chunk} - 1$ P frames. From this, $N_{fragments}$, the number of buffer-sized fragments that the chunk is divided into, can be calculated as $\frac{F_{chunk}-1}{n}$, where n is the number of frames that the buffer holds.

Using this notation, the time taken to download a chunk, $T_{download}$, can be redefined as:

$$T_{download} = \frac{S_{iframe}}{R_{bottleneck}} + \sum_{i=1}^{N_{fragments}} \frac{S_{pframe} \times n}{R_{bottleneck}} \tag{6.15}$$

This is the time taken to download a single I frame, and all of the remaining playout buffer-sized fragments of the chunk (which are comprised of P frames).

Given that chunks are still requested when the previous chunk's playback begins, the constraint that $T_{chunk} \geq T_{fetch}$ remains. However, T_{buffer} is no longer bounded by T_{chunk}: this is the basis for the latency improvements that progressive streaming provides. The buffer must be large enough to accommodate the chunk request, the download of a single frame,



(a) Encoded at 7.5Mbps

(b) Encoded at 12Mbps

(c) Encoded at 24Mbps

(d) Encoded at 53Mbps

Figure 6.11: Minimum buffer durations (T_{buffer}), at various encoding rates, required for smooth playback using the progressive streaming architecture over a *lossy* 30Mbps ADSL link

and a retransmission time:

$$T_{buffer} \geq T_{rtt} + \frac{S_{iframe}}{R_{bottleneck}} + T_{rexmit} \tag{6.16}$$

To simplify the analysis, I approximate S_{iframe} as:

$$S_{iframe} = \frac{R_{encoding} \times T_{chunk}}{F_{chunk}} \tag{6.17}$$

This assumes that *all* frames in a given chunk have the same size. As discussed in Section 6.1.1, this is unlikely: I frames are typically significantly larger than P frames. However, this assumption is safe: while the size of the single I frame that the chunk contains is underestimated, the size of *all* of the remaining P frames is overestimated. As a result, this analysis models the worst case: most frames will be smaller than assumed. Further, making a better approximation of S_{iframe} is difficult: the comparative sizes of I and P frames is highly dependent on encoding parameters, including the duration of the chunk, the codec, and the hardware used.

Figure 6.11 plots the lower bound on T_{buffer} specified in Equation 6.16. Where a combination of T_{buffer} and T_{rtt} are in the green hatched regions, playback can be smooth under the progressive streaming architecture. The dark green hatched region represents those combinations where playback would have been smooth under the request-response model (as

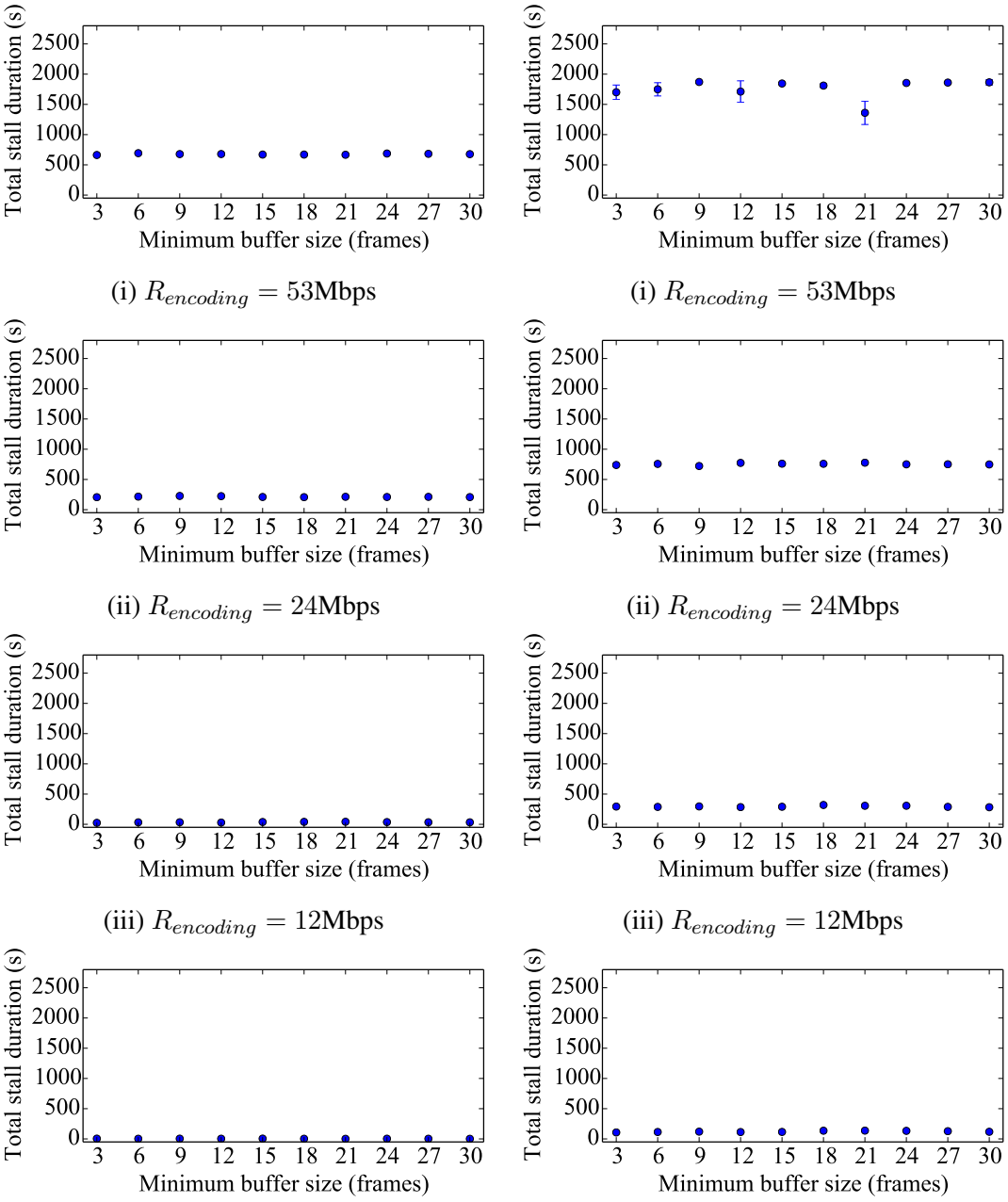
illustrated in Figure 6.7); the light green hatched region shows those combinations that are made viable by the progressive streaming architecture.

There are two notable changes to the relationship between T_{rtt} and T_{buffer} when comparing the request-response and progressive streaming architectures, as illustrated in Figure 6.11. Firstly, the impact of increasing the encoding rate ($R_{encoding}$) relative to the bottleneck link rate ($R_{bottleneck}$) is much less pronounced: this is due to the buffer storing a single frame, rather than the entire chunk. Finally, as shown in Figure 6.11d, the progressive streaming architecture does not result in buffering being able to create smooth streaming where $R_{encoding} > R_{bottleneck}$. Given that it takes longer to download a frame than it does to play it out, playback will not be smooth for any buffer duration.

6.2.2 Evaluations

Figure 6.12 revises Figures 6.5 and 6.8, performing the same simulations, but with the simulated HTTP/2 application progressively streaming. As noted in the previous section, chunk duration and playout buffer size have been decoupled. For these simulations, I set the chunk duration to 1 second. As shown, for all buffer durations, and at both round-trip times, progressive streaming results in lower total stall durations. Decoupling chunk duration and playout buffer size has the impact of reducing the variation in stalling duration across minimum playout buffer sizes: while the *minimum* buffer size varies, when the chunk is requested, the entire chunk is sent to the client; the amount of buffering, therefore, is likely to be significantly higher than the minimum given. Figure 6.12 shows that, at all bitrates and round-trip times, the progressive streaming architecture reduces total stall durations vs. the request-response architecture (Figure 6.8). Figure 6.13 shows the total delayed frame durations for the progressive streaming architecture. The delayed frame duration is generally lower in this architecture. The difference is significant at lower encoding rates and round-trip times. At higher rates, the difference is small, given that this metric is dictated by the time at which the first stall occurs.

However, these results show that stalling still occurs. As discussed earlier, this is problematic for low-latency applications, which are tolerant of *some* loss at the expense of timeliness. Standard TCP does not afford applications the choice of how much loss is tolerable: clients will receive all the data they request, in-order. This prevents applications from *skipping* missing data, removing the cumulative impact of stalling waiting for lost data to arrive. Enabling this playback mode – of skipping rather than stalling – requires using a different transport protocol. In the next section, I explore how TCP Hollywood can be used.



(i) $R_{encoding} = 53$Mbps

(i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps

(ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps

(iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(iv) $R_{encoding} = 7.5$Mbps

