Towards Faster Web Page Loads Over Multiple Network Paths

Lynne Salameh

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Computer Science
University College London
April 12, 2018

I, Lynne Salameh, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

The rising popularity of mobile devices as the main way people access the web has fuelled a corresponding need for faster web downloads on these devices. Emerging web protocols like HTTP/2 and QUIC employ several features that minimise page load times, but fail to take advantage of the availability of at least two interfaces on today's mobile devices. On the other hand, this spread of devices with access to multiple paths has prompted the design of Multipath TCP, a transport protocol that pools bandwidth across these paths. Although MPTCP was originally evaluated for bandwidth limited bulk transfers, in this work, we determine whether using MPTCP can reduce web page load times, which are often latency bound.

To investigate the behaviour of web browsing over MPTCP, we instrumented the Chrome web browser's retrieval of 300 popular web sites in sufficient detail, and computed their dependency graph structure. Furthermore, we implemented PCP, an emulation framework that uses these dependency graphs to ask "what-if" questions about the interactions between a wide range of web site designs, varied network conditions, and different web and transport protocols.

Using PCP, we first confirm previous results with respect to the improvements HTTP/2 offers over HTTP/1.1. One obstacle, though, is that many web sites have been sharded to improve performance with HTTP/1.1, spreading their content across multiple subdomains. We therefore examine whether the advice to unshard these domains is beneficial. We find that unsharding is generally advantageous, but is not consistently so.

Finally, we examine the behaviour of HTTP/2 over MPTCP. We find that MPTCP can improve web page load times under some regimes; in other cases, using regular TCP on the "best" path is more advantageous. We present enhancements to multipath web browsing that allow it to perform as well as or better than regular TCP on the best path.

Impact Statement

Since the first web site came online in August 1991 [1], the number of live web pages has continued to grow exponentially. As of today, the number of web pages on the Internet is rapidly approaching the 2 billion mark, with approximately 200 million active web pages in existence [2]. As a result of their wide proliferation and of people's reliance on them for various aspects of their lives, it comes as no surprise that improving web page load latencies will produce a far reaching impact on users' quality of experience. On the commercial side, delivering fast search results, shopping transactions or ads is crucial for web content providers. Even small speed ups in page latency can produce a measurable increase in revenue.

In the upcoming chapters, we present an in depth analysis of web page downloads over single and multiple paths. Understanding the subtle interactions between application and transport layer protocols has both academic and commercial importance. Exposing the areas where the benefits of using a particular protocol or strategy outweighs the alternative may inform operational design decisions and the development of new web or transport protocols. Additionally, given the general advice to unshard and coalesce web page content in light of increased HTTP/2 adoption, it is essential for web page designers to understand the full consequences of unsharding on page load times. As for MPTCP, we present the correct mechanisms to address its limitations. With these fixes in place, users and content providers will gain a marked improvement in page load times if web downloads take advantage of multiple paths.

To arrive at our results, we have implemented several tools that we believe may benefit researchers in the networking community. First, we constructed dependency graphs for 300 top Alexa web sites, which represent highly detailed models of real web pages profiled using Chrome. The dependency graphs are available at [3] for use in future research. Second, we designed a robust experimentation framework for emulating web page downloads using their dependency graphs. The framework examines web page downloads under a varied set of web and transport protocols, network parameters, server architectures, and network paths. Finally, we produced a working implementation of MPTCP for the open-source ns-3 network simulator, which can be found at [4].

Acknowledgements

My deepest gratitude extends to my supervisors, Mark Handley and Brad Karp, whose guidance and deep insight have been invaluable to my growth as a researcher. Without their abundant knowledge and their encouragement, I would not have been able to push forward towards completing this endeavour.

Mark Handley has never ceased to astound me with his uncanny ability to take one glance at a problem and immediately discern its underlying causes. His deep intuition, creativity, and intellect have been crucial components in learning how to tackle difficult research problems and viewing them with an entirely new perspective.

Brad Karp's rigour and meticulous attention to detail have taught me essential lessons about producing high quality work. His wisdom, perseverance, and strive towards excellence have helped shape my research and have improved it immensely.

To my parents, Leila and Rafik, I offer my sincerest thanks. From an early age they have instilled in me a deep love for science and discovery that continues to shape my life today. Without their unwavering faith in me and their support, I would never have embarked on the path towards a PhD at all. I owe them immensely for the opportunities they have provided me. My sisters Abeer and Zeina, and my brother Amjad, deserve my heartfelt thanks for their support and their encouragement. Abeer's comments on my thesis chapters have been indispensable.

I am very grateful to Stefano Vissicchio, whose feedback on various submissions and thesis chapters has been vital. Many thanks extend to Nikola Gvozdiev, Astrit Zhushi, Georgios Nikolaidis and Petr Marchenko, my fellow PhD students and brothers-in-arms, not only for their help in editing papers and polishing presentations, but for enriching my experience at UCL through their friendship.

Contents

1 Introduction **17**

1.1 Problem Statement . . . 19

1.2 Thesis Roadmap . . . 21

1.3 Contribution . . . 22

2 Background and Related Work **24**

2.1 Classic Web Protocols . . . 25

2.1.1 HTTP/1.0 . . . 25

2.1.2 HTTP/1.1 . . . 26

2.1.3 Optimisations . . . 27

2.2 Emerging Web and Transport Protocols . . . 30

2.2.1 HTTP/2 . . . 30

2.2.2 QUIC . . . 34

2.2.3 BBR . . . 37

2.3 Multipath Protocols . . . 38

2.3.1 MPTCP . . . 39

2.3.2 Path Schedulers . . . 40

2.4 Dependency Graphs . . . 43

2.5 Proxies . . . 44

2.6 Conclusion . . . 46

3 Page Completion Profiler **47**

3.1 Introduction . . . 47

3.2 Anatomy of a Page Load . . . 48

3.3 Design Overview . . . 52

3.4 Profiler . . . 54

3.4.1 Server Processing Times . . . 55

3.5 Graph Builder . . . 58

3.5.1 Dependency Activities . . . 59

3.5.2 Frame Completion Events . . . 64

3.5.3 Resource Prioritisation . . . 68

3.6 Graph Viewer . . . 71

3.7 Emulator . . . 71

3.7.1 Dimensionality of Problem Space . . . 71
3.7.2 Browser Model . . . 73
3.7.3 Measuring Completion . . . 78
3.7.4 Ensuring Consistency . . . 79
3.7.5 Browser Optimisations . . . 80
3.7.6 Server Model . . . 85
3.7.7 Network Model . . . 88
3.7.8 Conclusion . . . 98

4 Web Baseline Evaluation **100**
4.1 Introduction . . . 100
4.2 Web Page Characterisation . . . 101
4.2.1 Resource Counts and Sizes . . . 102
4.2.2 Resource Distribution Across Domains . . . 104
4.2.3 Latency to Web Servers . . . 106
4.2.4 Resource Dependencies . . . 108
4.3 Methodology . . . 109
4.4 Experiments . . . 112
4.4.1 Connections . . . 112
4.4.2 Bandwidth and RTT . . . 118
4.4.3 Web Protocols . . . 121
4.4.4 Prioritisation Schemes . . . 131
4.4.5 Server Design . . . 137
4.4.6 Conclusions . . . 138

5 Unsharding **141**
5.1 Introduction . . . 141
5.2 Methodology . . . 142
5.3 Results . . . 144
5.3.1 Overview . . . 144
5.3.2 Unsharded *vs.* Sharded HTTP/2 . . . 147
5.3.3 PLT Differences . . . 149
5.3.4 Case Studies . . . 151
5.4 Conclusions and Discussion . . . 159

6 Multipath for Web **161**
6.1 Introduction . . . 161
6.2 Methodology . . . 163
6.2.1 Topology . . . 163
6.2.2 MPTCP in ns-3 . . . 164

6.2.3 MPTCP Congestion Control . . . 167
6.3 Performance of MPTCP for Web . . . 168
6.3.1 Equivalent Paths . . . 168
6.3.2 RTT Mismatch . . . 171
6.3.3 Bandwidth Mismatch . . . 173
6.3.4 “Worst” Path First . . . 179
6.3.5 Combination Paths . . . 183
6.4 Improving MPTCP . . . 185
6.4.1 Stream Scheduler . . . 185
6.4.2 Preemption . . . 193
6.4.3 Out of Order Delivery . . . 193
6.4.4 Disabling Linked Increase . . . 194
6.4.5 Tail Loss Recovery . . . 194
6.4.6 Interface Heuristics . . . 197
6.5 MPTCP Improvements Evaluation . . . 198
6.5.1 RTT mismatch . . . 198
6.5.2 Bandwidth Mismatch . . . 207
6.5.3 Combination Paths . . . 212
6.6 Discussion . . . 212
6.6.1 Completion Estimation Inaccuracies . . . 212
6.6.2 Subflow Setup Delays . . . 214
6.6.3 Content Sharding . . . 215
6.6.4 Stack Changes . . . 215
6.6.5 Energy . . . 215

7 Conclusions **217**
7.1 Future Research . . . 219

Appendices **223**

A DOM Event Dependencies **225**

B Negative Acknowledgements **229**

C Congestion Window Validation **234**

Bibliography **237**

List of Figures

2.1 HTTP request issued from client, with associated HTTP response. 25
2.2 Frame exchanges under the different variants of HTTP. 27
2.3 The format of two types of HTTP/2 frames: `HEADERS` and `DATA`. 31
2.4 HTTP/2 running over the traditional network stack, vs. QUIC's new architecture. 34
2.5 A typical QUIC packet containing `STREAM` frames to transmit data. 34
2.6 Messages exchanged to establish a QUIC connection for the 1-RTT or 0-RTT scenarios. 35
2.7 Relationship between RTT and packet delivery rate *vs.* number of bytes in flight. 37

3.1 An example HTML file. 49
3.2 Measurement of server processing delay from *tcpdump* traces at the client. 55
3.3 Server processing delays for content served via the specified protocol. 57
3.4 JavaScript server processing delays from 2015. 58
3.5 Our example HTML file, reproduced with the relevant annotations. 63
3.6 Dependency graph of the example HTML page. 65
3.7 Page with subframes. 66
3.8 Frame load completion states. 66
3.9 Frame completion events chain. 67
3.10 Zoomed out view of dependency graph of `amazon.com`. 72
3.11 Example web page dependency graph with ambiguity regarding the order of executing Script Eval 1 and Script Eval 2. 75
3.12 Example web dependency graph where the Window `load` event may have different parents depending on the network conditions. 79
3.13 CDF of the resource maximum age. 82
3.14 Fraction of resources which require refresh after elapsed time. 83
3.15 Server models in PCP, demonstrating what will happen on receiving three pipelined requests R1, R2 and R3. The server may apply the processing times corresponding to each request (P1, P2 and P3) serially, in parallel, or out-of-order. 87
3.16 Overview of network topology for downloading a web page. 89
3.17 Topology for the emulated web downloads. 90
3.18 RED queue drop probability P_a as a function of the average queue size. 94

4.1 CDF of resource sizes across 300 top Alexa web sites. 103

4.2 CDFs for resource count, page size and average resource size across web sites. . . 103
4.3 CDFs of the number of domains per web site, the proportion of resources downloaded from the largest domain, and the count and size of domain shards. The CDFs are computed across 300 web sites. 105
4.4 CDFs for the proportion of resources which correspond to ads, and the number of iframes per web page. 106
4.5 CDFs showing the proportion of domains corresponding to CDN edge servers, and the proportion of bytes downloaded from those domains. 106
4.6 CDF of RTTs measured using the Profiler in PCP, when connection to CDN edge servers or origin servers. 107
4.7 Proportion of resource bytes and counts downloaded as a result of completing different dependency graph activities. 108
4.8 CDF of the proportion of time on the critical path spent performing the various network and browser activities. 109
4.9 The best number of connections with respect to median PLT for each web protocol, given the bandwidth and RTT. Each line on the graph corresponds to the labelled RTT. 113
4.10 The median packet drop rate for HTTP/1.1 under various bandwidth/RTT regimes, and an increasing number of connections shown as different lines in each graph. 114
4.11 The percentage increase in PLT if we use the chosen number of connections instead of the best. Each line on the graph corresponds to the labelled RTT. 117
4.12 Surface plot for the median PLT as a function of the bandwidth and RTT for web pages downloaded using HTTP/1.1 with six connections and TCP New Reno. . . 118
4.13 Surface plot for the median PLT as a function of the inverse bandwidth and RTT for web pages downloaded using HTTP/1.1 with six connections and TCP New Reno. 119
4.14 Contour plot for the median PLT for HTTP/1.1 with six connections, with respect to bandwidth and RTT. 120
4.15 Heatmaps comparing the mean PLT of HTTP/2 with one connection with HTTP/1.1 with six connections. Neither protocol applies resource prioritisation. The percentage improvement is computed using $\frac{\sum PLT_{http1.1} - PLT_{http2}}{\sum PLT_{http1.1}}$. 121
4.16 Heatmap indicating the percentage of web sites with equal or better mean PLT using HTTP/2 with one connection *vs.* HTTP/1.1 with six connections. Neither protocol applies resource prioritisation. 122
4.17 Heatmap of mean packet drop rate at different bandwidths and RTTs for HTTP/1.1 and HTTP/2. 123
4.18 Time/sequence plots for downloading bestbuy.com at 2.5Mbps and 200ms. 124

4.19 Time/sequence plot for `buzzhand.com` downloaded using HTTP/2 with 1 connection at a bandwidth of 2.5Mbps and RTT of 200ms, using the labelled TCP congestion control. 125
4.20 Time/sequence plot for `buzzhand.com` downloaded using HTTP/1.1 with six connections at a bandwidth of 2.5Mbps and RTT of 200ms. 126
4.21 Heatmap indicating the percentage of web sites with equal or better mean PLT using QUIC-lite with 1 connection and no prioritisation compared with HTTP/1.1 with six connections. 127
4.22 CDFs of the PLTs for 300 web sites across 50 runs under the different web protocols. The curves for QUIC-lite and HTTP/2 are overlaid. 129
4.23 CDFs of the difference between the mean PLTs of 300 web sites under the different web protocols compared with HTTP/1.1 with six connections. The curves for HTTP/1.1 Pipe, HTTP/2 and QUIC-lite are overlaid. 130
4.24 Percentage improvement in mean PLT for HTTP/2 with the specified prioritisation over HTTP/2 with no prioritisation. 131
4.25 CDFs of the differences of HTTP/2 with 1 connection and different priorities, and HTTP/1.1 six connections. CP is a shorthand for critical path prioritisation. . 133
4.26 Dependency graph for `steamcommunity.com`. 134
4.27 Timelines for downloading `steamcommunity.com` under different priority schemes, downloaded at 2.5Mbps and an RTT of 200ms. 135
4.28 Percentage improvement of QUIC-lite with the specified prioritisation over HTTP/2 with no prioritisation. 137
4.29 Timelines for downloading `steamcommunity.com` under different server architectures, downloaded at a bandwidth of 25Mbps, RTT of 40ms, and 1 connection per domain. 139

5.1 Histogram depicting the number of shards coalesced for the 300 web sites. 143
5.2 Heatmap showing the percentage of web sites with equal or faster mean PLT, as measured by the `load` event metric, using HTTP/2 (with MIME-type prioritisation) *vs.* HTTP/1.1 with 6 connections per domain, for web sites with sharded or unsharded domains. 144
5.3 Percentage reduction in mean PLT as a result of unsharding domains under HTTP/2 with MIME-type prioritisation, when compared with the sharded domain case. 145
5.4 Heatmap showing the percentage of web sites with equal or faster mean PLT, as measured by the `load` or `DOMContentLoaded` events, using HTTP/2 with unsharded domains *vs.* sharded HTTP/2. 148
5.5 Heatmap showing the percentage of web sites where the packet drop rate was higher with unsharded HTTP/2 than the sharded case. 148

5.6 CDF of the difference between the mean unsharded completion times and sharded completion times for some typical bandwidth and RTT values. 149
5.7 CDF across the 283 web sites showing, when we unshard each site, how many out of the 50 *(Bandwidth, RTT)* combinations give better (or equal) performance. . 150
5.8 Dependency graph for expedia.co.uk. 151
5.9 Timeline plot of expedia.co.uk downloaded at 1Mbps and 200ms RTT using HTTP/2 152
5.10 Timeline plot of quora.com downloaded over a 50Mbps link with 40ms RTT using HTTP/2. 154
5.11 Time/sequence plot for unsharded quora.com downloaded on a 50Mbps link with 40ms RTT. 155
5.12 Dependency graph generated for gizmodo.co.uk. 157
5.13 Timeline plots for gizmodo.co.uk downloaded on a 10Mbps link with 40ms RTT . 158

6.1 CDF of the difference between PLTs of 300 web pages relative to single-path TCP, 50Mbps bandwidth, 100ms RTT on 1st subflow, 500ms on 2nd one. 162
6.2 Multipath web topology. 164
6.3 MPTCP protocol stack. 165
6.4 Equivalent paths: heatmap of the mean percentage reduction in PLT for HTTP/2 running over MPTCP compared to HTTP/2 running over single path TCP. 169
6.5 For equivalent path MPTCP, CDFs of the sample standard deviation of the PLTs of each web site over 50 runs for the different bandwidths, and RTTs. 170
6.6 $5\times$ RTT on second path: heatmap indicating the percentage of web sites with equal or better PLT using MPTCP vs single-path TCP. 171
6.7 Timeline plot of pages.github.com downloaded over a 50Mbps link with 100ms RTT using HTTP/2. 172
6.8 $\frac{1}{5}$ bandwidth on second path: heatmap indicating the percentage of web sites with equal or better PLT using MPTCP vs single-path TCP. 173
6.9 Timeline plots for groupon.com downloaded using MPTCP with a bandwidth of 10Mbps and RTT of 40ms, with $\frac{1}{5}$ bandwidth on the second path. 175
6.10 Timeline plots for groupon.com downloaded using MPTCP with a bandwidth of 10Mbps and RTT of 40ms, with $\frac{1}{5}$ the bandwidth on the second path. 176
6.11 Time sequence plot of amazon.co.uk downloaded using MPTCP with a bandwidth of 10Mbps on the first path, 2Mbps on the second, and 8ms RTT on both paths. 177
6.12 Median packet loss rate for single path TCP *vs.* bandwidth for different RTTs. . . 179
6.13 Median tail loss rate *vs.* bandwidth for different RTTs. 180
6.14 Initiating an MPTCP connection and establishing two subflows. 181

6.15 Percentage of web sites with equal or better PLT using MPTCP vs single-path TCP when MPTCP uses the "worst" path first. 182
6.16 Downloading intuit.com using MPTCP which initiates connections on the "worse" path first. 182
6.17 Percentage of web sites with equal or better PLT using MPTCP *vs.* single-path TCP on the "best" path ($p = (B, R)$). 183
6.18 Percentage of web sites with equal or better PLT using MPTCP *vs.* single-path TCP on the "best" path. Above the red line, $(B, 5R)$ is the best path. Below the read line, $(\frac{1}{5}B, R)$ is best. 184
6.19 Choosing viable subflows 187
6.20 Real *vs.* Estimated resource completion time for amazon.co.uk downloaded using single-path TCP on a connection with bandwidth of 10Mbps and RTT of 40ms. . 192
6.21 Division of a resource across two subflows, and the effect of the PSH flag. 197
6.22 $5\times$ RTT on second path: heatmap indicating the percentage of web sites with equal or better PLT using modified MPTCP *vs.* single-path TCP. 199
6.23 Timeline plot for pages.github.com using MP QUIC with Stream Scheduler and preemption on path with 50Mbps bandwidth, 100ms RTT, and $5\times$ RTT on second path. 200
6.24 CDFs of the difference between the mean PLTs of 300 web sites under the depicted MPTCP regime and single-path TCP. The second path has $5\times$ the RTT. . . 201
6.25 The percentage improvement in mean PLT over single path when using modified or baseline MPTCP with the RTT mismatch case. The factor multiplying the RTT on the slower path is shown on the x-axis. Each figure's title shows the RTT on the faster path. 206
6.26 $\frac{1}{5}$ bandwidth on second path: heatmap indicating the percentage of web sites with equal or better PLT with modified MPTCP vs single-path TCP (with TLP). We use MP QUIC with no LIA and TLP. 207
6.27 Time/sequence plot of flows on Path 1 for suning.com downloaded on 50Mbps/100ms with $\frac{1}{5}$ bandwidth on second path. 208
6.28 CDFs of the difference between the mean PLTs of 300 web sites under the depicted MPTCP regime and single-path TCP. The second path has $\frac{1}{5}$ the bandwidth.209
6.29 Percentage of web sites with equal or better PLT using improved MPTCP *vs.* single-path TCP on the "best" path. Above the red line, $(B, 5R)$ is the best path. Below the read line, $(\frac{1}{5}B, R)$ is best. We use *MP QUIC Stream* with LIA and TLP. 212

B.1 Time/sequence plot for a connection to one of the domains of imgur.com, downloaded at 10Mbps and 40ms RTT using TCP New Reno. 230
B.2 Option format for SACK and NACK TCP options. 231

B.3 Time/sequence plot for a connection to one of the domains of imgur.com, downloaded at 10Mbps and 40ms RTT using TCP New Reno with NACKs. 232

List of Tables

3.2 Dimensions for the web download experiments, and our selected parameters for emulation. 73

3.3 TCP New Reno parameters in the ns-3 emulator. 93

3.4 RED queue parameters. 95

4.1 Simulated parameters and their values for the web baseline experiments. 111

5.1 Mean PLT values for sharded and unsharded HTTP/2. $\bar{x}$ is the mean over all values of corresponding $(PLT_{shard} - PLT_{unshard})$ in seconds, σ is the sample standard deviation for $\bar{x}$. The % Imp corresponds to $\frac{\sum(PLT_{shard}-PLT_{unshard})}{\sum PLT_{shard}} \times 100\%$. The p-value represents the probability that the mean PLTs of sharded and unsharded HTTP/2 are not significantly different, and is the result of computing one-way ANOVA across the two groups. 146

6.1 PLT difference of modified and baseline MPTCP with respect to single-path TCP for the delay mismatch scenario. PLT_{TCP} is the mean PLT in seconds for the single-path case downloaded on the best path. $\bar{x}$ is the mean over all values of corresponding $(PLT_{TCP} - PLT_{MPTCP})$ in seconds, σ is the sample standard deviation, and the % Imp is $\frac{\sum(PLT_{TCP}-PLT_{MPTCP})}{\sum PLT_{TCP}} \times 100\%$. The p-value represents the probability that the mean PLTs of modified and baseline MPTCP are not significantly different, and is the result of computing one-way ANOVA across the two groups. 204

6.2 PLT difference of modified and baseline MPTCP with respect to single-path TCP for the bandwidth mismatch scenario. PLT_{TCP} is the mean PLT in seconds for the single-path case downloaded on the best path. $\bar{x}$ is the mean over all values of corresponding to $(PLT_{TCP} - PLT_{MPTCP})$ in seconds, σ is the sample standard deviation, and the % Imp is $\frac{\sum(PLT_{TCP}-PLT_{MPTCP})}{\sum PLT_{TCP}} \times 100\%$. The p-value represents the probability that the mean PLTs of modified and baseline MPTCP are not significantly different, and is the result of computing one-way ANOVA across the two groups. 211

A.1 Events which target a DOM Document, and their dependency information. . . . 225

A.2 Events which target a DOM Element, and their dependency information. 226

A.3 Events with XMLHTTPRequest and MessagePort targets, and their dependency information. 227
A.4 Events that target a DOM Window, and their dependency information. 228

Chapter 1

Introduction

With the advent of the digital age, browsing the web has become an indispensable part of people's lives. Users expect web pages to load quickly, especially on their mobile devices, which have now become the main way most people access the web. Since mobile devices typically support two network interfaces, WiFi and 3G/LTE, they can in principle allow communications over multiple network paths. In this work, our main goal is to investigate whether using two paths for downloading web pages will in fact make them complete faster. Along the way, we will investigate the behaviour of state-of-the-art web protocols for the single-path case, in order to establish a viable baseline for comparison with multiple paths. We will also examine the advice to coalesce content dispersed across several server domains in order to speed up web page downloads under emerging web protocols.

With these goals in mind, let us briefly step back to explain some of our motivations. Web downloads currently consume a significant proportion of Internet traffic, second only to video. Studies performed by Cisco in 2016 estimate that around 15-17% of global IP consumer traffic corresponds to the web [5, 6].[1] Web traffic is expected to grow in the next five years, as indicated in the same studies, due to the popularity of social networking, file hosting and video streaming sites which continue to drive the upwards trend [7]. Additionally, the number of web pages has surpassed the one billion mark as of 2014 [2]. But, perhaps a metric of greater concern isn't how much traffic the web generates, but rather how much time people spend browsing online. According to a survey conducted by Ofcom in 2016, people in the UK spend on average 81.8 hours a month browsing the web, which translates to 2.6 hours a day. [8]. In consequence, ensuring a good web browsing experience is paramount.

A central determinant of quality of experience for web users is *page load time (PLT)*—the latency between when a user requests a page in a browser and perceives that the page has loaded sufficiently to be useful [9–12]. PLT plays an important role in determining whether a user will continue browsing a web page or abandon the task completely [13–15]. Even relatively small delays can be frustrating: studies show users begin to lose engagement with a task after delays over 250ms [16, 17]. Measuring user tolerance to delay, other studies show that

[1] This percentage represents HTTP traffic but excludes Internet video. It also includes email and data downloads using FTP, but we expect their contribution to be small.

even small increases in latency can have a profound impact on how users view a particular web site, adversely affecting their conception of the site's corporate image, and prompting a possible switch to its competitors [13–15].

Search engines incorporate how quickly a web site loads in their page rank computation [18], and Google has found that increasing web search latency by 100 to 400ms reduces the daily number of searches per user by 0.2–0.6% [19]. This reduction translates to a loss of eight million searches per day and therefore millions of advertisements that could have been served. Amazon, on the other hand, found that each 100ms delay in PLT cost them 1% of sales [20]. All these factors combined indicate that page load times are invariably tied with web page revenue figures, to the extent where a small improvement in latency will result in a pronounced increase in revenue from either sales or ads. It is no wonder that a relatively small decrease in PLT (8% on average) is sufficient to prompt Google to switch to a new application-layer protocol for the web [21], as we discuss in Chapter 2.

The importance of reducing web page load times has given rise to a growing interest in developing faster web and transport protocols. Until recently, the de facto protocol for delivering web pages to clients has been HTTP/1.1 [22], with TCP as transport. Although HTTP/1.1 supports pipelining, browsers generally do not pipeline HTTP/1.1 object requests. HTTP/1.1 consequently uses FIFO semantics to deliver one object at a time on a particular TCP connection, and the browser must queue requests for later web objects while it waits for an ongoing web object to arrive from the server. As a result, HTTP/1.1 exhibits a phenomenon called *head-of-line blocking*, where processing and transmitting a single web object blocks the progress of the rest of the objects in the page. To circumvent these limitations, browsers usually initiate more than one connection per domain to parallelise web downloads. Additionally, web site designers apply *domain sharding* [23,24], splitting web objects across multiple servers to further increase parallelism. Despite its inefficiencies, HTTP/1.1 persists in many web sites today.

Recent years have seen the emergence of new web protocols that address HTTP/1.1's shortcomings. HTTP/2 is beginning to experience wide spread adoption among web sites after its recent standardisation [25,26]. It offers tangible reductions in PLT by multiplexing several web objects onto one connection, thereby eliminating head-of-line blocking in the application layer. HTTP/2 also introduces the ability to prioritise web objects. Because a browser parses HTML incrementally, pausing to execute synchronous JavaScript for example, certain objects may block the download of others. HTTP/2 allows the client to annotate these important objects accordingly, so that the server can return them first.

Going one step further, Google has developed QUIC, an application layer protocol that combines web and transport-layer functionality in order to eliminate cross-layer inefficiencies and improve PLT. Google has incorporated QUIC into its Chrome browser with good results [21, 27–29]. QUIC moves away from TCP's traditional in-order delivery of packets to the application layer, and uses UDP-based out-of-order packet delivery to drive down single-

path PLT even further. It incorporates several other mechanisms for reducing PLT, the most important of which is setting up new connections with 0-RTT handshakes. Both QUIC and HTTP/2 have highlighted the need for a specialised transport which has a good awareness of the application layer's demands.

Despite the performance gains HTTP/2 and QUIC demonstrate, they overlook the changing landscape with respect to the kinds of devices that access the web today. The proliferation of mobile devices has prompted a substantial increase in the number of users browsing web pages from their phones or tablets. In 2017, more than 50% of global web page accesses originated from mobile devices [30,31], indicating that the percentage of web pages served to mobile phones has outstripped the percentage served to PCs. Similarly, of the 81 hours an average UK user spent on the web each month, nearly 60 of them were on smartphones [8].

A large proportion of mobile devices has more than one network interface, typically WiFi and 3G/LTE for phones and tablets. A device's operating system usually selects only one for establishing connections and downloading web pages. But, there may be a benefit to using *both* of them at the same time. The existence of multiple interfaces, and therefore paths, between the client and server creates path diversity, and an opportunity to deploy protocols that can pool bandwidth across both these paths.