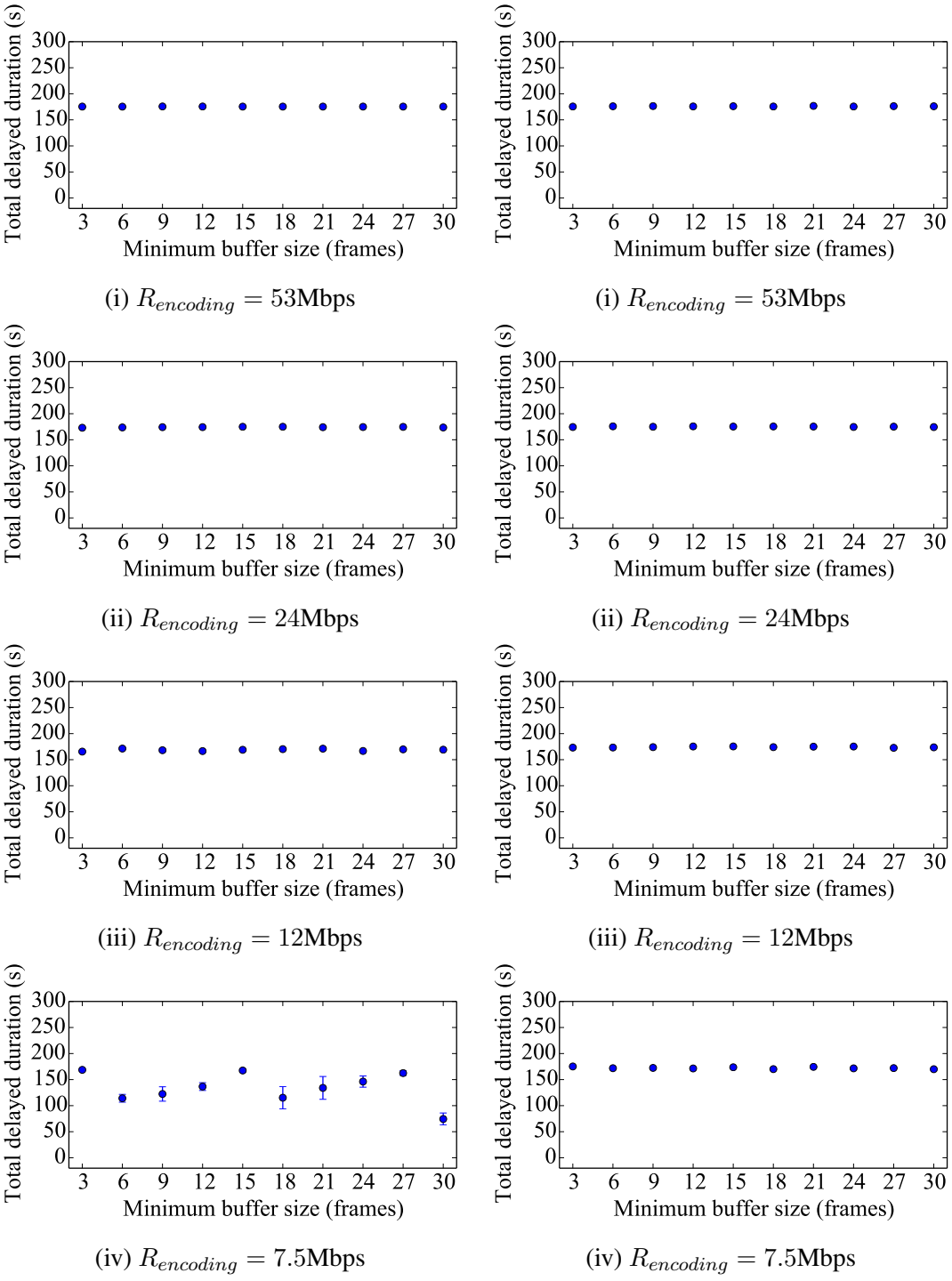
(a) 25ms RTT

(b) 100ms RTT

Figure 6.12: Total stall durations for various chunk durations in a simulated MPEG-DASH application *with progressive streaming*, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossy 30Mbps link with (a) 25ms and (b) 100ms RTT

6.3 Adapting to TCP Hollywood

As shown in the previous section, there can be a significant lower bound on chunk duration when relying upon application layer modifications alone. Ultimately, the size of chunks is bounded by the desire to guarantee smooth playback in the event of TCP segment loss.



(i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(a) 25ms RTT

(i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(b) 100ms RTT

Figure 6.13: Total delayed frame durations for various chunk durations in a simulated MPEG-DASH application *with progressive streaming*, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossy 30Mbps link with (a) 25ms and (b) 100ms RTT

When loss occurs, T_{rexmit} adds at least one T_{rtt} of delay. The analysis presented so far has assumed that applications wish to accommodate this by having larger chunks or by allowing

for *stalling* when loss occurs. Here, I define a stall to be the period of time when playback is paused, waiting for data to arrive, after which playback resumes with the delayed data. Stalling in this way is problematic for low-latency applications: stall durations are cumulative, with total play-out delay growing as the session progresses.

However, the low-latency applications that are the focus of this thesis would prefer not to stall: they prefer timeliness to reliability. Instead, these applications are better suited to *skipping*, where playback pauses for the duration of the missing data, but resumes with the next available frame. By resuming with the next available frame, rather than with the missing data on its arrival, the application can trade-off reliability – i.e., a complete video stream, with no missing frames – for timeliness: the data that is available will be played out within the application's time constraints.

In Section 6.3.1, I will analyse the impact of stalling on applications, and then compare this with skipping. I will validate this analysis with evaluations in Section 6.3.2.

6.3.1 Analysis

I begin by modelling the impact of stalling behaviour. Over the entire session, there are n stalls; T_{stall_i} is the duration of the ith stall. I also define the total stalling duration of the application up to and including the jth stall as:

$$T_{offset_j} = \sum_{i=1}^{j} T_{stall_i} \tag{6.18}$$

After the jth stall, T_{offset_j} is the sum of all of the stalls to that point – this is equivalent to the *total delayed frame* duration metric plotted in Figures 6.6, 6.9, and 6.13. Importantly, T_{offset} refers to the deviation between the session's playback point, and the live point as defined by the application (i.e., where playback would be if no loss had occurred). Drifting from the live point is not suitable for the low-latency applications targeted by this thesis: delay accumulates over the session, meaning that liveness and interactivity decay over time.

For the low-latency applications I consider, skipping behaviour would be preferred. In this architecture, playback pauses for the duration of any missing data, and then resumes with the next available data. Effectively, this means that T_{offset} at any given time is 0: the duration of pauses is not cumulative. However, this comes at the expense of the missing data: lost chunks are never played out. Given the target applications, this is a reasonable trade-off. Liveness and interactivity take priority over the minimal impact of a low level of loss.

Switching from stalling to skipping is not possible over standard TCP. As discussed in Chapter 4, head-of-line blocking at the transport layer means that, in standard TCP, TCP segments that arrive after a lost or out-of-order segment are not delivered to the application until the

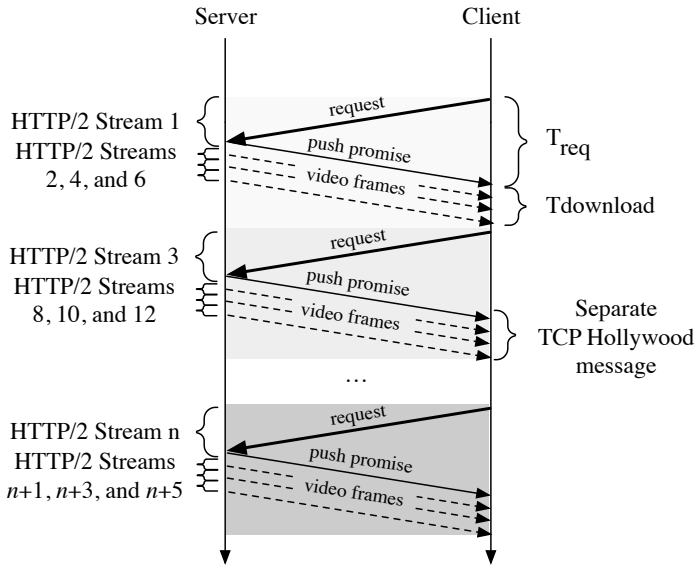
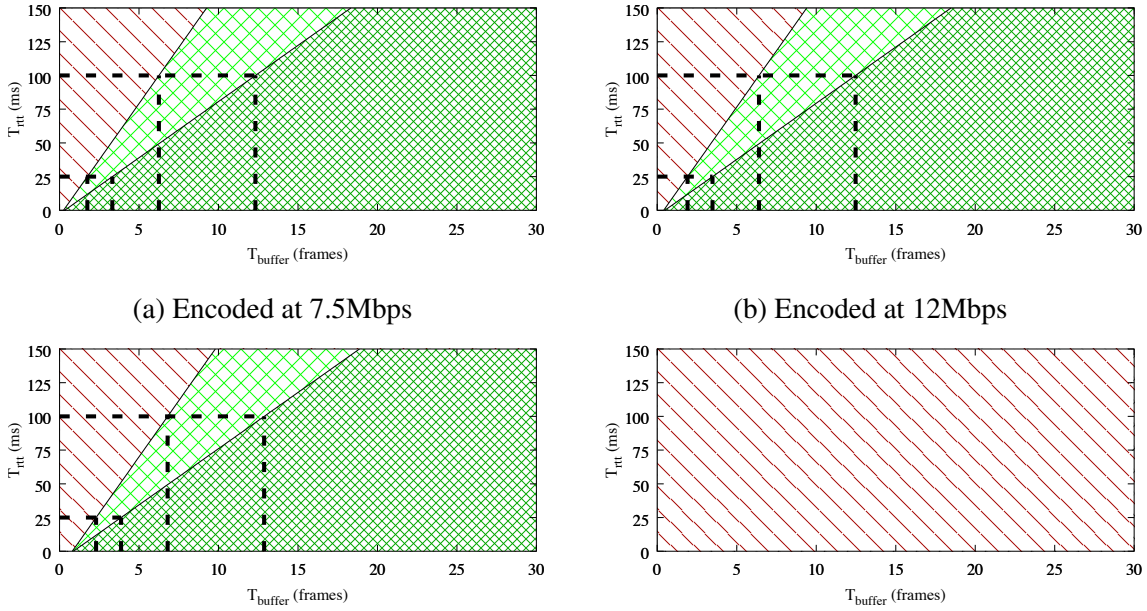


Figure 6.14: MPEG-DASH progressive streaming architecture over TCP Hollywood: video frames are sent in separate streams *and* TCP Hollywood messages

missing segment arrives. This means that in the event of loss, the application has no later data to which it can skip: it has no option but to incur the delay that results from head-of-line blocking, regardless of its preferred trade-off between loss and delay.

Removing head-of-line blocking requires a different transport layer protocol. Here, I will use TCP Hollywood; however, the approach presented in this chapter is largely applicable to other transport protocols, including QUIC. As discussed in Chapter 4, TCP Hollywood provides a partially-reliable unordered messaging abstraction: the application sends messages that are retransmitted within a specified lifetime, and these messages are delivered in the order they arrive. Importantly, this means that the head-of-line blocking is eliminated between messages: the loss of one message does not, at the transport layer, affect the delivery of other messages.

Figure 6.14 illustrates how an MPEG-DASH application can be mapped to the TCP Hollywood delivery model, to maximise the benefit of eliminating transport layer head-of-line blocking. As under the progressive streaming architecture, the client sends a request for each chunk, at the rate that it determines. The server responds by pushing each video frame on separate HTTP/2 streams. In contrast with the progressive streaming architecture, with TCP Hollywood, a separate *message* is used to hold the three components that a single video frame is mapped to: (i) the push promise that mimics the request that the client would have sent for the frame; (ii) the header for the pushed data; and (iii) the data itself. By combining these HTTP/2 elements into a single TCP Hollywood message, the semantic integrity of HTTP/2 is maintained: either the client receives all of these elements, or none of them. In addition, it is not possible for the client to receive these elements out-of-order, which would ordinarily be possible under TCP Hollywood's out-of-order delivery model. The benefit of using TCP



(a) Encoded at 7.5Mbps

(b) Encoded at 12Mbps

(c) Encoded at 24Mbps

(d) Encoded at 53Mbps

Figure 6.15: Minimum buffer durations (T_{buffer}), at various encoding rates, required for playback using the progressive streaming architecture over TCP Hollywood over a *lossy* 30Mbps ADSL link

Hollywood messages is that not only is application layer head-of-line blocking eliminated (by using separate streams), but transport layer head-of-line blocking is also eliminated.

Under this architecture, the previous lower bound on T_{buffer}, given in Equation 6.16, can be revised to remove T_{rexmit}:

$$T_{buffer} \geq T_{rtt} + \frac{S_{iframe}}{R_{bottleneck}} \tag{6.19}$$

Beyond this modification, it is worthwhile to note that the role of this lower bound on T_{buffer} changes when using TCP Hollywood. Rather than this being a lower bound that is required to be met for smooth playback (i.e., without stalls), this bound must be met for any playback. The architecture described in this section uses TCP Hollywood to remove the stalling behaviour seen in the request-response and progressive streaming architectures. As a result, instead of stalling, the application will skip to the next available frame. If the lower bound on T_{buffer} given in Equation 6.19 is not met, then all frames will be skipped, and playback cannot proceed. While this difference is noteworthy, it is unlikely to have a material impact on the quality-of-experience for end users, especially for low-latency applications: under the request-response and progressive streaming architectures, consistently not meeting the (higher) lower bounds on T_{buffer} would result in persistent stalling.