Multipath TCP, or MPTCP for short, is one such protocol [32,33]. MPTCP splits a single data stream across the available paths in order to achieve a bandwidth equivalent to or even larger than TCP over a single path. By dynamically adapting the amount of traffic it sends via each path, MPTCP increases a flow's resilience to varying network congestion conditions, and allows for seamless implementation of mobility and multihoming. The question is: can we use MPTCP to reduce the time is takes to download a web page?

MPTCP has already demonstrated some benefit in reducing latency over the network. Apple's Siri sends a duplicate request using MPTCP via the WiFi and 3G/LTE interfaces on an iPhone in an attempt to elicit a response to a particular voice query from the interface with the fastest path [34]. Siri's use of MPTCP highlights a genuine interest in exploiting multipath for the purpose of reducing latency.

1.1 Problem Statement

The higher-level goal of this work is to examine whether we can reduce PLT on mobile devices by leveraging their access to multiple network paths. At first glance, it appears that we might accomplish this goal by simply running HTTP/2 over MPTCP without any modifications. MPTCP's design makes it well suited for spreading traffic from large long-lived flows across multiple congested links, in order to improve the aggregate bandwidth. On the other hand, MPTCP is less suited to cope with small, short-lived flows like web downloads. At heart, reducing web page load times is a latency problem. Adding more bandwidth using MPTCP may help when the download is in fact bandwidth limited. In contrast, problems may arise when the bandwidth is no longer the bottleneck, and the two paths concerned are heteroge-

neous. If one path experiences a much longer delay, or much smaller bandwidth than its counterpart, placing any packets on this slower path may delay critical web resources, and therefore increase the overall completion time. Furthermore, MPTCP subflows can exhibit packet reordering delays: slower packets on one path will block the delivery of packets arriving on the faster path to the application layer, delaying page loads even further.

In light of MPTCP's known limitations with respect to short-lived flows, is using MPTCP for the web on mobile devices actually beneficial? To answer this question, we aim to thoroughly investigate how modern web protocols like HTTP/2 behave when running over MPTCP with two paths. We focus on the case where the bottleneck bandwidths over the paths corresponding to WiFi and 3G/LTE are unchanging, and the RTTs are predictable. In other words, we will not examine cases where there are outages or mobility. We are particularly interested in examining page load times with MPTCP under a wide set of network conditions, especially on two paths with heterogeneous properties. For the cases examined, we would ideally like the completion times under MPTCP to be smaller than the single-path case on the best path. If that goal is not possible, then MPTCP should do no harm *vs.* the single best path alone.

We show that for lower bandwidths in the operational range of WiFi and 3G networks, MPTCP can indeed speed up page loads. As for the the cases where the download is no longer bandwidth limited, we do not want MPTCP to actively lengthen PLT, especially when the paths have mismatched delays. Unfortunately, as a consequence of MPTCP's path scheduling algorithms, we discover that this goal can be unattainable for certain configurations.

MPTCP uses its scheduler to distribute packets across the available paths. By default, MPTCP chooses the path with the shortest RTT from among the set of paths that have some free space in their congestion windows. If a path with a long RTT is the only one with free space at a particular instant, MPTCP will still choose to transmit a packet on this path. While this strategy works well and reduces reorder buffering at the receiver when increasing bandwidth is the main concern, we will show that it falls short in reducing PLT. In addition, because MPTCP stripes packets across both paths in a manner agnostic to the actual objects they correspond to, it cannot make informed decisions about their importance or their completion times. As a result, we expect to find that for the combinations of network parameters, MPTCP can *increase* PLT. We aim to classify all of the cases where MPTCP underperforms, and explain their causes via an in depth analysis of the interactions between the application and transport layers.

In the extreme cases of these scenarios, the client will be better off just downloading the web page on the faster path. Yet, we do not want to merely enable MPTCP in response to the right network conditions. Such a design would be fragile, since the wrong choice would either increase PLT or cause the web page to miss an opportunity for PLT reduction. Instead, we would like to enable MPTCP by default. Preferably, we should be able to glean its possible benefits while simultaneously preventing it from performing worse than single-path TCP

operating on the best path. As a result, we address each of MPTCP's limitations in turn, and employ techniques that make MPTCP more viable for web workloads under varying network conditions. In particular, we will present a new MPTCP scheduler that achieves shorter page completion times by deciding:

- which packets from each resource to send next so as to minimise page load time;
- which path to send the packets on;
- whether to restransmit missing packets on a different path.

To perform these actions, the MPTCP path scheduler must know which resources are more important, and hence should be transmitted on the faster path. The best method for determining object priorities is to determine which objects lie on the critical path for downloading, parsing and rendering the web page in the browser. As a result, we must understand the inherent dependencies between web objects and use them to construct a *dependency graph* for the web page in question, which will in turn allow us to learn the objects' priorities. We will not worry about how MPTCP might gain access to this application-layer information. In this work, we attempt to answer fundamental questions about whether we can improve PLT for multipath web downloads independent of the underlying socket API, and the separation between layers.

1.2 Thesis Roadmap

There are several intermediate steps we must take before we can examine multipath for the web. To begin, we describe the widely deployed web and transport protocols for downloading web pages, and the existing mechanisms for measuring and reducing PLT in Chapter 2.

To compute web page dependency graphs, evaluate MPTCP for the web, and then assess any improvements we devise, we must construct a practical experimentation framework. There are several methods for building such a framework, but because we favour schemes that generate reproducible results, we choose to emulate the download of web pages captured from real traces. Using emulation implies that the results we will obtain should not be considered predictive of absolute PLT values obtained in the real world. Rather, the results explain the different sources of delay when downloading a web page, and the relative ordering between different protocols and strategies in terms of how well they reduce web page completion times. Ultimately, our goal is to explain the detailed behaviour and broad phenomena exhibited by these protocols, not to predict in-the-wild PLT.

Towards this end, we develop the Page Completion Profiler (PCP), an emulation framework that collects traces from real web pages in order to simulate their download under an array of different network conditions, server architectures, and web and transport protocols. We present a detailed description of the PCP in Chapter 3, and highlight the design choices we have made to model web page downloads. PCP is a hybrid design: it uses ns-3 [35] to model various aspects of the client's browser, the server, and the underlying network, and incorpo-

rates server processing delays sampled from the web traces into the model. With these models in hand, we can use PCP to simulate and test possible multipath protocol designs, which focus on the reduction of web page load latencies.

On the way to understanding multipath web downloads, we first need to evaluate the single-path case, and make sure we understand all the interactions between the application and transport layers adequately. As we have stated, we will compare MPTCP with single-path TCP operating on the best path. But even for single-path TCP, we must apply some measure of experimentation to determine the combination of web protocol, prioritisation scheme and server architecture that achieves the best results. We note that these may not be consistent for all the web sites we examine, or the network parameters we select. With our profiling framework in hand, we analyse some state-of-the art web and transport protocols to examine their impact on PLT in Chapter 4.

In Chapter 5, we examine whether a specific recommendation for improving HTTP/2's performance can actually lead to reduced PLT across the board. Given that HTTP/2 eliminates the need for increasing parallelisation by sharding web domains, the general advice for web developers is to *unshard* their domains so that HTTP/2 can reduce PLT even further. We will examine whether unsharding does indeed offer substantial improvements in PLT for a large collection of web sites, and for different combinations of bandwidth and RTT.

Finally, we present a full investigation of downloading web pages using HTTP/2 running over MPTCP for a large set of web pages in Chapter 6. We uncover several regions of operation where MPTCP *increases* PLT with respect to single-path TCP on the best path. We further provide in depth explanations for the underlying causes of this increase. Lastly, to combat MPTCP's shortcomings, we implement and evaluate several mechanisms that improve PLT for these regimes and allow MPTCP to perform as well as or better than single-path TCP operating on the best path.

1.3 Contribution

In this work, we present the following contributions:

- We construct dependency graphs for 300 web sites sampled from the Alexa top 500 [36], with sufficient detail to drive emulation. For these, we use real traces profiled from the Chrome browser using a modified WProf [10]. Unlike several previous results in the area, we capture dependencies resulting from JavaScript events and timers.

- We develop PCP as an emulation framework that consumes the dependency graphs from our corpus of web pages and simulates their download under different network conditions, server architectures, and web and transport protocols.

- We thoroughly investigate the benefits of unsharding for HTTP/2 using our full collection of 300 web pages emulated under a large range of network conditions.

- We demonstrate that, although unsharding is generally helpful in reducing PLT, there are some combinations of web site designs and network conditions where unsharding actually *increases* PLT. We present several case studies that highlight the importance of prioritisation and the browser preloader to alleviate extra delays caused by unsharding for some web sites.
- We highlight a pathological side effect of TCP New Reno's congestion control scheme on unsharded page completions times.
- We perform an in-depth analysis of HTTP/2's performance over MPTCP, and identify limitations of several of MPTCP's core components. Namely, we show how MPTCP's RTT scheduler causes PLT to increase for at worst 70% of the web sites we examine when the second path's RTT is five times the first. Additionally, MPTCP's Linked Increase (LIA) congestion control algorithm may prevent a path from growing its window to its full potential when the bandwidths of the two paths are asymmetric. Finally, tail losses increase for MPTCP, where small objects are striped across two separate paths.
- We devise and evaluate several mechanisms for combating MPTCP's poor performance in the scenarios we've highlighted. Most importantly, we design the Stream Scheduler specifically for reducing PLT for web objects transmitted on two paths.
- We offer strong motivations for designing a future multipath web protocol in the style of QUIC, such that it combines web and transport-layer logic into an application layer protocol.

Overall, this thesis contributes a systematic exploration of existing web and transport protocols for the single and multipath cases. After isolating the deficiencies of the current multipath transport, we propose and evaluate several schemes that will allow the deployment of multipath for the web. With our improvements in place, we show that MPTCP should improve PLT in many cases, and barring that, it should perform no worse than single-path TCP.

Chapter 2

Background and Related Work

Users exhibit a strong preference for shorter PLT, and content providers' revenue depends on shorter page loads as well. As result, there exists a wealth of research and optimisation techniques geared at reducing page load times. To what extent can page load times be decreased? Singla *et al.* make the case that it should be theoretically possible for web communications to approach the typical latencies of the speed of light between the browser and web server [37]. In reality, web page latencies are several orders of magnitudes longer. Since the process of downloading and displaying a web page is complex and involves a myriad of protocols, application software and infrastructure acting together, each of these components can contribute to the overall delay, and increase page load times. To name a few sources of delay:

- Delays due to domain name resolution using DNS. Browsers may use techniques like DNS pre-resolution to prefetch links on a particular web page speculatively [38], thus removing a DNS lookup's contribution from the page load time by performing it earlier in the background.
- Infrastructural delays due to traversing physical links, which are exacerbated by suboptimal routing paths. To combat these delays, content providers typically use Content Distribution Networks (CDNs), which bring content closer to the user and reduce RTTs.
- End-to-end delays due to the overheads of web and transport protocols. We will discuss these in more detail in the following sections.
- Queuing delays and loss in the network, which will trigger packet retransmissions. One method to combat packet loss is to apply Forward Error Correction (FEC) techniques in order to save the round trip times required for retransmission [21, 27].
- Secure connection setup latencies due to the Transport Layer Security (TLS) protocol [39].
- A web page's structure itself may introduce inefficiencies, since the browser must parse HTML incrementally, pausing to execute JavaScript and resolve CSS rules. These actions introduce dependencies between web objects, which may result in some objects blocking the download of others, when ideally they could have been downloaded in parallel. We explain these dependencies in more detail in Section 2.4 and Chapter 3.

A full survey of these delays and the techniques used to mitigate them may be found in [40]. Our interest lies mainly in delays occurring in the application, which encapsulates the web object dependencies and web protocols, and in the underlying transport over the network. In particular, we wish to examine the interaction between the application and transport layers, and how it evolves when multiple paths come into play. To this end, the following sections will focus on currently deployed web and transport protocols. First, we examine the previous state of the web which still persists in many web sites today (Section 2.1). Then we discuss techniques designed as workarounds for these web protocols' limitations in Section 2.1.3. In Section 2.2, we move on to emerging web and transport protocols designed to address their predecessors' shortcomings. We examine state-of-the-art transport protocols that facilitate transmission over multiple network paths in Section 2.3. Finally, we tackle dependency-related delays in Section 2.4 and split-architecture browsers in Section 2.5.

2.1 Classic Web Protocols

Modern servers deliver web pages to client browsers using the application-layer Hypertext Transfer Protocol (HTTP) running over TCP. HTTP operates as follows: when a browser navigates to a web page, it uses TCP to send an HTTP `GET` request to the origin server for the corresponding HTML file. The server replies with an HTTP response containing the file on the same TCP connection (Figure 2.1). Once the browser receives and parses the file, it can issue subsequent HTTP `GET` requests for the other resources referenced in the file, *e.g.* the images, JavaScript or CSS files.

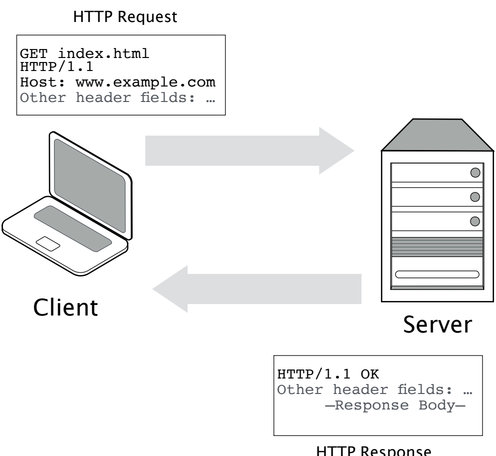


Figure 2.1: HTTP request issued from client, with associated HTTP response.

2.1.1 HTTP/1.0

HTTP has undergone several revisions since it was first introduced. Its initial design, HTTP/1.0 [41], was released in the late 1990s, and operated very simply: it issued a single resource request per TCP connection, then closed the connection after receiving a response.

We can see an example of this exchange in Figure 2.2(a). The client sends the first request to a server, which spends some processing time generating the response before transmitting it back to the client. The client requests a second resource on a new TCP connection, whose response is returned from the server after a shorter processing delay.

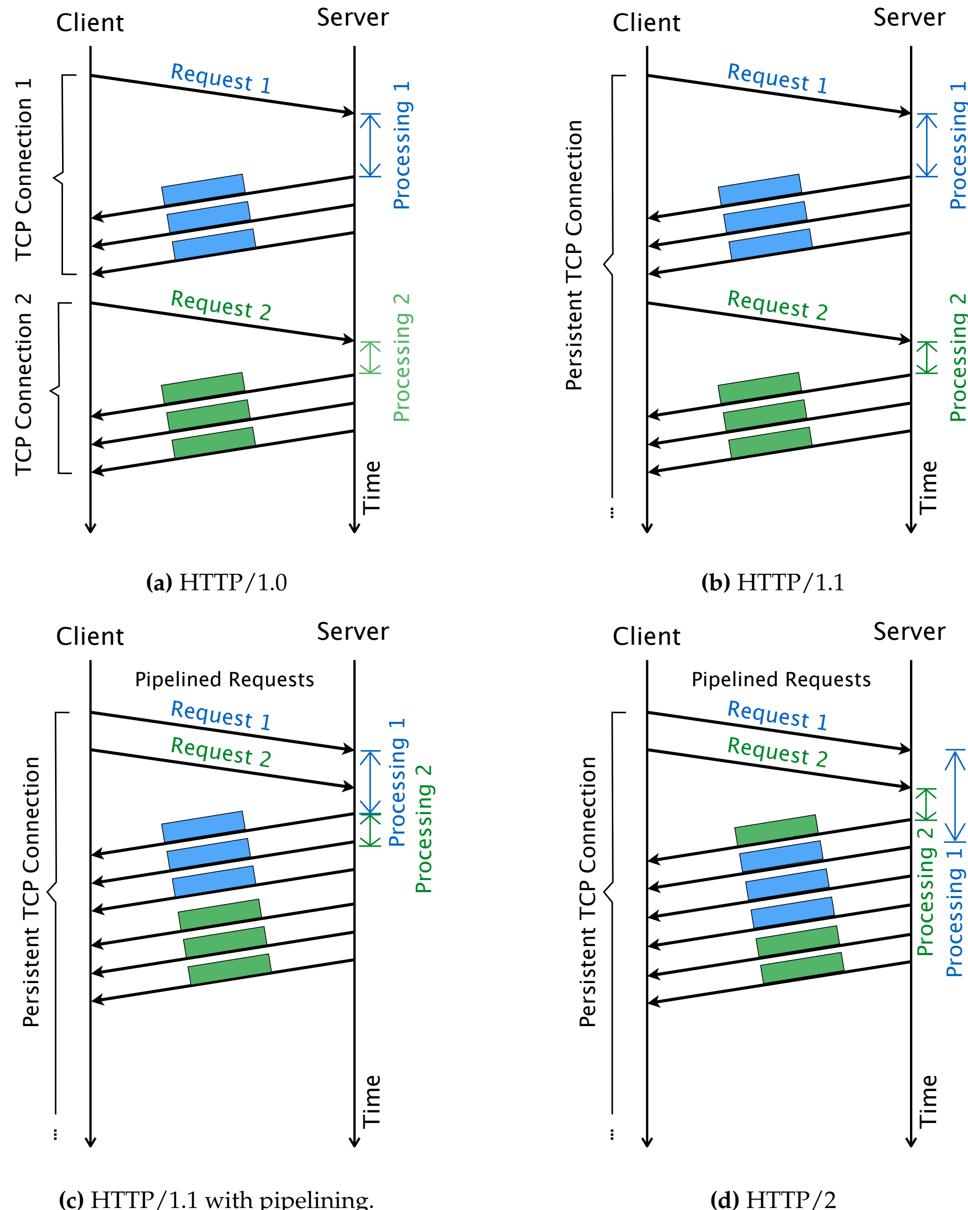
Undeniably this scheme is rather wasteful, for three reasons:

- **Slow-start overhead**: web objects are usually on the order of tens of kilobytes in size. Every time a new resource request opens a TCP connection, it incurs the cost of TCP's 3-way handshake, which consumes a significant portion of the transmit time due to its short duration. Furthermore, the connection will most likely never get a chance to ramp up to the available bandwidth because it may never leave slow-start before it terminates.
- **Resource consumption**: opening a lot of parallel TCP connections necessitates memory allocation at an endpoint in the form of socket buffers, and extra processing per connection.
- **Congestion**: parallel TCP connections must be multiplexed on a single physical outgoing link at the TCP client, which can become the bottleneck link. These TCP connections compete with each other for bandwidth, and cause congestion.

2.1.2 HTTP/1.1

In the face of these inefficiencies, HTTP/1.1 was released in 1999 as a successor to HTTP/1.0 [22]. Despite its release more than a decade ago, HTTP/1.1 has until recently been widely deployed as the principal protocol for the web. HTTP/1.1 does not close a TCP connection after serving one request/response pair, but rather uses persistent connections for downloading several web objects from any particular server [42] (Figure 2.2(b)). Additionally, HTTP/1.1 introduced request pipelining, where multiple requests for web objects can be transmitted consecutively without waiting for a response first (Figure 2.2(c)).

Unfortunately, in the absence of clear framing to facilitate the multiplexing of different HTTP responses on the same connection, a server must respond to requests in FIFO order. We see this clearly in Figure 2.2(c), which shows that the server must process incoming requests serially. If it were to process both requests in parallel, it would still need to complete sending the first response entirely before moving on to the second one, even if the second one is ready first. A slow response therefore blocks all other responses from being transmitted, *i.e.*, causing head-of-line (HOL) blocking at the server. Additionally, the server needs to reserve buffer space for the pipelined responses when processing in parallel, which may exhaust server resources, and make it vulnerable to DoS attacks. Finally, if a failed response terminates a TCP connection, the client will have to issue new requests for all subsequent resources again, which may result in duplicate processing on the server end. Due to these serious inefficiencies, pipelining is usually disabled in modern browsers, despite its potential to improve HTTP performance.



(a) HTTP/1.0

(b) HTTP/1.1

(c) HTTP/1.1 with pipelining.

(d) HTTP/2

Figure 2.2: Frame exchanges under the different variants of HTTP.

Without pipelining, sending one request at a time and waiting for its response on a persistent connection forces the sender to queue requests, and takes far too long. As a result, browsers usually open multiple concurrent TCP connections to a particular server in order to parallelise HTTP requests. For example, Chrome uses six connections to connect to each server with a unique domain name.

2.1.3 Optimisations

Working within the confines of HTTP/1.1's limitations, web page developers have introduced various workarounds and optimisations in an attempt to speed up page load times. Additionally, developers can use tools like Google's PageSpeed insights [43] to analyse their web pages and apply its resulting optimisation suggestions. The PageSpeed module [44] goes even

further, as a server module that helps remove inefficiencies by re-writing web pages to reduce their load times. The following section gives an overview of some of the optimisations applied to web pages running over HTTP/1.1, and some of the tradeoffs they introduce.

Domain sharding In modern web pages, there can be on the order of a hundred resources that must be downloaded per page. If all of the designated connections to a domain are busy, requests to this particular domain end up in a queue in the browser, incurring long delays. As a workaround, content providers apply a mechanism called domain sharding, where instead of serving content from one origin, they spread the page's resources across multiple subdomains, effectively circumventing the browser's connection limit, and increasing the number of parallel TCP connections. Domain sharding can hurt performance due to the additional DNS lookups required to resolve the sharded subdomains. Additionally, the usual drawbacks of initiating parallel short-lived TCP connections apply here. The larger number of connections established by the browser to these subdomains may experience self-congestion, and connection setup costs become more pronounced as web objects are spread across multiple connections.

Reducing HTTP Requests To reduce network overhead costs, the most logical option is to reduce the number of issued HTTP requests, or even eliminate them entirely. Web developers have devised several methods for achieving this goal, some of which include:

- **Resource Coalescing** where multiple JavaScript and CSS files are safely concatenated without affecting the behaviour and the execution of the code as long as execution order is maintained.

- **Image Spriting** where several images are combined to form one large composite image. CSS can later be used on the client side to access smaller sections of the image and display them.

- **Resource Inlining** eliminates all outbound requests. One example is to use `data` URIs, which embed small files such as icons within the main HTML file. CSS files and JavaScript may be inlined as well.

- **Resource Caching** to reduce consecutive requests for resources which have been downloaded in the recent past. Web developers typically use cache control directives to inform a browser how long it can store a particular object. Section 3.7.5 discusses caching in more detail.

Although reducing HTTP requests using these techniques helps reduce overall page download latency, there are several caveats worth discussing. Both resource coalescing and image spriting may result in sending unnecessary resources, for example in the case of a large JavaScript library where the client only uses some of its enclosed methods. Similarly, these techniques may increase the complexity of the design by requiring additional processing, as is the case of image sprites, which need CSS support.

As for inlined resources, they now form part of the web page and cannot be cached as separate entities by the browser or CDN. As a response, Mickens *et al.* developed Silo [45], which manages to aggressively inline all JavaScript and CSS files without sacrificing cache operation for these objects. A Silo-enabled web server breaks inlined HTML into addressable chunks, and only sends chunks that do not reside in the client's chunk cache as a response to a particular HTTP request.

Reducing Resource Sizes This technique reduces network traffic by reducing the size of the objects transmitted, an idea which complements reducing the number of HTTP requests. Developers are encouraged to compress images, and minify CSS, JavaScript and HTML. Minification compacts the files involved by removing unnecessary or redundant data without affecting the overall semantics of the contents, *e.g.* by reducing code comments and formatting, removing unused code, using shorter variable and function names, and so on.

Prioritizing Visible Content Arguably, the most important part of a web page is the contents immediately visible in the browser's viewport without scrolling. Web developers typically attempt to reduce the time it takes to download and render this "above-the-fold" content. Typical approaches include structuring a web site such that it downloads objects which appear at the top of the page first, and reducing the number of blocking JavaScripts at the beginning of the page.

Connection setup overhead HTTP/1.1 uses multiple TCP connections to download web objects from different servers. Since most web objects are rather small, on the order of tens of kilobytes in the median, the duration of an HTTP transaction is mostly consumed by TCP's three-way handshake. Both TCP Fast Open [46] and ASAP [47] attempt to reduce page load times by targeting connection setup overhead. They retain the IP source spoofing protection established by TCP's three-way handshake by authenticating the client in a prior step.

TCP Fast Open is the simpler of the designs. An initial connection to the server allows it to transmit a cookie back to the client, which is the client's IP address encrypted under a secret key. Subsequent connections to the server append the cookie and the actual HTTP request to the data portion of the SYN packet. If the server verifies the cookie, it returns the HTTP response in the SYN-ACK, thereby reducing connection overhead by one full RTT. ASAP, on the other hand, uses a provenance verifier to authenticate the client, and issue a certificate to the client indicating that it is reachable at a particular IP address. It also attempts to merge connection setup with DNS resolution.

Ultimately, the lessons learned from applying these optimisation techniques, and from understanding their limitations, have informed design decisions that have been applied to subsequent web protocol implementations, our next topic to review.

2.2 Emerging Web and Transport Protocols

From the previous discussion, it appears that HTTP/1.1 would benefit greatly if it were able to multiplex different streams separately onto a smaller number of connections. As a possible solution, HTTP/1.1 could perform the multiplexing in the transport layer, using the Stream Control Transmission Protocol (SCTP) [48]. Packets transmitted using SCTP are divided into several chunks belonging to different streams. Each chunk has its individual stream identifier and stream sequence number to allow for in-order reassembly of packets in a stream.

SCTP is most commonly used for telecoms signalling traffic, where control messages that set up and tear down connections correspond to independent streams. A proposal to use HTTP over SCTP is outlined in [49]. Nonetheless, SCTP adoption as a transport protocol has been rather slow, and is therefore an obstacle to widespread deployment of HTTP over SCTP.

2.2.1 HTTP/2

HTTP/2 has recently been standardised to address HTTP/1.1's shortcomings and to decrease page load times [25]. HTTP/2's design builds on SPDY [26, 50], a protocol originally proposed by Google and implemented in its Chrome browser.

Instead of implementing stream multiplexing in the transport layer, HTTP/2 adds a framing layer between the conventional HTTP functionality and TCP. It treats each HTTP request/response pair as an independent stream marked with a specific identifier, where a stream is bidirectional sequence of frames exchanged between the client and server.

An HTTP/2 frame, shown in Figure 2.3, allows for the correct reassembly of packets from different streams, as marked by the stream identifier. The figure shows two types of frames commonly used for HTTP/2. The `HEADERS` frame contains a compressed HTTP header for either a request or response. It is used to set up a new stream and notifies the server of the stream's priority (see Section 3.5.3 for more details). The second type is the `DATA` frame which contains the HTTP response payload. Because HTTP/2 clearly delineates the streams in a TCP segment, it can multiplex several concurrent HTTP requests and responses on the same TCP connection, thereby eliminating HOL blocking at the application level.

With HTTP/2, servers can process pipelined requests in parallel and return the responses out of order without the need for extra buffering, as seen in Figure 2.2(d). In our example, suppose Response 1 is of higher priority than Response 2. With HTTP/2, the server can begin transmitting packets from Resource 2 immediately after processing completes. When Response 1 is finally ready, the server can easily transmit it without completing the transmission of Response 2.

HTTP/2 attempts to reduce page load latency by limiting the number of TCP connections to one per domain, over which it can send HTTP requests without delay. Transmitting resources on a single connection which has already ramped up its congestion window is more beneficial than spreading traffic over multiple connections still in slow-start. For a single connection, slow-start is less of a limiting factor. Furthermore, HTTP/2 can coalesce traffic onto

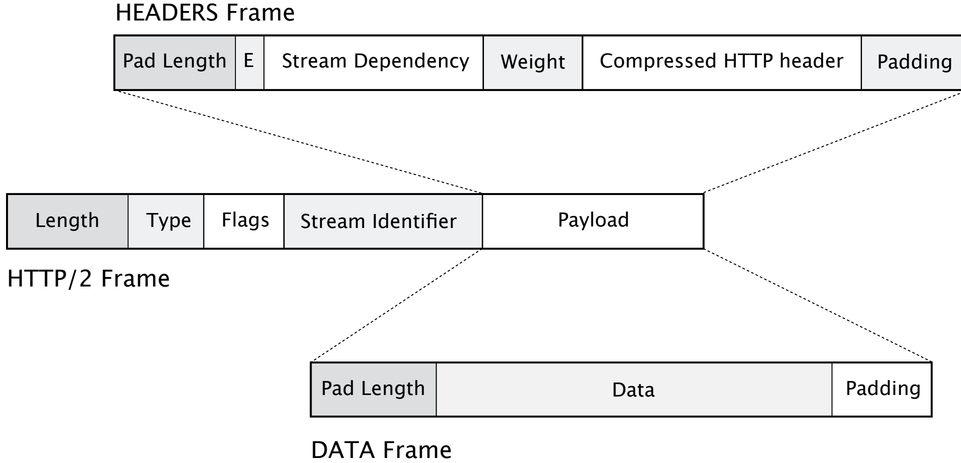


Figure 2.3: The format of two types of HTTP/2 frames: `HEADERS` and `DATA`.

this single connection, reducing any congestion that might arise from an aggregate of multiple competing connections. HTTP/2 reduces the load of handling large numbers of concurrent connections on the server. Finally, certain service guarantees become easier to uphold and reason about when communicating to the server via a single channel, *e.g.*, request prioritisation and header compression.