Figure 6.15 illustrates the relationship between T_{buffer} and T_{rtt} under the architecture described in this section. Combinations of T_{buffer} and T_{rtt} in the green hatched regions will allow playback to proceed when TCP Hollywood is used, while only those in the *dark* green

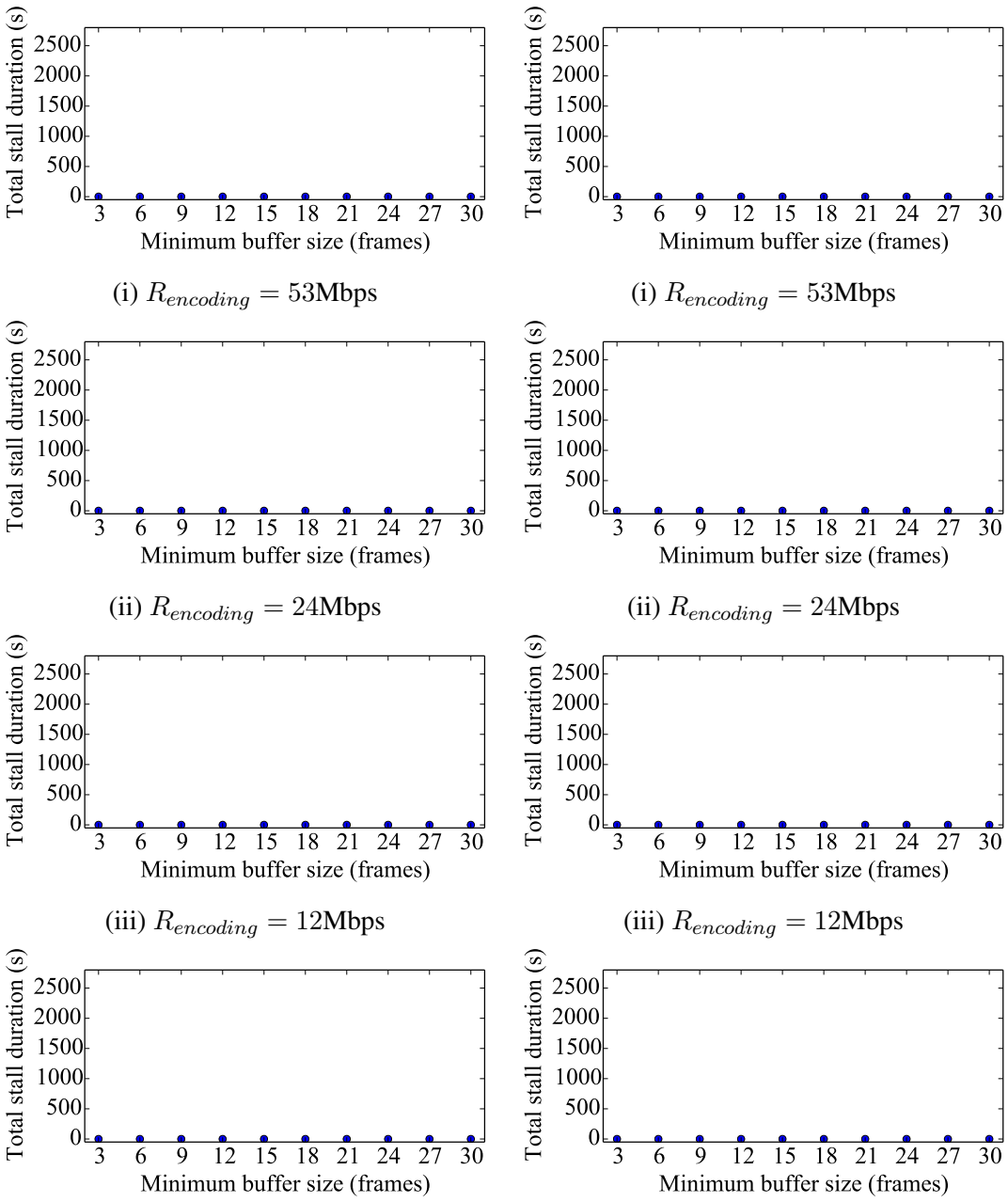
hatched region are viable under the progressive streaming architecture when standard TCP is used. As shown, when the buffer does not have to accommodate loss, the buffer can be sized significantly smaller. As shown in Figure 6.15d, TCP Hollywood does not help in the case where $T_{encoding} > T_{bottleneck}$: frames take longer to download than they do to play-out, resulting in continuous skipping.

6.3.2 Evaluations

Figure 6.16 shows the total stall duration of the application, repeating the experiments discussed in Section 6.2.2, but with TCP Hollywood replacing standard TCP at the transport layer. As shown, the application does not stall: by using TCP Hollywood, it can instead skip the missing data, and resume playback with the next available frame. This difference is crucial for improving performance: skips, rather than stalls (where playback pauses, waiting for the missing data, and then resuming with that data when arrives), are not cumulative: no additional latency is introduced, as the application resumes with the next available frame. This delivery mode is not possible with standard TCP, where data is only made available in-order.

Figure 6.17 plots the total *skip* duration for the same set of evaluations. There are a number of broad conclusions that can be made from this plot, and it is particularly instructive to compare the total skip duration given here with the total delayed frame duration plots given for the other architectures. When compared with Figure 6.13 (which plots the total stall duration under the progressive streaming architecture running above standard TCP), it is clear that at lower encoding rates, round-trip times, and minimum buffer sizes, TCP Hollywood performs better. However, at higher encoding rates, it is likely that the impact of the additional overheads under TCP Hollywood – as discussed in Section 4.2 – is magnified by lower minimum buffer durations. Further implementation work is required to identify and optimise those components of the application and TCP Hollywood that produce this additional delay.

However, at lower bitrates and round-trip times, TCP Hollywood performs much better, with the skipped duration being less than the delayed frame duration under the progressive streaming mode over standard TCP. When comparing skip and stall durations, it is worthwhile to note the difference in their impact on application quality-of-experience. Where a frame is skipped, it is not displayed to the user: it is effectively lost. Where the application stalls, all frames are played out, but with a potentially significant delay, versus the time that those frames *should* have been played out. This difference matters for the low-latency applications that are the focus of this thesis, where a frame that is played out at a significant delay (versus its desired playout time) *is* effectively lost. This main argument put forward by this thesis is that enabling applications to make this trade-off (between delay, loss, and quality-of-experience) is desirable, and requires transport layer support.



(i) $R_{encoding} = 53$Mbps

(i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps

(ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps

(iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps

(iv) $R_{encoding} = 7.5$Mbps

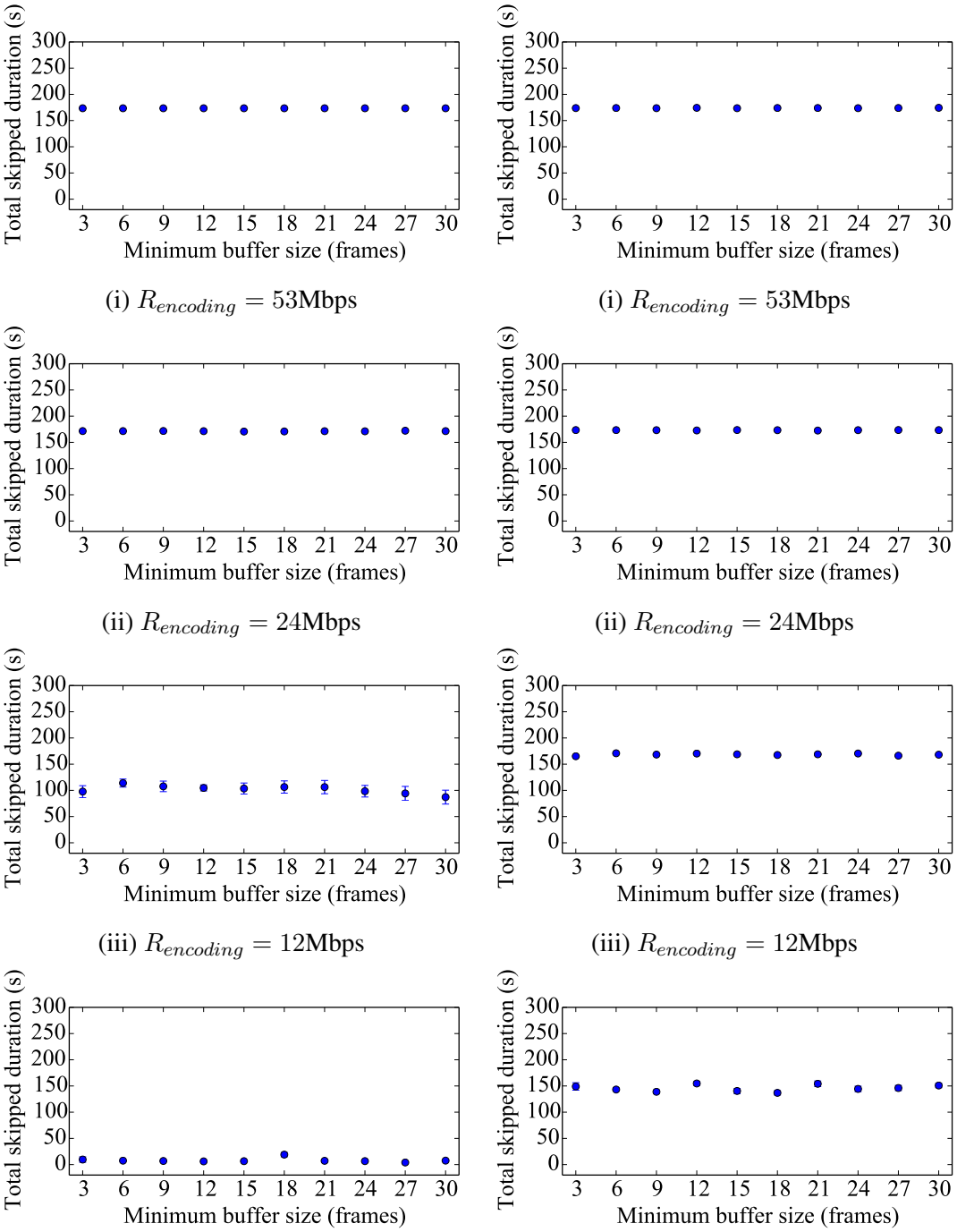
(a) 25ms RTT

(b) 100ms RTT

Figure 6.16: Total stall durations for various chunk durations in a simulated MPEG-DASH application with progressive streaming *over TCP Hollywood*, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossy 30Mbps link with (a) 25ms and (b) 100ms RTT

6.4 Summary

In this chapter, I have described the set of modifications that are required to enable an MPEG-DASH application to make use of TCP Hollywood. In particular, I have shown that by



(i) $R_{encoding} = 53$Mbps (i) $R_{encoding} = 53$Mbps

(ii) $R_{encoding} = 24$Mbps (ii) $R_{encoding} = 24$Mbps

(iii) $R_{encoding} = 12$Mbps (iii) $R_{encoding} = 12$Mbps

(iv) $R_{encoding} = 7.5$Mbps (iv) $R_{encoding} = 7.5$Mbps

(a) 25ms RTT (b) 100ms RTT

Figure 6.17: Total skip durations for various chunk durations in a simulated MPEG-DASH application with progressive streaming *over TCP Hollywood*, at encoding rates, $R_{encoding}$ of 7.5Mbps, 12Mbps, 24Mbps, and 53Mbps, over a lossy 30Mbps link with (a) 25ms and (b) 100ms RTT

carefully considering the contents of each TCP Hollywood message, the semantic integrity of upper-layer protocols – in this case, HTTP/2 and MPEG-DASH – can be maintained,

whilst permitting a new delivery model that is desirable for low-latency applications. I have shown that migrating from standard TCP to TCP Hollywood allows the application *skip* missing data, rather than introducing cumulative delay by *stalling*, and discussed the impact that this shift has on the overall quality-of-experience that can be provided by low-latency multimedia applications.