In addition to multiplexing streams, HTTP/2 supports several other techniques intended to reduce page load times:

- **HTTP Header Compression**: HTTP headers are redundant. By compressing HTTP request and response headers, HTTP/2 reduces the number of transmitted bytes, and can fit several HTTP requests into one TCP segment.

- **Stream Priority**: If the network is bandwidth limited, assigning higher priorities to critical streams would allow the server to tackle them first and respond quickly, as opposed to less critical streams that can afford to wait. For example, a sender might assign a higher priority to a CSS file that is required to render a web page, as opposed to one of the images within the page.

- **Stream Flow Control**: To remove stream contention on a shared TCP connection, HTTP/2 implements a flow-control scheme at both the stream and connection levels. A client maintains a receive window per stream, in addition to a per-connection receive window that combines the stream windows. It notifies the server when a stream can receive more data using a `WINDOW_UPDATE` frame.

- **Server Push**: HTTP/2 allows a server to speculatively send multiple responses in reply to a single HTTP request using a `PUSH_PROMISE` frame. If the server knows which resources the client will request next as a result of downloading a primary resource, it can save

several round trips by preemptively pushing the resources to the client.

2.2.1.1 HTTP/2 Performance

Early evaluations of HTTP/2 have been performed with its precursor, SPDY. Initial measurements for SPDY as presented in Google's original whitepaper [50] showed improvements in PLT of up to 27% – 64% for HTTP connections. On the other hand, subsequent studies have concluded that SPDY and HTTP/2's performance improvements are not consistent, and vary greatly depending on the web pages examined, and on the network conditions [10, 51–55]. For the most part, HTTP/2 only offers a minor win over HTTP/1.1, and is very load dependent. While Wang *et al.* show that HTTP/2 can indeed halve the completion times for some web sites under specific network conditions [51], on average HTTP/2 can achieve a reduction in PLT between of 0.8% to 20% at most [51–53]. Even then, we suspect that improvement figures from [51] and [53] are slightly inflated because their experiments use a single origin server, and do not create a new TCP connection to each server domain. Additionally, the origin server does not incorporate realistic server processing delays when responding to requests from the client. For some web pages, the same studies show that HTTP/2 may actually *increase* completion times, despite its intended design goal.

The extent to which HTTP/2 is beneficial depends on factors including bandwidth, RTT, packet loss rate, and web resource sizes and counts. We summarise some of the findings with respect to these factors:

Packet Loss The consensus among the studies is that connections with high packet loss rates increase PLT for HTTP/2 when compared to HTTP/1.1, especially when the downloaded web sites contain large objects. Although HTTP/2 eliminates HOL blocking from the application layer, TCP connections will still stall while they recover from packet losses. On a single connection, a packet retransmission for one web resource will indefinitely block the completion of all other web resources until the retransmission succeeds. As a result, HOL blocking has moved to the transport layer instead.

Low Bandwidth HTTP/2 outperforms HTTP/1.1 at low bandwidths because it reduces packet drops. Because HTTP/2 uses one connection as opposed to several per domain, self-congestion between the multiple connections is eliminated, and packet drops are decreased, thereby reducing completion times.

High RTT In the cases where packet losses are low and the RTT is long, HTTP/2 provides a win over HTTP/1.1 because it amortises connection setup time. Instead of performing slow-start on multiple connections with long RTTs, HTTP/2 reduces PLT by transmitting all of the objects on a single connection that has grown its congestion window sufficiently.

In addition to the core attributes of HTTP/2, several of the studies examine some of its other features. In terms of prioritisation, [51] and [53] both indicate that applying stream priorities can only offer a minor improvement over regular HTTP/2. Wang *et al.* [51] also compare prioritisation schemes, for example using the MIME type of objects or their depth in the web

page's dependency graph. They conclude that neither scheme can achieve a significant win, due to implicit prioritisation of resources in the web page structure itself.

As for server push, naively pushing all web objects to the client eliminates several RTTs involved in requesting them individually, and produces larger reductions in PLT, as seen in [51, 52, 55]. But, there exists a tradeoff between aggressively pushing all of a web page's objects and wasting bandwidth on objects already cached by the browser. Due to a lack of guidance in the HTTP/2 standard about how to push objects to the client, Wang *et al.* [51] experiment with two other push methods in addition to the naive one. The first is to use the object's "embedding-level" in the HTML page, where the server pushes objects up to one embedding level to the client on receiving a particular request. The second is to push objects up to one dependency level, *i.e.*, the server pushes a web resource's immediate children from the web page's dependency graph. Although the dependency level policy does not achieve as as great a PLT reduction as the other two schemes, it in facts saves bandwidth by reducing the number of pushed bytes.

When examining HTTP/2 for mobile networks, Erman *et al.* [56] show no clear winner in page load times between HTTP/1.1 and HTTP/2 when operating on cellular 3G and LTE networks and using an HTTP/2 proxy. They argue that while a user waits for some "think time" between navigating to new web pages, the cellular radio on their device transitions to the idle state. Long promotion delays for transitioning from idle to active states again will cause spurious retansmissions in TCP due to incorrect RTT estimates. While HTTP/1.1's multiple connections can mitigate this problem, HTTP/2 suffers because it uses a single connection.

We believe that the use case examined by [56] is not common in real world deployments of HTTP/2. The browser does not typically establish one connection to a proxy over which it downloads all resources from multiple consecutive web sites. Rather, it creates a new TCP connection for every domain it sends requests to. Idle times due to web object dependencies will still be an issue, but these are typically on the order of hundreds of milliseconds, and not long enough for the radio to become idle again.

Finally, domain sharding has the expected negative impact on HTTP/2, as demonstrated in [51]. Sharding spreads resources over multiple connections, thereby preventing HTTP/2 from multiplexing the maximum amount of resources. Google has already begun encouraging content providers to move away from the practice of sharding to combat this effect [23], yet resource consolidation is a rather difficult endeavour, especially given that web pages are accessing third-party resources more often.

In this work, we attempt to answer the question of whether some of the performance wins expected of HTTP/2 can be improved by unsharding domains. Varvello *et al.* [24, 57] analyse live web sites for HTTP/2 adoption, demonstrating that there is significant sharding exhibited in the wild. Furthermore, Elkhatib *et al.* [52] show that sharding hurt HTTP/2's performance for three deliberately sharded websites running on their testbed. We will expand on this study

to perform a more detailed analysis of 300 web sites, and delve into the interactions of the transport protocol and application layer.

2.2.2 QUIC

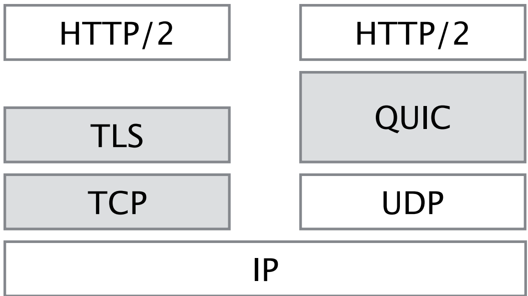


Figure 2.4: HTTP/2 running over the traditional network stack, vs. QUIC's new architecture.

The previous section has demonstrated that TCP's reliable in-order delivery of packets to the upper layer is detrimental to the performance of HTTP/2. A single packet loss containing data from a particular stream will cause HOL blocking, and prevent other streams from progressing until TCP retransmits and recovers the missing packet for that stream.

The interaction of several application-layer mechanisms with TCP seem to incur unnecessary overheads which could be avoided with a dedicated transport protocol that understands the requirements of HTTP/2. Take, for example, TLS [39] running over TCP: in addition to TCP's 3-way handshake, TLS needs two extra RTTs to exchange cryptographic parameters required for a secure connection.

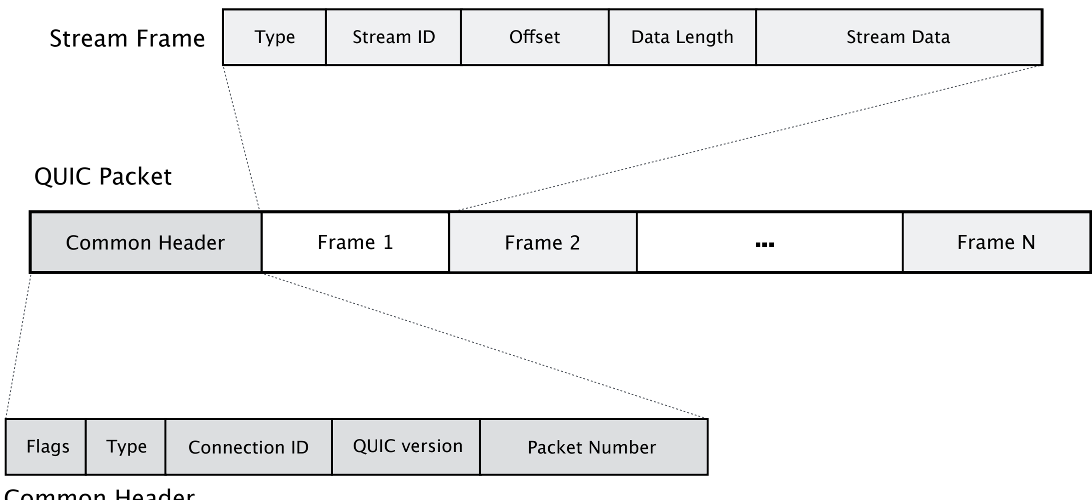


Figure 2.5: A typical QUIC packet containing `STREAM` frames to transmit data.

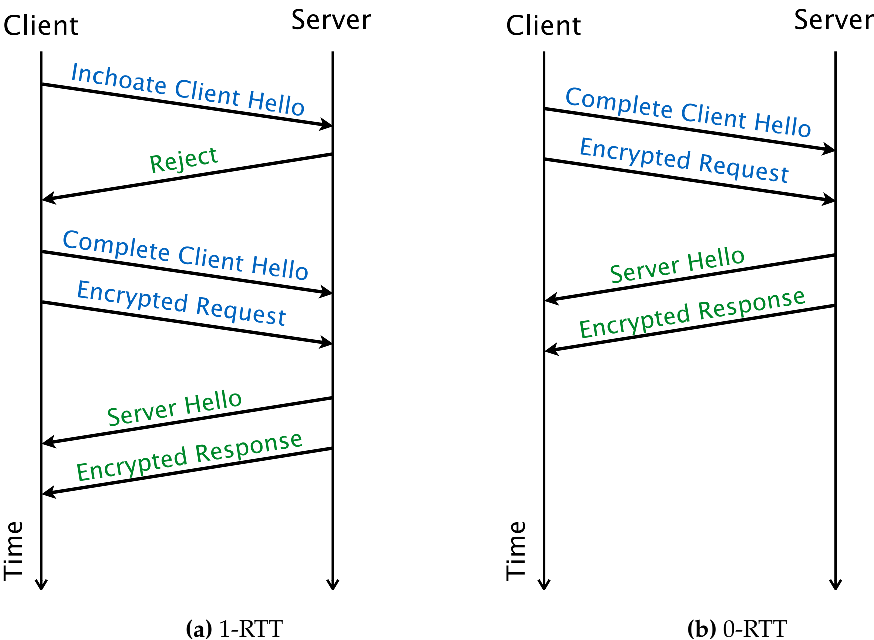
To alleviate these avoidable overheads, Google has proposed a new protocol called QUIC: Quick UDP Internet Connections [21, 27–29]. QUIC was built from the ground up with HTTP/2 in mind, and combines several optimisation strategies in order to produce a secure,

multiplexed transport over UDP that functions like TLS running over TCP (Figure 2.4). Using UDP as the generic transport protocol allows QUIC to apply its custom transport functionality at the application layer and still be deployed on today's Internet, traversing middleboxes without any problem.

Figure 2.5 shows an example of a QUIC data packet, which contains multiple stream frames, each corresponding to a single HTTP/2 request and response pair [58]. By virtue of multiplexing stream frames over UDP, QUIC decouples stream processing from the underlying protocol, thereby eliminating HOL blocking. A lost QUIC packet only affects the streams it encloses, which QUIC retransmits. QUIC will reassemble frames from the streams which have not experienced any loss, allowing them to make forward progress without blocking.

Applying lessons learnt from experience with Chrome, QUIC incorporates several other optimisation techniques which we discuss below.

0-RTT Handshakes: QUIC attempts to achieve a 0-RTT connection establishment latency, whilst at the same maintaining the security and source spoofing protection offered by TCP's 3-way handshake and TLS. In a manner similar to TCP Fast Open and ASAP, QUIC establishes the required security parameters in a previous connection to the server. The very first time a client connects to the server, the server performs the correct steps to authenticate it. Any subsequent connections from the client use these cached credentials to immediately send encrypted requests to the server, circumventing any connection setup [59,60].



(a) 1-RTT

(b) 0-RTT

Figure 2.6: Messages exchanged to establish a QUIC connection for the 1-RTT or 0-RTT scenarios.

Figure 2.6 shows this process in more detail. The initial handshake to a new un-cached origin server requires a full RTT for connection setup, as shown in Figure 2.6(a). On sending an inchoate *client hello* message, the client receives a *reject* message from the server which contains several authentication primitives like the server's public key, certificate chain, and source ad-

dress token (details are found in [21, 61]). After the client authenticates the server, it responds with a complete *client hello* message that contains the client's public key. If the handshake is successful, the server will respond with a *server hello* message. The client caches the server's public key and source address token for use in future connections to the same server.

If the client establishes a new connection to the same cached server, it can immediately begin sending QUIC frames after sending a complete *client hello*, as seen in Figure 2.6(b). Therefore, establishing connections to cached servers requires 0 RTTs.

Flexible Congestion Control : Although QUIC uses TCP CUBIC [62] as its default congestion control mechanism, QUIC implements pluggable congestion control to leave room for experimentation with alternative protocols. It also adds richer signalling information to its headers to better inform the congestion control mechanisms.

Packet numbers are monotonically increasing and never repeat, *i.e.*, when a packet is dropped, its retransmission has a new packet number. Unique packet numbers allow a QUIC sender to distinguish between ACKs from retransmissions and ACKs from the original transmission (effectively avoiding TCP's ACK retransmission ambiguity [63]).