This demonstrates the need for cross-layer approaches to improving performance: complementary modifications to both the application *and* transport layers are required to realise the performance improvements here. As demonstrated by Bhat et al. [4], transport protocol support alone is insufficient, and illustrated by the analysis in this chapter, application layer change is limited by the transport protocol used.

Chapter 7

Transport Services for Low-Latency Multimedia Applications

In Chapter 3, I described that ossification of the transport layer had left UDP and TCP as the only widely deployed and deployable transport protocols. This means that, in order to achieve wide reachability, novel transport protocols are required to use UDP or TCP as substrates. The potential for this approach is clear: TCP Hollywood demonstrates that TCP can be modified to support low-latency applications, while QUIC similarly shows that novel delivery models can be provided by UDP-based protocols. Substrate-based protocol design may lead to an increase in the number of domain-specific protocols, each requiring different metadata from the application. For example, TCP Hollywood requires a deadline for each time-lined message that the application sends. However, the Berkeley Sockets API, developed within the design philosophy that file and network access should have a common interface, has limited expressivity. This API is not suitable if the set of available transport protocols expands to include protocols that require much more metadata from the application, when compared with UDP or TCP. Defining a new API that *does* allow the application to express the metadata required by the transport layer is an obvious first step. However, broader architectural change should also be considered if novel transport protocols are to see wide deployment.

In this chapter, I will describe the architectural principles behind the transport services approach, and formalise the delivery model of TCP Hollywood into the set of transport services that it provides. Further, I will present an abstract API for this set of transport services, set within the context of an augmented Berkeley Sockets API. Finally, I will discuss potential alternative architectures, and consider how they might support the services proposed.

This chapter is structured as follows:

Section 7.1 formalises the delivery model of TCP Hollywood into a set of transport services

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 7.1: Transport services for low-latency applications

that are desirable for low-latency multimedia applications;

Section 7.2 takes the set of transport services that are needed for low-latency multimedia applications, and presents an abstract API, in reference to the Berkeley Sockets API, that might be used to provide them;

Section 7.3 describes how the set of transport services and the abstract API might be realised;

Section 7.4 summarises the chapter.

7.1 Desirable 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 transport services, allowing the system to select an appropriate transport protocol based on application needs. It is hoped that this will loosen the coupling between the application and transport layers, and so facilitate the deployment of new transport services and protocols.

The work in TAPS provides a vocabulary for discussing the components of transport protocols. This vocabulary is useful when discussing the needs of low-latency applications, and the protocols to support them. In this section, I use this to describe the transport services that are required for low-latency applications. Table 7.1 summarises the transport services discussed.

Timing and Deadlines

Timing is the most salient feature of low-latency applications. Since the data that

they send must be conveyed within low-latency bounds, 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 or displaying the video frame) range from tens to a few hundred milliseconds. Non-interactive applications, such as broadcast and on-demand programming, have deadlines in 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, respecting deadlines requires that the packet delivery service provide *partial reliability*. For example, when used to repair loss, the limits of forward error correction imply that, with some probability, a packet will become non-recoverable. By contrast, retransmissions used to recover from loss have potentially unbounded delay, given that any retransmission may itself be lost. Accordingly, a transport protocol that respects deadlines should provide partial reliability, acknowledging that it may be unable to deliver all data by its deadline.

Many low-latency applications run over standard TCP, though this protocol is fully, rather than partially, reliable. TCP's full reliability can lead to playout stalls when the application is blocked by retransmissions that take too long, as demonstrated in the previous chapter. These stalls are one of the primary causes of poor user experience in streaming applications. For the applications that are the focus of this thesis, 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 if it relies upon 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 [50] divides video frames into three types. I frames are independently encoded, while P and B frames contain only the changes since the previous frame (P), or between frames (B), and so can only be decoded on the successful arrival of other frames. Newer codecs, such as H.264 [82], 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 [13] to make the best use of payload data. At the transport layer, this implies a *message oriented* service that maintains application data unit boundaries. Messages are delivered to the application in the order they arrive. As seen in standard TCP, in-order delivery can introduce significant latency: incoming segments may be head-of-line blocked waiting for the delivery of an earlier segment.

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 streams. 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

I note the importance of congestion control. Historically, low-latency 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 bitrate, an increasing number are either, or both, adaptive and variable bitrate. 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.

I note that a connection-oriented service is a lesser requirement for many low-latency multimedia applications. Indeed, flexibility to change the destination within a VoIP call, for example, 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 the 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 when it can be torn down. Accordingly, it is often desirable for the transport protocol to be connection oriented.

I believe that these concerns outweigh the benefits of connectionless transport, and

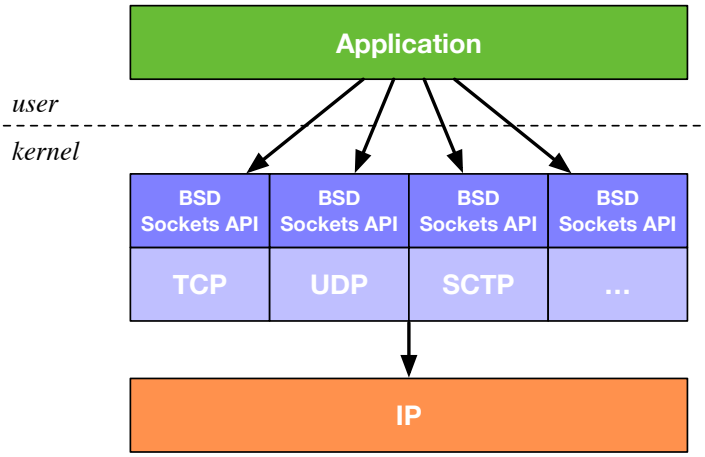


Figure 7.1: Architecture of the existing BSD Sockets API

so add a requirement for a connection-oriented service. Similarly, while not strictly needed by low-latency multimedia applications, it is beneficial if the transport provides a keep-alive service, to refresh NAT and firewall bindings if the application goes silent.

With the set of desirable transport services for low-latency applications defined, it remains to describe how these might be implemented. Before defining a novel abstract API, it is worthwhile to consider why the BSD Sockets API (whose current architecture is illustrated in Figure 7.1) is largely unsuitable.

The BSD Sockets API is not sufficiently expressive to allow the required metadata to be passed between the application and transport layers in a sensible, structured way. For example, support for the timing and deadline service requires that the application expresses timing information about each message that it sends. With the BSD Sockets API, the application is restricted to passing this metadata using `setsockopt()`. However, the nature of socket options [45] means that there is no guarantee that a particular platform will support a given option, or that the same option will be exposed in the same way between different platforms. As a result, providing the required timing metadata to the transport layer using `setsockopt()`, as is necessary with the current API, will inevitably result in additional application code to determine the availability of the socket option, and its form on a particular platform. This is clearly undesirable: in the next section, I will describe an abstract API that can be used to provide the services required by low-latency applications.

7.2 Abstract API

Given the set of transport services summarised in Table 7.1, I sketch an abstract API in Table 7.2. The primitives are divided into four 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 [48]). Care must be taken to ensure that the set of socket options provided does not explicitly bind the application to a particular transport protocol.
- The connection primitives are the same as those of standard TCP sockets. Servers `bind()` to a particular address and port, then `listen()` for an `accept()` incoming connections. Clients `connect()` to a server.
- 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 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 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.

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)
Message oriented	send_message	sd – Socket descriptor buf – Message data len – Length of buf seq_num – Sequence number	Number of bytes sent
Deadline		deadline – Relative deadline	
Dependencies		depends_on – seq_num of dependency	
Sub-streams		substream – Sub-stream identifier	
	recv_message	sd – Socket descriptor buf – Message data len – Size of buf	

Table 7.2: Outline transport API for low-latency applications. Return values shown are for successful calls; in all cases, -1 is returned in the event of an error

This abstract API is broadly the same as that exposed by the intermediary layer of the TCP Hollywood implementation described in Chapter 4 and used throughout Chapters 5 and 6. However, the intermediary layer communicates with the kernel using the standard Berkeley Sockets API, and so while it masks many of the issues described from the application, those issues remain, and introduce complexity within the intermediary layer.

While the API discussed here supports the transport services described in the previous section, expanding the expressivity of the Berkeley Sockets API does not represent the architectural shift that is required to broaden support for new transport protocols. The Berkeley Sockets API couples transport protocols with the services that they provide. This means that applications must select a particular transport protocol. However, the transport services described in the previous section are *not* tied to a particular protocol: they can be provided by a TCP variant (as demonstrated by the design of TCP Hollywood) or over UDP. There are functional (e.g., reachability on a particular path) and non-functional (e.g., efficiency) requirements that may determine which of a set of protocols should be used by the application over a particular path. However, without support at the transport layer, those choices need to be made at the application layer, requiring significant effort on the part of application developers. To properly decouple transport services from the protocols that provide them, it is necessary to provide an API that is protocol-agnostic.

There have been several efforts to formalise a new architecture for the transport layer API, that realises the transport services paradigm. To go beyond the abstract API discussed above, it is worthwhile to consider how TCP Hollywood and the particular set of services needed by low-latency applications might be integrated with these alternative approaches. To do so, I consider three alternative architectures: NEAT (a New, Evolutive API and Transport-Layer architecture), Post Sockets, and the Transport Services architecture.