ACK frames also encapsulate the duration between the receipt of a frame and transmitting its acknowledgement, which allows the sender to calculate more precise RTT estimates. Finally, QUIC acknowledgement frames operate like TCP SACK [64]. They mark the greatest packet number seen, but additionally support up to 256 ACK ranges, allowing QUIC to keep more data on the wire in the face of reordering and loss than TCP with SACK.

Stream and Connection Flow Control: QUIC performs per-stream flow control to match the HTTP/2 specification. A QUIC receiver advertises the byte offset per stream up to which it is willing to accept data, in addition to a per-connection receive window which is the amalgamation of all the streams. As the receiver delivers stream frames to the upper layer, it sends a special `WINDOW_UPDATE` frame back to the sender to indicate an increased byte offset for that stream. Applying unified flow control prevents undefined interaction between HTTP/2's stream based flow control and TCP's flow control in conventional HTTP/2 running over TCP.

Connection migration: instead of using the 4-tuple to identify connections, QUIC uses a 64-bit Connection Id (CID) which allows connections to seamlessly move to new IP addresses.

Packet authentication and Encryption: QUIC supports full encryption of packet payloads. Although some parts of the QUIC common header are not encrypted, the entire QUIC packet is still authenticated by the receiver to prevent packet injection or manipulation by malicious third parties.

Preliminary studies on QUIC's performance have shown that QUIC generally outperforms HTTP/2, especially on connections that experience high packet loss rates or long RTTs [65–67]. Google performed an Internet scale study of QUIC, which it had deployed on its front end servers and several client applications (Chrome on the desktop, and YouTube and Search on mobile devices) [21]. On average, they found that QUIC reduces search latency by 8% on the

desktop, and 3.6% on mobile devices, with similar numbers for video latency. The improvement for search latency increased in response to increasing the RTT. The study indicates that the 0-RTT handshake plays a crucial role in improving PLT under these long RTT regimes, with QUIC's loss recovery mechanisms and improved signalling partly responsible as well.

2.2.3 BBR

The web protocols we've examined so far have used a variant of TCP as their transport, which relies on packet loss to alert the sender about congestion on the link. But, packet loss is often a poor indicator of congestion. Transport protocols that depend on loss to signal congestion (*e.g.*, TCP New Reno and CUBIC) either do not fully utilise the available link capacities when link buffers are small [68], or suffer longer delays due to bufferbloat [69].

BBR congestion control, proposed by Cardwell *et al.* [68, 70] and currently deployed on Google's B4 network, is a new congestion protocol that has been demonstrated to increase utilisation and decreases latency when compared to TCP CUBIC. It moves away from the loss-based paradigm, and conceptualises congestion as the state of sustaining more packets in flight than the path's bandwidth-delay product (BDP). We see where a connection first encounters congestion in Figure 2.7. As the number of bytes in flight increases from zero, the packet delivery rate increases without affecting the connection's RTT. But, when the number of bytes in flight exceeds the BDP, the connection becomes congested. Network buffers fill up, prompting a corresponding increase in RTT.

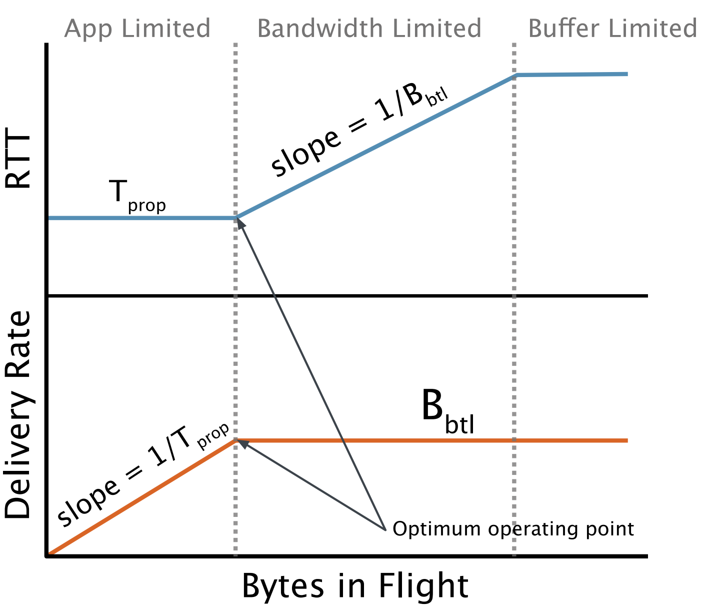


Figure 2.7: Relationship between RTT and packet delivery rate *vs.* number of bytes in flight.

Since we can fully characterise a connection using its propagation delay (T_{prop}), and the bottleneck bandwidth (B_{btl}), BBR estimates these values using measurements of the connec-

tion's maximum packet delivery rate [71] and minimum RTT. It uses the two metrics to build a model of the network path, which BBR uses in turn to seek the best operating point with high throughput and low delay, as shown in Figure 2.7. BBR then adjusts the transmission rate and the maximum amount of bytes in flight at any time in order to reach this optimal point. BBR sets the transmission rate to match the bottleneck bandwidth, and keeps BDP bytes in flight to maintain a full pipe.

At the core of BBR's steady-state operation is a control loop that probes for the value of the bottleneck bandwidth, and, with much less frequency, the propagation delay, which is used to compute BDP. BBR must send faster to discover if B_{btl} has increased. Likewise, BBR must send packets at a slower rate to discover any changes in T_{prop}. BBR therefore runs periodic, sequential experiments, where it either increases or decreases the send rate in order to obtain fresh estimates of these two metrics. Each round of increase in the number of bytes in flight is accompanied by a round of decrease, in order to drain any potential queues that form in the network.

Finally, at the start of a connection, BBR grows its congestion window (and subsequently the bytes in flight) exponentially to perform a binary search for B_{btl}. The growth continues until the estimate for B_{btl} plateaus (see Figure 2.7), a signal to BBR that the bottleneck bandwidth has been reached. Once B_{btl} is found, any excess queues built up are drained by reducing the transmission rate to half its value at startup, thereby reducing the bytes in flight back to one BDP in one RTT.

If our goal is to reduce PLT, using BBR for transport seems like an attractive alternative to TCP. However, its design had only been standardised towards the conclusion of this work. We will discuss the potential benefits of using BBR as we go along as a possible future transport for web downloads.

2.3 Multipath Protocols

Although HTTP/2 and QUIC are steps in the right direction, they fail to take advantage of a key feature of modern wireless devices: the ability to access web pages via multiple interfaces (3G/LTE and WiFi for example). In theory, adding a second path may increase the web download's aggregate bandwidth if the paths do not share a bottleneck, and as a result, decrease the overall time required to download a web page.

Two transport protocols have been developed to facilitate communication over multiple paths. The first is Concurrent Multipath Transfer SCTP (CMT-SCTP). CMT-SCTP is a multipath extension of SCTP that takes advantage of SCTP's built-in multihoming capabilities to send packets on multiple physical paths corresponding to one logical connection [72]. It was originally developed with no resource pooling and congestion control was performed separately on each path. Coupled congestion control was added in more recent years. Once again, we will not go into the details of CMT-SCTP due to its limited adoption for web traffic, but we will discuss some of its path schedulers in Section 2.3.2. We now turn to the second multipath

transport protocol.

2.3.1 MPTCP

Multipath TCP, or MPTCP for short, is the current state-of-the-art multipath extension of TCP [33]. It allows a TCP sender to initiate additional subflows if it detects the availability of other alternative paths (*e.g.*, the existence of a second interface on a mobile device). The goal of MPTCP is to pool bandwidth across these subflows. Using the linked increase congestion control protocol, it shifts traffic away from congested paths to less congested ones [32, 73]. We describe the details of MPTCP's functionality and its congestion control in Sections 6.2.2 and 6.2.3 respectively.

MPTCP has been demonstrated to improve throughput for bulk transfers in various settings, from data centres to mobile devices [32,74–76]. Additionally, Paasch *et al.* have investigated MPTCP's ability to allow seamless handover between WiFi and 3G/LTE [77]. Ultimately, MPTCP works best for large bulk transfers over long-lived flows. With small, short-lived flows, the cost of subflow setup may outweigh any potential benefits, especially since MPTCP cannot tell in advance which path will have the lowest delay. It may end up sending all of its traffic on one subflow, which may never leave slow-start.

Prior work evaluates the limitations of using MPTCP for short transfers on mobile platforms [78,79], and highlights MPTCP's shortcomings in this area. But this thread of work does not specifically evaluate MPTCP for web traffic, which is influenced greatly by the dependencies between web resources. Nikravesh *et al.* [80] perform large scale user study of MPTCP on mobile devices, and investigate web downloads as part of their analysis, but their investigation is performed with HTTP/1.1, which generally performs worse than HTTP/2 on MPTCP, since it opens up to six connections per domain.

Given that they coalesce several small flows into a large stream-based flow, protocols like HTTP/2 and QUIC may interact better with MPTCP. More recently, Han *et al.* investigate mobile web performance over HTTP [81], but mostly investigate a scenario where the two paths have equal RTT and bandwidth. They later present a more in-depth analysis of mobile web performance using HTTP/1.1 and HTTP/2 over MPTCP [82], where they highlight some of MPTCP's deficiencies with respect to web traffic. Han *et al.* suggest switching between MPTCP and single-path TCP based on a cost-benefit analysis. We present a more in-depth analysis of the areas where MPTCP hurts PLT. Furthermore, with the modifications suggested in this work, we show that we can switch MPTCP on by default without adversely affecting completion times.

In parallel with our work, De Coninck *et al.* have developed a multipath version of QUIC (MPQUIC) in [83]. While MPTCP preserves TCP's in-order semantics for packet delivery, which may cause HOL blocking in the cases of packet loss on some paths, MPQUIC applies out-of-order packet delivery to the application layer. MPQUIC outperforms MPTCP in scenarios of high packet loss. But, evaluations of MPQUIC were only performed with bulk and

short transfers, and not web traffic. De Coninck *et al.* argue that multipath is not useful for short transfers because connection handshake latency is too long. In this work, we show that transferring web pages over two paths can indeed decrease PLT when the "best" interface, *i.e.* the one with the shorter path RTT, is chosen to establish new connections.

2.3.2 Path Schedulers

In the presence of multiple available paths on which to transmit web objects, or more generally a stream of packets, how to schedule these packets on paths that may be asymmetric in terms of bandwidth or delay is an area of active research. A multipath protocol typically invokes its scheduler when any of its available subflows has enough space in its congestion window to send a new packet, or when new data arrives from the application layer. The simplest scheduling scheme is round robin. Although this design attempts to distribute packets across the available paths in an alternating fashion, any mode of scheduling that sends packets on a subflow as soon as an ACK frees up some congestion window space is typically governed by the subflows' ACK clocks, not the scheduling scheme itself. Only when the flows are application limited will the scheduler need to make a real choice. After experiencing an idle period, the scheduler must choose between multiple available subflows upon the arrival of new data.

Ultimately, any scheme that overlooks some measure of the "goodness" of the available paths will be outclassed by any scheme that does. For example, MPTCP's Linux implementation uses a scheduler which, by default, chooses to transmit on the path with the minimum RTT from among the likely candidates [76,84]. By choosing the path with the shortest RTT, this scheduler reduces packet delay in application limited flows [85].

2.3.2.1 Receive Buffer Blocking

Even with the minimum RTT scheduler, Raiciu *et al.* expose a serious MPTCP limitation that manifests when packets arrive out-of-order at the receiver as the result of transmitting over heterogeneous paths [76]. They show that an MPTCP host typically requires a larger receive buffer than its TCP counterpart to store out-of-order packets and facilitate in-order delivery to the application layer. When the receive buffer is not large enough, the gaps in sequence space corresponding to packets that should arrive over the slowest path consume a large proportion of the buffer's capacity. Consequently, the transmission process experiences HOL blocking, remaining stalled until the packets sent over the slowest path arrive, filling the gaps, and freeing up space in the receive buffer. As a workaround, Raiciu *et al.* [76] suggest opportunistic retransmission and penalisation (RP), which entails retransmitting the leftmost packet in the window on a faster subflow in order to fill in gaps in the receive buffer and kickstart the transmission again. Additionally, the offending slower subflow is penalised by halving its congestion window and setting its slow-start threshold to the reduced window size.

The RP mechanism is reactive in nature, designed to detect and mitigate HOL blocking in the receive window. In contrast, other approaches minimise the out-of-order arrival of packets using a more appropriate scheduler design, which attempts to schedule packets so that they

arrive in-order at the receiver to reduce HOL blocking. Solutions range in how dynamically they react to changes in the underlying network conditions on the available paths. Delay Aware Packet Scheduling (DAPS) assigns a group of packets to each available subflow according to the ratio of their RTTs at the beginning of a scheduling cycle [86, 87], when packets are dequeued from send buffer. Because it determines the fate of several future packets based on current network conditions, DAPS cannot dynamically adapt to changes in the network.

On the other hand, the out-of-order transmission for in-order arrival scheduler (OTIAS) assigns a new packet to a subflow as soon as it arrives at the send buffer [88], choosing the subflow with the smallest time between scheduling the packet and its arrival at the receiver. OTIAS enqueues scheduled packets to subflow level buffers even if the subflows are not ready to send these packets yet, such that they may do so in the future. While more adaptive than DAPS, OTIAS still prematurely binds packets to subflows according to network conditions that may change.

Finally, the BLocking ESTimation scheduler (BLEST) adapts more dynamically to changes in the network [89]. BLEST uses the fact that the send buffer is a reflection of the receive buffer to its advantage, and targets send buffer blocking. It avoids filling the send window with packets to be transmitted over the slower subflow. Instead, it waits for the faster subflow to free more space in its window so that it can send more packets during the slower subflow's RTT, allowing it to free up space in the connection level send buffer. By determining the scheduling decision on a packet by packet basis when it is time to transmit them, BLEST adapts to changes in the underlying network in a more fine-grained manner than the previous scheduling protocols.

Although the design of these schedulers highlights some important lessons about adaptive scheduling protocols, the schedulers were fundamentally designed to address a limitation that doesn't quite affect web downloads to the same extent. Recall that MPTCP will send a packet if *both* the receive and send windows have available space (*i.e.* according to $min(cwnd, rwnd)$). Clients typically do not experience high load, and are rarely receive buffer-limited.[1] Conversely, send buffers at the server may be limited in size; if the server must deal with 100,000 incoming requests, it must proportion its memory accordingly. Nevertheless, it is not unreasonable to assume a server can afford up to 1-2MB of send buffer per connection. This fact combined with the small sizes of most web objects, on the order of tens of kilobytes on average, implies that send and receive buffer occupancy will remain low for web downloads, and will not elicit long delays due to HOL blocking. We further hypothesize that out-of-order packet arrivals should not be a concern of the scheduler, but should be dealt with in a manner very similar to QUIC, which circumvents any receive buffer problems by delivering out-of-order packets to the application layer immediately (Section 6.4.3).