The **NEAT** [27] architecture (as illustrated in Figure 7.2) provides an API based around transport services. The NEAT User Module is comprised of five broad components: Framework, Selection, Policy, Transport, and Signalling. The Framework components define the broad structure of the NEAT system, defining the API, and implementing the core system structures. Applications use the API to indicate their desired transport services. The Selection components take the information provided by applications, and, in combination with path and system-level metadata held by the Policy components, determine which set of transport layer protocols is appropriate. The Policy components store information about the available interfaces, supported protocols, current connections, and path properties, alongside rulesets for how transport services should be matched to transport protocols. The Selection and Policy components are those which encapsulate the transport services paradigm: applications themselves do not need to make these choices, and instead defer them to a subsystem that can decide at runtime. The Transport components are responsible for configuring and managing a particular transport protocol. Finally, the Signalling components allow for signalling

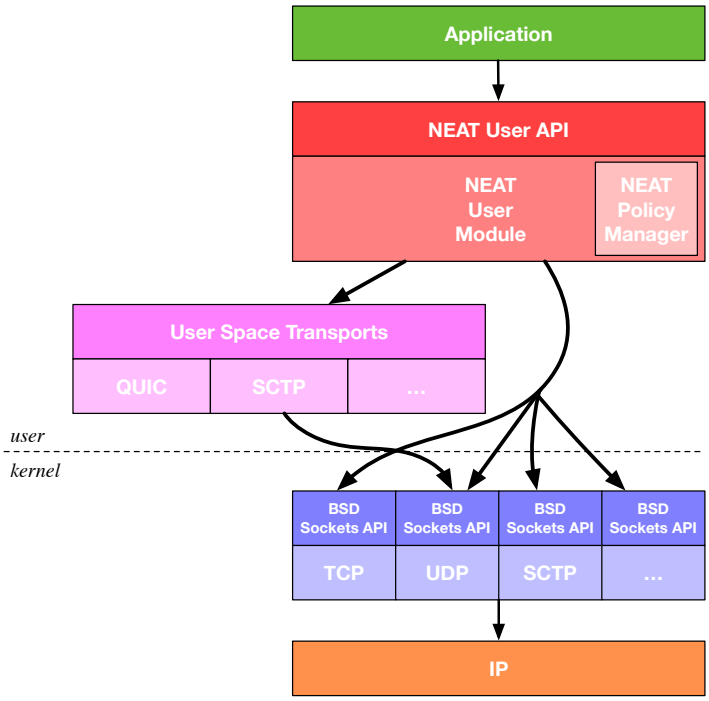


Figure 7.2: The NEAT architecture (from [27])

between two NEAT endpoints, and for between a NEAT endpoint and middleboxes along the path. This allows endpoints to share metadata about path characteristics.

The **Post Sockets** [80] architecture (as illustrated in Figure 7.3) shares NEAT's motivation, in that it aims to define a transport independent API, with a particular protocol being selected dynamically at runtime. Post Sockets replaces the BSD Socket `SOCK_STREAM` abstraction with an API that is closer to that of SCTP's `SOCK_SEQPAKCET` abstraction. Post Sockets provides a message-oriented delivery service, where messages are transmitted via Message Carriers. Messages are, by default, sent fully reliably. However, each message can have a lifetime associated with it, after which it is no longer transmitted. Associations encapsulate connection-oriented behaviour, but are transport-independent, with associations potentially extending beyond any particular transport connection. The Post Sockets architecture includes a Configuration component that incorporates the application's desired transport services; this is combined with path-level metadata (held by the Association) to determine which particular transport protocol should be used to carry messages. An instantiation of a transport protocol is managed by a Transient object, which temporarily binds a Message Carrier to a particular protocol.

The **Transport Services** [63] architecture (as illustrated in Figure 7.4) broadly combines the NEAT and Post Sockets architectures. The Transport Services architecture retains the core design principles of the NEAT and Post Sockets architectures: the relationship between the application and transport protocols should be decoupled, allowing for innovation at the transport layer, in terms of new protocols, and runtime protocol selection, given path limitations.

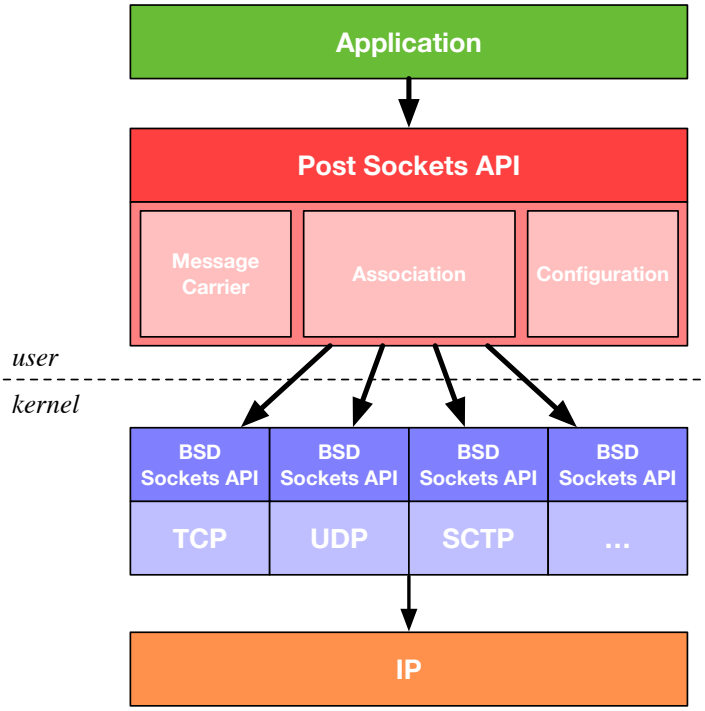


Figure 7.3: The Post Sockets architecture (from [80])

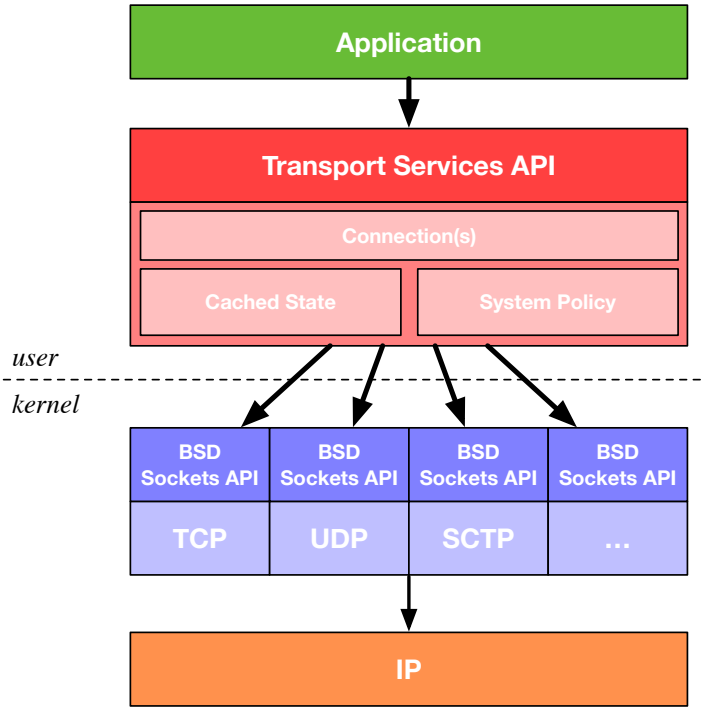


Figure 7.4: The Transport Services architecture (from [63])

In the Transport Services architecture, this design principle is embodied by the Transport System Implementation component, which is responsible for determining which candidate transport protocols are available, and which candidate is most appropriate, given metadata provided by the application.

In summary, it is clear that the services described in the previous section, and the abstract API described here, are likely to be able to be supported by future transport services oriented architectures. This is demonstrated by the widespread adoption of messages as the basic delivery unit, and the abstraction that delivery of these messages may be partially reliable based on timing or dependency metadata provided by the application.

In the next section, I will summarise how transport services might be realised at the protocol level.

7.3 Realising Transport Services

In the descriptions and diagrams of all three alternative architectures discussed in the previous section – NEAT, Post Sockets, and the Transport Services architecture – I have assumed an eventual mapping onto the BSD Sockets API. It should be noted that such a mapping is not necessary: each architecture could remove this API altogether, and subsume transport protocol implementations within them. For example, Apple's Network framework [1], a widely deployed Transport Services API, makes use of user-space protocol implementations, avoiding a mapping through the Berkeley Sockets API.

While further measurement studies are required to confirm the extent to which wire-visible changes to TCP (such as inconsistent retransmissions, required to support partial reliability) are deployable on the Internet, it is clear that all of the transport services required by low-latency applications can be provided using either TCP or UDP.

Evidence that these services can be deployed above UDP exists in the form of the WebRTC data channel [43] and QUIC protocol [41]. 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 [30] offer/answer exchange [69] as part of a WebRTC session [42] (WebRTC media uses RTP over UDP also, further showing the utility of UDP-based transports). This has been deployed in popular web browsers, with global deployment, and has been shown to be effective. The latter is implemented by Google in their Chrome browser, and used as an alternative to TCP, gaining a significant fraction of overall web traffic.

Deployments using UDP are popular and work well. However, as described in Chapter 3, there are also reasons for providing these services over TCP, since there is a significant fraction of networks that block UDP traffic. It is clearly possible to run real-time traffic over

TCP, as demonstrated by the deployment of applications such as Netflix or the BBC iPlayer, which comprise the majority of Internet traffic. However, TCP has an inconvenient API that imposes work on application developers, and introduces higher than desired latency. I have shown how to address these issues, and provide the full set of transport services proposed in Section 7.1, in the design of TCP Hollywood, as described in Chapter 4. Further, in Section 5.3, I demonstrated that TCP Hollywood is widely deployable on the Internet.

There are two important considerations for implementations of the Transport Services architecture. First, given that TCP Hollywood and QUIC demonstrate that both TCP and UDP are capable of providing the services described in Section 7.1, it may be the case that the appropriate protocol for a transient connection may be determined by non-functional, rather than functional, properties. Such properties include the efficiency of the protocol – that is, how many non-payload bits are introduced – or its availability on a given path. These properties must be taken into consideration when selecting a suitable transport protocol. Secondly, the decoupling between applications and transport protocols must be maintained. This is necessary to foster innovation at the transport layer: while middleboxes pose the greatest barrier to protocol deployment, it is clear that requiring change at the application layer is also restrictive.

7.4 Summary

In this chapter, I described the transport services that are required by low-latency multimedia applications, and the architectural changes that are needed to realise these services, both at the API and network levels. The transport services architecture described in this chapter, and TCP Hollywood (whose design is described in Chapter 4) have a mutually beneficial relationship: if the transport services architecture sees widespread deployment, then a protocol like TCP Hollywood is needed to ensure that transport services have the widest reachability; similarly, if TCP Hollywood is to see widespread deployment, then it is likely to require an architectural shift like that encapsulated by the transport services approach.

Chapter 8

Conclusions and Future Work

Low-latency multimedia applications are characterised by tight constraints on end-to-end latency. Operating within these bounds is difficult, given that most network environments provide a best-effort service, introducing unreliable delivery and unpredictable latency. How loss and delay are handled across the entire protocol stack has a significant impact on quality-of-experience.

In this thesis, I have focused on the transport layer. The delivery model provided by a particular transport protocol can significantly increase the amount of latency that is introduced. Reliability and order introduce delay while loss is detected and recovered from. This is at odds with the requirements of low-latency applications, which prefer predictable and bounded latency over strict reliability and order. However, neither UDP or TCP – the two widely deployed transport protocols – provide a delivery model that *does* meet these requirements.

While, in principle, it should be possible to implement a new transport layer protocol with the required delivery model, ossification (Chapter 3) makes this impossible. New protocols must use either UDP or TCP as a substrate, appearing as one of those protocols on the wire, while introducing new end-to-end semantics.

In this thesis, I described the design of TCP Hollywood, which maintains wire compatibility with standard TCP, while supporting the requirements of low-latency multimedia applications. TCP Hollywood provides an unordered, partially reliable message-oriented delivery model, allowing applications to determine their desired trade-off between reliability and latency, and ultimately improving their performance. Finally, I distilled the features of TCP Hollywood into the set of transport services that are required by low-latency multimedia applications.

This chapter is structured as follows:

Section 8.1 revisits the thesis statement from Section 1.1, and details how the thesis has addressed the claims it makes;

Section 8.2 summarises the contributions made by this thesis;

Section 8.3 identifies potential directions for future work that arise from this thesis; and

Section 8.4 summarises this chapter and concludes the thesis.

8.1 Thesis Statement

For reference, the thesis statement, as given in Section 1.1, was:

> I assert that low-latency multimedia applications are poorly served by the existing, widely deployed transport protocols, and that the performance of these applications can be improved with appropriate transport layer support. To test this hypothesis, I will design, implement, and empirically evaluate a protocol that allows applications to define an appropriate trade-off between reliability and delay. Exposing this trade-off leads to improvements in performance in low-latency multimedia applications, where sufficient delay has the same impact on quality-of-experience as loss.

To validate my assertion that low-latency multimedia applications are not adequately served by the existing, widely deployed transport protocols, I began in Chapter 2 by discussing the current state of multimedia delivery on the Internet. In that chapter, I identified several sources of delay and loss, as the application captures and encodes the media into packets, as those packets cross the Internet, and as they are buffered for decoding and eventually playback at the receiver. These delays may be significant, and when combined with the relatively tight bounds imposed by low-latency applications, may negatively impact quality-of-service if care is not taken to minimise the additional latency introduced by the transport and application layers. Fundamentally, it is important to understand the trade-off between reliability and delay that is inherent to packet delivery over best-effort networks.

While for many applications – such as web browsing or large file downloads – it is desirable for loss to be concealed, even at the expense of delay, this is not the case for low-latency multimedia applications. After a certain time, delay becomes equivalent to loss: the impact on quality-of-experience is the same. In a live video stream, for example, it is more important that frames are played out on time, than that *all* frames are played. If a frame does not arrive before its play-out time, it is effectively lost.

With this trade-off in mind, Chapter 2 considers how both RTP and MPEG-DASH handle loss and delay. In particular, it is instructive consider how UDP and TCP – the two widely deployed transport protocols – expose the trade-off between reliability and delay. UDP provides a best-effort packet delivery service, providing no guarantees about datagram delivery or ordering. This provides applications with the flexibility to determine how much loss

and delay they can tolerate. However, it introduces significant complexity at the application layer: UDP does not provide any reliability, flow, or congestion control mechanisms. By contrast, TCP provides flow and congestion control, alongside a fully reliable, ordered bytestream abstraction. This is at the cost of undesirable latency: detecting and repairing loss introduces delay that the application cannot control. In summary, neither UDP or TCP provides all of the services that low-latency multimedia applications require: they need *partial* reliability, within their latency bounds, alongside the network stability that comes from flow and congestion control.

Given that neither UDP or TCP is entirely suitable for low-latency applications, it remains to develop a protocol that does provide all of the desired services. In Chapter 3, I explored how innovation can continue within a transport layer that has been ossified by middleboxes. In that chapter, I positioned UDP and TCP as substrates for the development of future protocols. In essence, new protocols can only provide novel end-to-end delivery abstractions *and* see wide deployment if they look like UDP or TCP on-the-wire. As the chapter highlights, it is clear that UDP is the obvious choice as the substrate for new protocols: there are few behaviours associated with UDP that can be enforced by middleboxes. However, it is likely that middleboxes ossify around protocols carried *within* UDP: RTP and DNS packets may pass through middleboxes while those of new, unknown protocols do not. This is exemplified by the relatively recent deployment of Google's QUIC, where on 5-10% of paths QUIC falls back to TCP, either as the result of the protocol being blocked or severely rate limited. Where applications fall back to TCP, they are likely to see significantly more latency being introduced by its in-order, reliable delivery abstraction. In addition to those applications that are forced to use TCP as a result of UDP reachability issues, as shown by the design of MPEG-DASH (Section 2.3), it is often desirable to use TCP to make use of the existing TCP and HTTP infrastructure (e.g., content delivery networks). As a result, it is instructive to consider how TCP may be used as a substrate in the development of a protocol for low-latency multimedia applications.

In Chapter 4, I described the design and architecture of TCP Hollywood, a protocol that maintains wire-compatibility with standard TCP, but relaxes its reliability guarantee, and removes in-order delivery. TCP Hollywood provides a message-oriented abstraction, with applications providing timing and dependency metadata for each message. When a message is unlikely to be useful to the receiving application, TCP Hollywood stops retransmitting it, using *inconsistent retransmissions* to send an alternative, useful, message in its place. Further, by removing in-order delivery, TCP Hollywood eliminates *head-of-line blocking*, preventing messages from being delayed so long as to be rendered useless.

To validate the claim that TCP Hollywood is deployable, I describe an implementation of the protocol in Chapter 5, alongside a set of deployability evaluations. These evaluations indicate that TCP Hollywood, despite its inconsistent retransmission mechanism being visible to

middleboxes, is deployable across all major fixed-line and mobile ISPs in the UK. Further, in Chapter 5, I describe a series of preliminary performance evaluations that demonstrate TCP Hollywood's delivery model, highlighting the latency improvements that can be gained by eliminating head-of-line blocking, and preventing the retransmission of data that is no longer useful.

To go beyond these preliminary evaluations, in Chapter 6 I describe the changes needed to an MPEG-DASH application when adopting TCP Hollywood, and validate these changes using a simulated HTTP/2 MPEG-DASH application. Importantly, in that chapter I demonstrate the limitations of standard TCP when attempting to reduce latency. For the MPEG-DASH application described, standard TCP limits the possible responses to loss. When a standard TCP segment is lost, its ordering guarantee prevents subsequent packets from being delivered until a retransmission of the lost packet has arrived. This means that playback must *stall*, introducing cumulative latency. However, as TCP Hollywood removes standard TCP's ordering guarantee, the application can instead *skip* the missing data, and continue playback with the data that has arrived. It is by giving applications the flexibility to determine their own trade-off between loss, delay, and quality-of-experience that transport-layer support improves performance.

Finally, in Chapter 7 I formalise the set of *transport services* that TCP Hollywood provides, and describe an abstract API that is suitable for providing them. Further, I discuss alternative proposals for new architectures and APIs that are designed to break the "transport logjam" [23] by decoupling applications from the transport protocols that they use. These novel architectures are vital if TCP Hollywood, and similar protocols, are to see widespread deployment. However, it is also the case that for a transport services architecture to have wide reachability, services must be able to be provided over multiple protocols, requiring protocols like TCP Hollywood, that explore the semantic boundaries of TCP (as enforced by middleboxes), rather than relying upon UDP as a substrate.

In summary, the assertions made in the thesis statement from Section 1.1, and repeated above, have been validated. UDP and TCP do not serve low-latency applications well, either providing too much freedom as to be burdensome to developers, or providing an overly restrictive delivery model that prevents applications from making appropriate trade-offs. In designing, implementing, and empirically evaluating TCP Hollywood, I have demonstrated that with better transport layer support, the performance of low-latency multimedia applications can be improved.

8.2 Contributions

The contributions made by this thesis are as follows:

The design and architecture of TCP Hollywood
TCP Hollywood embodies the substrate-based design principle described in Chapter 3: it appears as standard TCP on-the-wire, while providing an entirely different delivery abstraction. Under TCP Hollywood, applications send messages that can be delivered out-of-order, and with reliability guarantees that are bounded by timing and dependency properties. TCP Hollywood's architecture, by being split across a userspace API library, and a set of kernel extensions, enables partial deployments.

An analysis of TCP Hollywood TCP Hollywood provides two main mechanisms that are aimed at reducing latency: inconsistent retransmissions, that prevent data that is no longer useful from being resent, and out-of-order delivery, which removes head-of-line blocking. In Chapter 5, I identify, by way of analysis, the combinations of application buffering and network round-trip times in which TCP Hollywood reduces the amount of latency introduced, versus standard TCP.

A deployable implementation of TCP Hollywood I have implemented TCP Hollywood in the Linux 3.18.34 kernel, with a userspace library in C. Using this implementation, in Chapter 5, I describe a set of deployability measurements. These measurements show that TCP Hollywood is deployable on all major fixed-line and mobile ISPs within the UK.

An MPEG-DASH architecture modified to use TCP Hollywood In Chapter 6, I describe the modifications needed to allow an MPEG-DASH application to adopt TCP Hollywood. This process highlights the differences in TCP Hollywood's delivery abstraction, versus standard TCP: message-oriented, rather than a bytestream abstraction, with *partial* reliability, and no ordering guarantees. Importantly, TCP Hollywood enables the application to determine an appropriate trade-off between loss, delay, and quality-of-experience, where standard TCP masked those choices.

A description of the *transport services* required by low-latency multimedia applications
In order to elevate TCP Hollywood beyond a set of tweaks to standard TCP, in Chapter 7, I describe the *transport services* that are required by low-latency multimedia applications, alongside an abstract API that is capable of providing them. By decoupling applications and transport protocols, novel protocols like TCP Hollywood are likely to see wider deployment than they would otherwise.

8.3 Future Work

While TCP Hollywood embodies substrate-based transport protocol design, validating the approach, there are a number of features that remain to be explored. In addition, by providing

a broader API that allows applications to provide metadata about the messages that they are sending, TCP Hollywood enables future work in a number of areas. In this section, I describe some potential directions for future work.

8.3.1 Message security & integrity

TCP Hollywood supports partial reliability using inconsistent retransmissions. This maintains wire compatibility with standard TCP: when a TCP segment that contains an expired message is scheduled for retransmission, TCP Hollywood swaps the payload for a message that is still likely to be useful to the receiver. This results in segments having been sent with the same TCP sequence number, but with different payloads.

As discussed in Section 5.3, this is likely to have implications for deployability. Middleboxes that perform deep-packet inspection (that is, that look at the payloads of TCP segments) will be able to detect anomalous TCP behaviour. This is unlikely to be prohibitive to deployability, as confirmed by the deployability results in Section 5.3, and by other measurement studies [33].