[1] Linux and Mac OS use self-tuning TCP to grow the receive buffer. The maximum receive buffer size is 1MB in Mac OS High Sierra.

2.3.2.2 Streams

MPTCP deals with byte streams, and has little knowledge of web objects and their boundaries, which implies that its schedulers must be more generic. To investigate previous work which incorporates knowledge of web objects boundaries into the scheduling decision, we must turn to CMT-SCTP [72]. Web objects can be naturally expressed as SCTP streams, and multiple previous studies have examined how to schedule these streams on asymmetric paths. In SCTP, packet ordering is a requirement only *within* a particular stream. Therefore, HOL blocking in the receive buffer transforms into HOL blocking per stream if packets from the stream are transmitted over a slower path.

The simplest method for preventing packet reordering in streams transmitted over heterogeneous paths is to assign all packets from a particular stream to a single path. Dreibholz *et al.* investigate the benefits of such a scheme for bulk transfers and show that it indeed reduces HOL blocking [90].

Eklund *et al.* develop dynamic stream-aware scheduling (DS) for CMT-SCTP in [91], which takes current network conditions into account when making scheduling decisions about stream packets. The scheduler decides whether or not to send a message on the current path. It compares time to send a message on the current path with the time to transmit a full send buffer on the other path. If transmitting the full queue on another path is faster, DS chooses not to schedule any messages on the current path. If the current path can indeed send a message, the DS scheduler favours sending messages from the stream that transmitted on the current path last. In other words, the DS algorithm attempts to coalesce most of the data from one stream on a single path.

Despite similarities between the scheduler algorithm in DS and our own Stream Scheduler, DS attempts to bind streams to certain paths. Sending an entire stream on a particular path is rather fragile and will suffer delays if the path experiences any loss. Our work, on the other hand, is more flexible. It attempts to relax as many constraints as it can, and does not favour certain streams for a particular path. Striping packets from one stream on both paths is a better scheme since it is more reactive to loss, and can move stream packets away from congested paths. DS has been evaluated for short signalling traffic using CMT-SCTP, rather than web traffic with multiple origin servers using MPTCP. It does not take web objects, dependencies, nor priorities into account. If DS decides that a path is too slow and should not transmit a particular packet, the path remains idle, wasting possible capacity that could have been spent transmitting a lower priority object.

Finally, the Earliest Completion First (ECF) scheduler was developed for MPTCP operating over heterogeneous paths, and adopts a very similar scheduling algorithm as DS and our Stream Scheduler [92]. ECF is concurrent work with our research. It is evaluated mostly for video traffic, which is transmitted over one long lived connection. ECF does not consider slow-start in its estimation of the completion time. For web traffic, slow-start does in fact consume

a significant proportion of the download and cannot be discounted. ECF does not distinguish between streams, and suffers from the same limitations as DS with respect to leaving subflows idle. The evaluation of ECF for web pages is rather limited, with only one web page examined.

2.4 Dependency Graphs

One of the requirements for extracting web resource priorities, and for emulating web page downloads, is the ability to construct accurate dependency graphs for web pages. By computing which objects are crucial for reducing PLT from the critical path within a dependency graph, one can develop intuition about how to design schemes which will speed up the page load process. Several tools for generating web page dependencies are readily available, like HTTP Archive (HAR) [93], Chrome Developer Tools [94] and WebProphet [11], but they do not provide the means to produce comprehensive dependency graphs that include parsing and computation dependencies.

In our work, we use a heavily modified version of WProf [10] to generate detailed dependency graphs for our corpus of web pages. WProf instruments the Chrome browser and records the duration and dependencies of several activities, including web object loads, HTML parsing, JavaScript evaluation and CSS style resolving. It formalises several dependency rules that we extend and apply in Section 3.4 to build the dependency graphs for each web page.

A browser generally uses conservative algorithms to fetch and evaluate web objects because it has no knowledge of whether JavaScript will modify upstream HTML, or if CSS style properties will be accessed by later JavaScript. As a result, the browser will block parsing while it downloads and evaluates JavaScript. Similarly, it will commonly block JavaScript evaluation while it downloads and applies CSS rules. WProf models these lexical dependencies. In reality, these are too conservative and may not at all be necessary. JavaScript may not make any references to CSS style properties, or modify DOM elements, which means that the extra wait time for resolving these dependencies is wasteful.

As a result, Netravali *et al.* have developed Scout [95]. As opposed to only tracking lexical dependencies like WProf, Scout tracks precise data flows from within the JavaScript heap and locates more fine grained dependencies between objects. To achieve this goal, Scout re-writes the JavaScript and HTML code in a page. Scout wraps all the JavaScript objects with a proxy object, which allows it to instrument read/write, write/read and write/write dependencies between JavaScript variables and DOM nodes.

With fine-grained dependency information in hand, Netravali *et al.* also developed Polaris. Polaris is a JavaScript-based scheduler that uses the dependency graphs generated by Scout, and runs on the client's unmodified browser. In response to a client's HTTP request, the server replies with a scheduler stub containing four components: the scheduler, the generated Scout dependency graph, and the page's chunkified HTML and finally DNS prefetch hints. Polaris attempts to minimise PLT by prioritizing critical objects and requesting objects "out-of-order" with respect to lexical constraints, thereby improving performance. Using MPTCP as transport

is independent from employing a dependency based scheduler like Polaris. In fact, a multipath protocol that operates in the transport layer may improve the performance gains of Polaris even further.

Although our analysis might benefit from an even finer-grained dependency graph builder, Scout is not publicly available. We believe the dependency graphs we have generated are sufficient for our purposes and they have allowed us to draw interesting conclusions with regards to the object dependencies within a web page. More importantly, unlike Scout and Polaris, our goal isn't to create a scheduler that will improve web page loads. Instead, we want to model the *current* behaviour of existing browsers, which are conservative and obey the lexical dependencies.

2.5 Proxies

So far, we've discussed efforts to reduce web PLT by redesigning the web and transport protocols, or minimising critical path delay using dependency-aware schedulers. In parallel with these efforts, others have investigated the idea of reducing the long delays resulting from consecutive requests to the server by communicating instead with a reverse proxy, which requests web resources on behalf of the client. Located close to the web server, not only would this proxy reduce the RTTs experienced while requesting a web page, but if we assume it is sufficiently provisioned in terms of CPU and memory, it can construct the web page even faster and transmit it to the client.

In the work describing Mahimahi [96], which is a record-and-replay tool for web pages, the authors also present Cumulus. Cumulus combines a remote proxy which runs a headless browser on the cloud server with a local caching HTTP proxy. The remote proxy headlessly loads a particular web page and sends the compressed resources to the client's local caching proxy in a bulk response. The next time the client's browser requests a resource, it should hopefully find it in the cache already. By moving the resolution of resource dependencies closer to the server, Cumulus reduces the effective latency and speeds up page load times.

Bhattacherjee *et al.* go one step further, and suggest that to combat last mile delays that clients suffer even in the presence of CDNs, they should instead establish a point of presence close to the web servers in the form of content gathering networks (CGNs) [97]. These CGNs are reverse proxies that operate in a similar manner to Cumulus, downloading web resources and forwarding them to a local caching proxy on the client.

The examples we've described seem to suggest a split browser architecture, where a remote proxy does part of the work in downloading and building the web page. Several browsers for mobile devices adhere to this split architecture, because they operate on devices that can benefit from remotely constructing the web page on faster machines. Examples include Opera Mini [98] and Amazon Silk [99], which pre-process pages remotely, and Google Flywheel [100], a data compression proxy which operates between origin servers and the client.

The main caveat with proxy approaches lies in the potential divergence between web

pages generated by the client and the ones generated by the headless browser in the proxy. Because many web pages use JavaScript to dynamically construct and request URLs, different accesses to the same web page may trigger the download of different URLs. Additionally, only the client can know its internal state which will be used to implement certain JavaScript functionality. For example, JavaScript often checks the client's user agent (*i.e.*, browser type and version) and performs different actions or requests different resources as a result. Similarly, information like the client's window size, its local time, geolocation, cookies, HTML 5 storage and web database may influence the results of the page's JavaScript and hence the its final appearance.

Two different mechanisms have been proposed to deal with this divergence. The first involves sampling a large number of loads for the same web page, and merging the dependency trees for these samples offline to construct the web page's invariant dependency structure. Klotski [101] employs this mechanism to provide a good user experience on mobile devices, even if actively attempting to reduce PLT further cannot be accomplished. Klotski is composed of a back end measurement engine and a front-end proxy, both located close to the web server. The back end measures key invariant characteristics of a web page offline and the front end uses these characteristics and user preferences to prioritise content delivered to the client. Given a utility metric for each resource, a tolerance threshold for an acceptable user wait time, and a dependency representation, the front end is responsible for determining the subset of resources that will maximise the total utility given the tolerance.

The second mechanism is implemented in Shandian [102]. To speed up web pages, Shandian restructures the page load process so that it is split into two stages. It correctly prioritises resources that are needed for the beginning of the page, before the load event, and those needed after the page is loaded. As a result, a remote proxy is responsible for conveying the load-time state to the client, which includes only resources, JavaScript and CSS rules required for the DOM before the load event. The proxy therefore pre-loads the web page, communicates an initial compact representation of web page's DOM to the client, and loads secondary resources in the background. As for the client, it is responsible for executing the post-load state, *i.e.*, evaluating JavaScript and CSS rules, allowing interactivity with the user, and requesting further resources. To accurately replicate the page load remotely, the client sends its state, including its user agent, window size, *etc.* to the proxy. With a little added overhead, we believe this method produces more accurate web pages than Klotski.

Since our work investigates multipath transport as the means for reducing PLT, it is orthogonal to split browser architectures and reverse proxies. In fact, since they convert a mostly application-limited transfer into a bulk transfer, the described techniques may work better with MPTCP than a regular web download.

2.6 Conclusion

Despite the development of new web protocols like HTTP/2 and QUIC that target long page load times, and the design of different mechanisms for reducing web page loads, speeding up page loads by taking advantage of the multiple interfaces on mobile devices is an avenue that hasn't been explored in great detail. An in-depth analysis of how web pages behave when downloaded over MPTCP would help reveal how to apply multipath transport to the web. While we expect an improvement in PLT when the paths involved have equal bandwidth and RTT, others have shown that heterogeneous paths give less than ideal results. In our work, we aim to highlight these inefficiencies for web pages in particular, determine their causes, and tackle them systematically. We will emphasise the subtle interactions between web resource dependencies and the underlying web and transport protocols.

In the process of understanding web downloads for multiple paths, we will also examine the effects of unsharding web pages in more detail. While coalescing server shards appears to be good practice for increasing the gains of HTTP/2, a full scale analysis of unsharding with multiple different types of web sites and under different network conditions has yet to be performed.

Protocols like QUIC have demonstrated that combining application and transport-layer functionality can be beneficial. In a similar manner, we would also like to incorporate application-layer knowledge into the process of determining the best paths on which to send the next object in MPTCP.

Chapter 3

Page Completion Profiler

3.1 Introduction

Nuanced interactions between the application and transport layers determine end-to-end PLT:

- At the application layer, dependencies among resources in a web page's content—really, *web page structure*—determine when the browser sends requests to a web server. These dependencies also determine any *prioritisation* information the browser may provide the server when multiple requested resources are available to send. On the opposite end, the corresponding responses will experience varying *processing delays* at the server.

- At the boundary between the application and transport layers, the *number of servers* from which the browser retrieves a page's content and the *number of TCP connections* from the browser to each such server constrain the mapping of requests and responses to TCP connections. Similarly, the type of *web protocol* employed, whether HTTP/1.x, HTTP/2 or QUIC, informs the number of connections to use, and how quickly a web page's resources can be delivered in light of head-of-line blocking.

- At the transport layer, *end-to-end RTT* and *bottleneck bandwidth* determine the evolution of TCP's congestion window over time on each of these TCP connections, and thus in turn determine TCP transfer times for these requests and responses. Additionally, transmitting web packets over *multiple paths* as opposed to a single path may reduce PLT even further.

We would like to explore how the above italicised factors affect the PLT of popular web pages. Their impact is not immediately obvious, but rather requires some measure of experimentation. In addition to understanding the intricacies of status quo web protocols and their underlying transport, we also aim to evaluate several new mechanisms with respect to multipath web browsing. One way to achieve both these goals would be to experiment with a real browser modified to implement the protocols in question. Unfortunately, performing this evaluation is infeasible. Not only would we need to modify the client's browser and kernel, but we must also modify all the content servers contacted by the browser in order to understand any

new mechanisms we implement in the application and transport layer. An alternative would be to snapshot a particular web page, and create several local servers with our chosen protocols and configuration that respond to HTTP requests with the correct resources. But this design still lacks the level of determinism we require for our analysis. In particular, we cannot obtain the same behaviour for each run of our experiments if we clone dynamic content that, due to JavaScript, produces a different result every time it is executed. Besides, by using this configuration we would already need to emulate the server and the network functionality. Instead, we have built a hybrid system, which simulates real web traces within an emulation framework that will allow us to answer "what-if" questions with respect to the factors above, and to quickly prototype and evaluate several existing and potential web protocols and transport.

It is completely inaccurate to model web pages as a series of consecutive resource requests. The page load process is much more elaborate: browsers mix the fetching of web resources with parsing, and evaluating JavaScript and CSS. These activities are interrelated and create inherent dependencies among themselves. On a similar note, an accurate web dependency representation is necessary to communicate useful cross-layer information to the transport protocol. If the transport protocol wants to understand which web resources are delay-sensitive, it must understand the relationships between the various page load activities. To perform an analysis of web page load times, we have implemented *PCP: a Page Completion Profiler* that traces and emulates web page downloads. PCP allows us to better understand the factors responsible for PLT and the possible areas for latency improvement.

3.2 Anatomy of a Page Load

To build a page completion profiler that accurately captures the time it takes to download a web page, we must first understand where most of this time is spent. We must also take into account how the structure of the web page influences its download time. As a result, in this section we break down the underlying activities browsers perform to download and render a web page. In essence, a web page is a collection of frames which display content. A frame contains resources like images, text and media, whose structure and appearance are concisely specified by the frame's HTML file. The browser downloads these resources, and transforms the HTML description into the document object model, or DOM tree.