However, the use of inconsistent retransmissions may interact negatively with middleboxes that cache and re-segment TCP streams. This could result in messages becoming corrupt along the path between sender and receiver. The receiver may produce a message that has been formed from some combination of the original message and an inconsistent retransmission. This can easily be detected and prevented by computing a checksum for each message, and attaching this to the transmission. The role of a checksum may also be fulfilled by using a secure transport, such as DTLS [68].

8.3.2 Improved sub-stream support

As described in Chapter 4, TCP Hollywood supports multiple streams within the same TCP connection. Applications can specify a sub-stream identifier for each message that they send, and these will be conveyed to the receiving application. While this provides for logically separate streams, TCP Hollywood does not provide any ordering guarantees within each stream. That means that order for upper-layer primitives can only be guaranteed if these are combined within a single TCP Hollywood message. This is illustrated in Section 6.3, where adapting an MPEG-DASH application to use TCP Hollywood requires combining multiple, ordered HTTP/2 frame types into a single TCP Hollywood message.

Without ordering within each sub-stream, TCP Hollywood suffers from the "large datagram" problem [22]: large TCP Hollywood messages will be comprised of multiple TCP segments, the loss of which delays the delivery of the entire message. If messages were ordered within

each sub-stream, then upper-layer primitives that require order would not have to be bundled into a single message, creating smaller messages, and reducing the impact of the "large datagram" problem. This would only require changes within TCP Hollywood's user-space library, easing deployment.

8.3.3 Multipath support

Multipath support is desirable for low-latency multimedia applications. Beyond the rationale that is applicable to most applications – that is, improvements to resilience and throughput – multimedia applications are typically comprised of multiple flows, with different properties. For example, a video streaming application will have at least two flows for audio and video transmission; these flows have different properties, such as timing and bandwidth requirements. It would be beneficial to map these flows to multiple paths through the network.

Multipath TCP [21] is rendered unsuitable for low-latency applications for the same reasons that standard TCP is unsuitable: its delivery model enforces order and reliability at the expense of latency, and the BSD Sockets API lacks the expressivity required to enable better decision-making at the transport layer. The scheduling options for a given segment are severely restricted by the reliable, in-order delivery model.

The metadata exposed by the TCP Hollywood API, and the services described in Chapter 7, allow for a better multipath scheduler. Reliability and order are not guaranteed or enforced, and timing and dependency metadata is exposed by the application. As a result, the scheduler can determine which message should be sent on a given path, at a given time, as a result of combining the message metadata provided by the application, along with information about the path's properties.

Frömmgen et al. [24] explore the potential for application-defined scheduling within Multipath TCP, demonstrating the success of such an approach within the context of an HTTP/2 application. It remains to explore what the impact of different, non-TCP delivery models might have on scheduling objectives, and what additional metadata might be needed from the application.

8.3.4 Transport Services architecture integration

In Chapter 7, I formalised TCP Hollywood into the set of transport services that it provides. By adopting the nomenclature of the IETF's TAPS working group, that chapter illustrated how the deployability of TCP Hollywood might be maximised. Instead of applications specifying the transport protocol that they require, they express the set of transport services that they need. Decoupling applications from transport protocols in this way allows for novel

transport protocols to see use without additional effort from application developers, reducing the barrier to deployment.

While it is clear that TCP Hollywood is complementary to the Transport Services architecture, it remains to integrate the protocol with such an approach. Implementation work is likely to be difficult in the short term as the standardisation process for the Transport Services architecture is ongoing. However, it may be instructive to explore whether such an implementation would be possible, and to identify any issues that could contribute to the standardisation of the architecture.

8.3.5 Input into standardisation work

TCP Hollywood is an instantiation of two general architectural concepts that are seeing increasing standardisation activity. First, IETF's TAPS working group is exploring the transport services architecture described in Chapter 7. The work done in separating TCP Hollywood as a protocol from the services it describes may be valuable to the standardisation of this architecture.

Second, the IETF QUIC working group is standardising the QUIC protocol. QUIC is designed as a modern TCP, limiting the scope of ossification, and opening up the transport layer to innovation. There are features of TCP Hollywood – such as partial reliability – that are not yet incorporated into the QUIC protocol. The development of these features in TCP Hollywood, and there impact that they have on applications, may be a useful contribution to the standardisation process of the QUIC protocol.

8.4 Conclusion

This dissertation has explored the design space of deployable transport services for low-latency multimedia applications. By positing that transport layer innovation is only possible if UDP and TCP are used as substrates, I have designed, implemented, and empirically evaluated TCP Hollywood, a protocol that better serves low-latency multimedia applications. Importantly, TCP Hollywood exposes the trade-off between loss, delay, and quality-of-experience that is inherent to low-latency applications, allowing those applications to determine the most appropriate trade-off. Finally, by elevating TCP Hollywood from an experimental protocol into the space of deployable transport services, I have shown how its deployability might be maximised.

If the Transport Services architecture is to successfully decouple applications from transport protocols, then it is desirable for transport services to be provided by more than one protocol. Services provided by a single protocol limit reachability and ultimately bind applications to

protocols. The success of TCP Hollywood, in both maintaining wire compatibility with standard TCP, and in providing a fundamentally different delivery model, demonstrates the viability of TCP as a substrate.

Appendix A

Reproducibility

To aid with reproducibility, I provide all of the source code used in generating the results described in this thesis, alongside a Makefile that describes and performs the process of performing the experiments, processing and graphing the results, and producing the thesis. The source code for the thesis is available at `http://dx.doi.org/10.5525/gla.researchdata.856`. The evaluations require a modified Linux kernel, with simulated network environments, and different applications. To manage this complexity, I have split the build process into a series of stages. In this appendix, I describe the inputs and outputs of each stage. I begin by describing the required dependencies.

A.1 Dependencies

The build process for the thesis is largely carried out within a virtual machine. This greatly reduces the number of dependencies required, and allows for a modified Linux version – with the TCP Hollywood kernel extensions and API installed – to be used. VirtualBox and Vagrant are used to manage this virtual machine programmatically. A Makefile orchestrates the build process, both instantiating the virtual machine as required, and continuing the build process within the virtual machine. The versions of these tools that were used in my testing are:

- GNU Make (4.2.1)
- Vagrant (2.2.2)
- VirtualBox (5.2.22)

The experiments use the TCP Hollywood kernel and API. These are located in the following GitHub repositories, and the versions used by the experiments are specified in the Makefile:

- Kernel:
  `https://github.com/lumisota/tcp-hollywood-linux-thesis`
- API:
  `https://github.com/lumisota/hollywood-api-video`

These versions of the TCP Hollywood kernel and API are included with the source code for the thesis, in the data repository version indicated above.

A.2 Stage 0: Big Buck Bunny Download

The evaluations (performed in Stage 3) make use of Big Buck Bunny [1]. While a copy of the required file is provided as part of the source code deposited in the data repository described above, the Makefile will download the movie if required. The Standard 2D version is required, encoded at 60fps in 4K Quad HD. Further processing of the video file is performed in Stage 3.

A.3 Stage 1: TCP Hollywood Vagrant box creation

TCP Hollywood is comprised of a set of modifications to the Linux kernel, and a userspace API layer. Stage 1 involves building a virtual machine image that has the TCP Hollywood kernel. Vagrant is used for this process: a clean Ubuntu 14.04 box is downloaded, upon which the specified version of the TCP Hollywood kernel is installed. While the data repository version of the source code includes the revision of the TCP Hollywood kernel required, it will be retrieved from GitHub if it is not included.

Inputs

- TCP Hollywood kernel
- Ubuntu 14.04 (trusty) Vagrant base box

Outputs

- TCP Hollywood kernel Vagrant box

[1] `https://peach.blender.org`.
Big Buck Bunny is ©2008, Blender Foundation / `http://www.bigbuckbunny.org`

A.4 Stage 2: TCP Hollywood API installation

The next step is to install the TCP Hollywood API within the Vagrant box produced in the previous stage. The TCP Hollywood API is provided in the data repository version of the source code; if it is not provided, the required version will be fetched from GitHub. This stage also involves installing the tooling required for the later stages, including ffmpeg, Mininet, and nghttp2 – required for performing the evaluations – and LaTeX – required for producing the thesis PDF.

Inputs

- TCP Hollywood kernel Vagrant box
- TCP Hollywood API

Outputs

- TCP Hollywood Vagrant box

A.5 Stage 3: Evaluation runs

The third stage begins by segmenting the Big Buck Bunny reference video (described as part of Stage 0 above) into the required bitrates and chunk durations, as described in Chapter 6. This occurs within an instantiation of the TCP Hollywood Vagrant box produced by the previous stage. Conducting the video segmentation within a virtual machine reduces the dependencies required to build the thesis.

The experiments themselves, described in Chapter 6, are also conducted within the same virtual machine. Each run produces a set of files: logs from both the client and server, describing the application-layer activity (e.g., what is sent and received).

Inputs

- TCP Hollywood Vagrant box
- Big Buck Bunny (as described in Stage 0 above)

Outputs

Each experiment run produces:

- Client logfile
- Server logfile

A.6 Stage 4: Results post-processing

The previous stage produces output files for each experiment, giving application-layer activity. In this stage, these output files are aggregated, allowing for analysis and graphing in the next stage. The purpose of splitting these tasks into two separate stages is to decouple the output of each experiment run from the graphs or plots that are produced from them. As a result, changes in how the results are presented do not require the experiments themselves to be performed again.

Inputs

- Stage 3 output files
- Stage 4 Python scripts

Outputs

Each group of evaluations (as described in Chapter 6) produces:

- Aggregated playback data
- Aggregated stall rate data
- Aggregated skip rate data

A.7 Stage 5: Plot generation

In this stage, the processed results files generated by the previous stage are graphed.

Inputs

- Stage 4 output files
- Stage 5 matplotlib and gnuplot scripts

Outputs

- Result plots for Chapter 6

A.8 Stage 6: PDF generation

With the results of the experiments graphed, the final stage is to build the thesis PDF.

Inputs

- Thesis TeX and BibTeX files
- Figures (static and those generated in Stage 5)

Outputs

- Thesis PDF

Bibliography

[1] Network Framework. Technical report, Apple Developer Documentation. URL `https://developer.apple.com/documentation/network`.

[2] A. Aggarwal, S. Savage, and T. Anderson. Understanding the performance of TCP pacing. In *Proceedings of the 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 3, pages 1157–1165. IEEE, 2000.

[3] A. Bentaleb, C. Timmerer, A. C. Begen, and R. Zimmermann. Bandwidth prediction in low-latency chunked streaming. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 7–13. ACM, 2019.

[4] D. Bhat, A. Rizk, and M. Zink. Not so QUIC: A performance study of DASH over QUIC. In *Proceedings of the 27th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 13–18. ACM, 2017.

[5] N. Bouzakaria, C. Concolato, and J. Le Feuvre. Overhead and performance of low latency live streaming using MPEG-DASH. In *Proceedings of the 5th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pages 92–97. IEEE, 2014.

[6] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of the 1994 ACM Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, pages 24–35. ACM, 1994.

[7] E. Brosh, S. A. Baset, V. Misra, D. Rubenstein, and H. Schulzrinne. The delay-friendliness of TCP for real-time traffic. *IEEE/ACM Transactions on Networking (TON)*, 18(5):1478–1491, 2010.

[8] C. Byrne and J. Kleberg. Advisory Guidelines for UDP Deployment. Internet-Draft (Work in progress) draft-byrne-opsec-udp-advisory-00, IETF, July 2015.

[9] M. Cha, P. Rodriguez, J. Crowcroft, S. Moon, and X. Amatriain. Watching television over an IP network. In *Proceedings of the 8th ACM Internet Measurement Conference (IMC)*, pages 71–84. ACM, 2008.

[10] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. TCP Fast Open. RFC 7413, IETF, Dec. 2014.

[11] S. Cheshire and M. Baker. Consistent overhead byte stuffing. *IEEE/ACM Transactions on Networking (TON)*, 7(2):159–172, 1999.

[12] M. Claeys, N. Bouten, D. De Vleeschauwer, K. De Schepper, W. Van Leekwijck, S. Latré, and F. De Turck. Deadline-aware TCP congestion control for video streaming services. In *Proceedings of the 12th International Conference on Network and Service Management (CNSM)*, pages 100–108. IEEE, 2016.

[13] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *ACM SIGCOMM Computer Communication Review*, 20(4):200–208, 1990.

[14] L. De Vito, S. Rapuano, and L. Tomaciello. One-way delay measurement: State of the art. *IEEE Transactions on Instrumentation and Measurement*, 57(12):2742–2750, 2008.

[15] J. Dickinson, S. Dickinson, R. Bellis, A. Mankin, and D. Wessels. DNS Transport over TCP - Implementation Requirements. RFC 7766, IETF, Mar. 2016.

[16] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing residential broadband networks. In *Proceedings of the 7th ACM Internet Measurement Conference*, pages 43–56, 2007.

[17] S. Duraimurugan and P. J. Jayarin. Analysis and Study of Multimedia Streaming and Congestion Evading Algorithms in Heterogeneous Network Environment. In *Proceedings of the 2nd International Conference on Intelligent Computing and Control Systems (ICICCS)*, pages 1248–1252. IEEE, 2018.

[18] K. Edeline, M. Kühlewind, B. Trammell, E. Aben, and B. Donnet. Using UDP for Internet transport evolution. arXiv preprint arXiv:1612.07816, arXiv, Dec. 2016.

[19] A. El Essaili, T. Lohmar, and M. Ibrahim. Realization and Evaluation of an End-to-End Low Latency Live DASH System. In *Proceedings of the 2018 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, pages 1–5. IEEE, 2018.

[20] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing. RFC 2827, IETF, May 2000.

[21] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824, IETF, Jan. 2013.

[22] B. Ford. Structured streams: a new transport abstraction. *ACM SIGCOMM Computer Communication Review*, 37(4):361–372, 2007.

[23] B. Ford and J. R. Iyengar. Breaking Up the Transport Logjam. In *Proceedings of the 7th ACM Workshop on Hot Topics in Networks (HotNets)*, pages 85–90, 2008.

[24] A. Frömmgen, A. Rizk, T. Erbshäußer, M. Weller, B. Koldehofe, A. Buchmann, and R. Steinmetz. A programming model for application-defined multipath TCP scheduling. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, pages 134–146. ACM, 2017.

[25] J. Gettys and K. Nichols. Bufferbloat: Dark Buffers in the Internet. *ACM Queue*, 9(11):40:40–40:54, Nov. 2011.

[26] K.-J. Grinnemo and A. Brunstrom. Evaluation of the QoS offered by PRTP-ECN: a TCP-compliant partially reliable transport protocol. In *Proceedings of the International Workshop on Quality of Service*, pages 217–230. Springer, 2001.

[27] K.-J. Grinnemo, T. Jones, G. Fairhurst, D. Ros, A. Brunstrom, and P. Hurtig. Towards a flexible Internet transport layer architecture. In *Proceedings of the 2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–7. IEEE, 2016.

[28] S. Ha, I. Rhee, and L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

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

[30] M. Handley, V. Jacobson, and C. Perkins. SDP: Session Description Protocol. RFC 4566, IETF, July 2006.

[31] M. Handley and C. Perkins. Guidelines for Writers of RTP Payload Format Specifications. RFC 2736, IETF, Dec. 1999.

[32] 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 Internet Measurement Conference (IMC)*, pages 260–266. ACM, 2010.

[33] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? In *Proceedings of the 11th ACM Internet Measurement Conference (IMC)*, pages 181–194. ACM, 2011.

[34] K. Hughes and D. Singer. Information technology–Multimedia application format (MPEG-A)–Part 19: Common media application format (CMAF) for segmented media. *ISO/IEC*, pages 23000–19, 2017.

[35] Y. Humeida, M. Nilsson, and S. Appleby. TCP Congestion Response for Low Latency HTTP Live Streaming. In *2018 IEEE 19th International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*, pages 14–21. IEEE, 2018.

[36] ISO/IEC. Information Technology: Generic coding of moving pictures and associated audio information. International Standard 13818-2.

[37] ITU-T. G.726: 40, 32, 24, 16 kbit/s Adaptive Differential Pulse Code Modulation (ADPCM). Technical report, ITU, December 1990.

[38] ITU-T. G.114: One-way transmission time. Technical report, ITU, May 2003.

[39] ITU-T. G.723.1: Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s. Technical report, ITU, May 2006.

[40] ITU-T. Contribution 545: Consideration on Channel Zapping Time in IPTV Performance Monitoring. Technical report, ITU, April 2007.

[41] J. Iyengar and M. Thomson. QUIC: A UDP-based multiplexed and secure transport. Internet-Draft (Work in progress) draft-ietf-quic-transport-22, IETF, Aug. 2019.

[42] C. Jennings, T. Hardie, and M. Westerlund. Real-time communications for the web. *IEEE Communications Magazine*, 51(4):20–26, 2013.

[43] R. Jesup, S. Loreto, and M. Tuexen. WebRTC data channels. Internet-Draft (Work in progress) draft-ietf-rtcweb-data-channel-13, IETF, Jan. 2015.

[44] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance. In *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 4, pages 2490–2501. IEEE, 2004.

[45] T. Jones, G. Fairhurst, and C. Perkins. Raising the datagram API to support transport protocol evolution. In *Proceedings of the 2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–6. IEEE, 2017.

[46] M. J. Karam and F. A. Tobagi. Analysis of delay and delay jitter of voice traffic in the Internet. *Computer Networks*, 40(6):711–726, 2002.

[47] N. Khademi, D. Ros, and M. Welzl. The new AQM kids on the block: An experimental evaluation of CoDel and PIE. In *Proceedings of the 17th IEEE Global Internet Symposium*, pages 85–90. IEEE, 2014.

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

[49] J. Kua, G. Armitage, and P. Branch. A survey of rate adaptation techniques for dynamic adaptive streaming over HTTP. *IEEE Communications Surveys & Tutorials*, 19(3):1842–1866, 2017.

[50] D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–59, 1991.

[51] S. Liang and D. Cheriton. TCP-RTM: Using TCP for real time multimedia applications. In *Proceedings of the International Conference on Network Protocols (ICNP)*, 2002.

[52] S. McQuistin and C. Perkins. Reinterpreting the transport protocol stack to embrace ossification. In *Proceedings of the IAB Workshop on Stack Evolution in a Middlebox Internet (SEMI)*. IAB, January 2015.

[53] S. McQuistin, C. Perkins, and M. Fayed. Implementing real-time transport services over an ossified network. In *Proceedings of the 2016 Applied Networking Research Workshop*, pages 81–87. ACM, 2016.

[54] S. McQuistin, C. Perkins, and M. Fayed. TCP goes to Hollywood. In *Proceedings of the 26th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, page 5. ACM, 2016.

[55] S. McQuistin, C. Perkins, and M. Fayed. TCP Hollywood: An unordered, time-lined, TCP for networked multimedia applications. In *Proceedings of the 2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 422–430. IEEE, 2016.

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

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

[58] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[59] M. F. Nowlan, N. Tiwari, J. Iyengar, S. O. Aminy, and B. Ford. Fitting square pegs through round pipes: Unordered delivery wire-compatible with TCP and TLS. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 28–28. USENIX Association, 2012.

[60] Ofcom. UK Home Broadband Performance: The performance of fixed-line broadband delivered to UK residential consumers. Research report, Ofcom, May 2018.

[61] J.-R. Ohm. Advances in scalable video coding. *Proceedings of the IEEE*, 93(1):42–56, 2005.

[62] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, et al. De-ossifying the Internet transport layer: A survey and future perspectives. *IEEE Communications Surveys & Tutorials*, 19(1):619–639, 2017.

[63] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, C. Perkins, P. Tiesel, and C. Wood. QUIC: A UDP-based multiplexed and secure transport. Internet-Draft (Work in progress) draft-ietf-taps-arch-03, IETF, 2019.

[64] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP's Retransmission Timer. RFC 6298, IETF, June 2011.

[65] C. Perkins, M. Westerlund, and J. Ott. Web Real-Time Communication (WebRTC): Media Transport and Use of RTP. Internet-Draft (Work in progress) draft-ietf-rtcweb-rtp-usage-26, IETF, Mar. 2016.

[66] L. Popa, A. Ghodsi, and I. Stoica. HTTP as the narrow waist of the future Internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, page 6. ACM, 2010.

[67] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 399–412, 2012.

[68] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, IETF, Jan. 2012.

[69] J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264, IETF, June 2002.

[70] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, IETF, June 2002.

[71] 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.

[72] H. Schulzrinne and S. Casner. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 3551, IETF, July 2003.

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

[74] R. Stewart. Stream Control Transmission Protocol. RFC 4960, IETF, Sept. 2007.

[75] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960, IETF, Oct. 2000.

[76] T. Stockhammer. Dynamic Adaptive Streaming Over HTTP – Standards and Design Principles. In *Proceedings of the Conference on Multimedia Systems (MMSys)*, San Jose, CA, USA, February 2011. ACM. doi:10.1145/1943552.1943572.

[77] S. D. Strowes. Passively measuring TCP round-trip times. *Communications of the ACM*, 56(10):57–64, 2013.

[78] I. Swett. QUIC deployment experience at Google. `https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf`, 2016.

[79] B. Trammell and M. Kuehlewind. Applicability of the QUIC Transport Protocol. Internet-Draft (Work in progress) draft-ietf-quic-applicability-03, IETF, Oct. 2018.

[80] B. Trammell, C. Perkins, T. Pauly, M. Kuehlewind, and C. Wood. Post Sockets, An Abstract Programming Interface for the Transport Layer. Internet-Draft (Work in progress) draft-trammell-taps-post-sockets-03, IETF, Oct. 2017.

[81] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.

[82] 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.

[83] M. Zanaty, V. Singh, S. Nandakumar, and Z. Sarkar. Congestion Control and Codec interactions in RTP Applications. Internet-Draft (Work in progress) draft-ietf-rmcat-cc-codec-interactions-02, IETF, Mar. 2016.