As a language-independent intermediate representation, the DOM provides a standardised API for interacting with HTML objects. Each HTML tag corresponds to an Element node in the DOM tree, and each Element owns a set of attribute nodes. The Document object acts as the root node for an HTML document, and owns all the other nodes (element, attribute, text and comments). In turn, the Document's parent is the DOM Window object, which represents an open window in the browser. If a page contains several iframes (inline frames) in addition to the main frame, then the browser creates a Window object for each frame (and by extension a Document object). The rest of this chapter capitalises DOM objects, *e.g.* Document, to distinguish them from their counterparts.

Web page developers use Cascading Style Sheets (CSS) to apply style rules to DOM nodes (*e.g.* font size), and JavaScript to manipulate the DOM dynamically. With the constructed DOM as an intermediate representation, the browser renders the page for the client to see.

```
<html>                                                                      0
  <head>                                                                    1
    <script>                                                                2
      function downloadImage()                                              3
      {                                                                     4
        //Create a new Image                                                5
        var anImage = new Image();                                          6
        anImage.setAttribute("src", "rainbow.jpg");                         7
                                                                            8
        //Append it to the div tag                                          9
        var element = document.getElementById("firstDiv");                  10
        element.appendChild(anImage);                                       11
      }                                                                     12
      function fireTimer(){                                                 13
        setTimeout(downloadImage, 10);                                      14
      }                                                                     15
                                                                            16
    function loadFrame() {                                                  17
      fireTimer();                                                          18
    }                                                                       19
    </script>                                                               20
  </head>                                                                   21
  <body onload=loadFrame()>                                                 22
    <link rel="stylesheet" type="text/css" href="theme.css"></link>         23
    <div id="firstDiv">                                                     24
      <p2> Some Paragraph </p2>                                             25
    </div>                                                                  26
    <script src="image_create_sea_append.js"></script>                      27
    <div id="secondDiv">                                                    28
     <p> Some more text </p>                                                29
    </div>                                                                  30
  </body>                                                                   31
</html>                                                                     32
```

Figure 3.1: An example HTML file.

Suppose a client navigates to the web site described by the HTML code in Figure 3.1. After the browser sends the initial request and downloads the main HTML file, the browser's following components work together to display the web site:

- **HTML Parser**: reads and tokenises the HTML tags in the main page as well as any iframes enclosed within the page. It also tokenises document fragments, generated from JavaScript calls to `Document.write()`. The parser does not wait for the entire page to download, but rather begins operation when the first chunk of the web page arrives over the network. As it runs through the HTML tags, the parser builds the DOM tree by converting each tag into an DOM Element and setting its attributes correctly. If the parser encounters an HTML tag that references an embedded resource, such as the `<link>` tag on Line *23* in this example, it triggers its download via the Resource Loader. Parsing is single threaded: the parser pauses when it encounters inline or synchronous JavaScripts that the browser must download and execute, for example Lines *2* and *27*. As an optimi-

sation, browsers usually also employ a light weight parser, or preloader, which runs on a separate thread and facilitates downloading resources while the main parser is blocked (discussed in more detail in Section 3.7.5). Parsing runs for fixed epochs in WebKit [103], the browser engine used by Safari and Chrome. As such, the parser will yield execution to JavaScript handlers for events or timers when they fire. When the browser downloads an iframe, parsing its HTML does not preempt the parsing of the main frame.

- **Resource Loader**: is responsible for requesting and downloading the various resources referenced in the web page if they are not already found in the browser's cache. The resources can have different MIME types, like images, JavaScript or CSS files, iframes and other media. The Resource Loader requests each resource using a version of HTTP as the underlying protocol, although new Chrome browsers use QUIC. The browser uses DNS to resolve the IP addresses of resource domain names if they have not been cached already. Additionally, the browser will also transmit and handle the necessary TLS handshake for new connections that require HTTPS. Once DNS is resolved, and both client and server have exchanged TLS keys if necessary, the Resource Loader can then send the request to its target server and begin the download.

- **CSS Style Resolver**: CSS is referenced via the `<link>` tag (*e.g.* Line *23*) to download an external file, or inline using the `<style>` tag. After downloading a CSS file, the browser parses it, and converts the CSS into style rules. These rules are often used to build a tree-based CSS object model, or CSSOM, as is the case with Firefox. WebKit on the other hand parses each CSS rule and inserts it into a StyleSheet object, which encapsulates a set of CSS rules. Once parsing a CSS file completes, the browser invokes the style resolver, which iterates through all the style sheet rules for each node and collates them, starting with the most general and then recursively applying more specific rules, in order to generate the derived styles for all DOM nodes to be displayed. Finally, the browser recalculates the style of these DOM nodes, iterating over them, and applying to collated style rules. As CSS files are downloaded, the style rules are applied incrementally.

  Since JavaScript functions may check the style properties of DOM nodes during their execution, the functions will get this information wrong if the CSS files have not been downloaded and evaluated. As a result, browsers commonly block JavaScript execution while there is a pending CSS file which hasn't completed loading. Firefox blocks all scripts as it waits for pending CSS to load, whereas WebKit only blocks scripts when they try access style properties that may be influenced by unloaded CSS.

- **JavaScript Engine**: executes any inline or external scripts. After the parser encounters the `<script>` tag on Line *27* in our example, and requests the referenced *.js* file using the Resource Loader, the engine executes the script after the resource download completes. Modern JavaScript engines, like Chrome's V8, use just-in-time compilation to produce

optimised code. In addition to evaluating inline or external JavaScript code, the engine also evaluates JavaScript event handlers and timer functions. The JavaScript engine is single threaded, but can be re-entrant. In other words, an event handler for example may interrupt the execution of an enclosing script. When the event completes, the JavaScript engine will resume the execution of the parent script.

- **Renderer**: uses the DOM to display the web page progressively in the browser. As the browser constructs the DOM tree, it also constructs a corresponding render tree which contains the web page's visual elements in the order they will be displayed. Building the render tree requires computing the style of each object, using the CSS style rules computed for each DOM element.

  After the browser builds the render tree, it computes the layout of all the render objects in the tree, *i.e.* it determines their position and size in the browser's displayed frame. With the render tree and object layout in hand, the browser finally paints the web page on screen.

  The browser can begin painting the web page without waiting for all the resources to load for the page. The critical rendering path will only include the HTML markup, CSS and JavaScript files, foregoing any images or iframes that may be downloaded later. WebKit also uses a flag to indicate whether all top level style sheets have been loaded, and constructs the render tree with placeholders for missing elements, allowing "first paint" to happen even earlier. The placeholders are recalculated as the style sheets are loaded, and paint is refreshed.

  Although the browser performs rendering activities on the main thread, which also parses HTML and evaluates JavaScript, Wang *et al.* have shown in their WProf work that for 80% of the 200 web sites examined, page rendering accounts for less than 5% of the total time on the critical path [10]. Even on mobile devices, which spend a larger proportion of their time performing computation activities like parsing and script execution, Nejati *et al.* [104] have shown that rendering activities are the least dominant on the critical path. As such, our following examination of page loads will *not* model browser rendering activities.

The activities performed by the parser, CSS style resolver, JavaScript engine and renderer are computation processes, whereas the Resource Loader performs the only network process. In theory, a browser can possibly execute the download, parsing and computation processes in parallel, reducing a page's load time to that of its slowest activity. Reality, on the other hand, necessitates the execution of some activities before others, adding dependencies between them. Some dependencies are flow based: the parser, for example, cannot process an HTML tag until the chunk of data containing this tag arrives over the network. Others are serialisation artefacts, for example the parser and JavaScript or CSS cannot run at the same time, because they both

modify a shared data structure, the DOM tree. To model web page loads correctly, we aim to quantify these dependencies, and generate faithful dependency graphs for the various browser activities.

3.3 Design Overview

Our goal is to measure the page load times of web sites of varied structures under a wide range of network conditions, using different web protocols, network transport protocols, server architectures, and different browser optimisations. Additionally, our eventual goal requires modifying the browser and underlying transport in order to incorporate new methods for reducing PLT for multiple paths.

At first glance, the simplest mechanism for measuring web PLT would be to apply any modifications solely to the client, with changes to the browser and to the network stack. With this method, we could measure page completion times in the wild, with real RTTs to each origin server, real server processing delays, and real background traffic. We could use the Linux traffic control facility `tc` to control the rate, and add extra delays on top of the empirical RTTs to vary the effect of the network and observe the results. Unfortunately, we do not have control of the servers, and are therefore limited by their choice of architecture and protocol support.

In order to experiment with different server architectures, and to expand their design to incorporate new web and transport protocol features, we must emulate the servers locally. We could either implement our own record and replay scheme to emulate the servers, using `dummynet` to emulate different network bandwidths and RTTs [105, 106], or instead augment available tools like Mahimahi [96] for our purposes. We would need to record each resource requested by our target web site, emulate its download from multiple origin servers locally, and make sure to apply the correct server processing delays. This experimentation framework would be advantageous because it still uses a real browser, the operating system's network stack, and allows us to use common server implementations, like Apache with mod_http2 [107], modified for our purposes of course.

Nevertheless, the framework above still lacks a crucial element for performing the types of experiments we have in mind: reproducibility. In particular, we would like to repeat experiments that encounter the same packet drops, in order to profile how minor changes in these drops influence web page completion. Reproducibility is also a requirement if we want to examine the effects of incremental changes to the application and transport protocols. Ideally, we would like to implement a new protocol feature, detect any possible reduction in performance, fix the problem and rerun the same experiment in an iterative process.

We would also like to use a more consistent browser. As we will see in Section 3.7.4, the number of resources downloaded before the `load` event fires in a browser (a typical indication that a page has completed) may vary with network conditions. JavaScript events and timers may fire at different instances, triggering the download of a different set of objects each time. This lack of determinism makes it very difficult to perform an apples-to-apples comparison

between different protocols. Instead, we would like to measure the same amount of work done for each experiment we conduct.

For reproducibility, our best recourse is to use simulation. Our approach is a hybrid one. First, we measure and profile a wide range of popular web sites taken from the Alexa top 500. From this we construct a web dependency graph for each web page; this graph contains all the dependencies between the large number of events that make up the downloading and processing of the page. Nodes in the graph include resource download, HTML parsing, JavaScript execution, timers, and other events. Edges exist in the graph if one event must only occur after another event has finished.

Based on this measured graph, we emulate the download of each web page while varying parameters of interest. We can vary the network bandwidth, for example, simulate the resulting TCP downloads, and predict the effect this will have on page load times. For correct emulation, we need to accurately model how the different activities within a modern web browser depend on each other. For example, inline JavaScript will block further parsing of HTML after the script's location in the HTML file, but the browser's preloader may still preparse this HTML as it arrives and enqueue requests for some of the URLs it finds. We implemented a trace-driven browser simulator that takes this dependency graph and emulates the browser, initiating HTTP requests when all their dependencies are satisfied. All the timings for parsing, JavaScript evaluation, and so forth are taken from measurements of the real browser when profiling the original web page. To model the network aspect, we use the *ns-3* packet-level simulator [35], as we need to fully understand the interactions between HTTP, TCP, and network conditions. Combining these components together, we built the Page Completion Profiler, which consists of:

- *Profiler*: an instrumented copy of the Chrome web browser that logs all events during page download and processing. This is complemented with a *tcpdump* capture of all packets, allowing us to infer additional parameters unseen by the browser, such as server processing latency.

- *Graph Builder* takes the event log from the profiler, uses a model of how dependencies work in a web browser, and builds a directed acyclic graph that represents all the inter-event and inter-object dependencies within that web page.

- *Graph Viewer*: a visualiser that can present the dependency graph in PDF form, allowing us to understand dependency graphs, so as to perform root-cause analysis. An example is shown in Figure 3.6.

- *Emulator*: a trace-driven web browser emulator that takes a dependency graph, and simulates downloading the relevant web page under specific conditions and with specific protocol features enabled or disabled.

3.4 Profiler

Our profiler builds upon WProf [10], which instruments the Chrome browser and inserts hooks into WebKit to log information about an individual web page as it loads. WProf logs a record for each resource request, parsed HTML token, and evaluated computation during the page load process. Each record contains the activity's unique identifier, its immediate parent (*i.e.* the HTML tag, DOM element or computation that triggered the activity), and some timing information to mark the start and end of each activity.

Although it logs a great deal of data, WProf was not designed to provide sufficient information to drive emulation. In contrast, we need to identify *all* the inter-resource dependencies so that the parents of every resource and computation are logged, a complicated feat owing to the copious web features and corner cases the Chrome browser must support. In addition to the original activities logged by WProf, a sample of some of the new activities the Profiler tracks are:

- Inline JavaScript executions. Although scripts inline with a web page's HTML do not require requesting a JavaScript file over the network, they pause the parsing process nonetheless, thereby increasing PLT.
- Programmatic DOM elements created from JavaScript. These are not created by parsing an HTML tag, but rather, for example, by a JavaScript call to `document.createElement("script")` or `var i = new Image()` (Line 5 of Figure 3.1). By accurately recording DOM elements that have been appended to the document, we can reliably determine which script evaluation has triggered the download of a resource or the evaluation of another script.
- Timers registered from JavaScript functions, when each timer fires, and the duration of the timeout handler function.
- Events and when their handlers are executed. We log the event names, in addition to their targets. For example a `load` event called from an image tag, indicates that the load event's parent is the downloaded image resource. The target may also be the JavaScript document or a Window object, as with the page load event.
- The difference between parsing activities generated by the parser itself, or by evaluating `Document.write()` from a JavaScript function.
- Frame identifiers for each record. Computations, resources and parse tags may either belong to the page's main frame, or to an iframe within the page, in which case the activity depends on the download of the iframe itself.
- Frame hierarchy and source changes. We note a frame's parents and children in our traces, so that we can model frame dependencies. We also log when an iframe's URL is programmatically changed during the download of a web site.

- Resources accessed from the cache. Before our modifications, WProf only recorded resources which were explicitly downloaded from the network. Although we clear the cache between page downloads, some resources are referenced multiple times within the same web page. Therefore, later references to the same resource will go straight to the cache, and since the resource has not been explicitly requested, we obtain an incomplete dependency graph. Because we would like to model caching in the emulator, we instead log all the references to the resource, even if they resulted in a cache hit.

To record a trace for a particular web site, we use our instrumented Chrome browser to navigate to that site, and wait for one minute to allow for the page to complete. We clear the browser cache between each consecutive page we profile, because we would like to capture all the resources that the web page can potentially download. After capturing all the resources and their download times, it would be simple to model a warm cache download in the emulator if we so choose. Using our modified version of Chrome, we obtained logs for 300 of the Alexa top 500 web sites [36], sampled in September 2016. The instrumented Chrome browser used HTTP/1.1 consistently for downloading all of the sites. We use traces for these 300 web sites throughout the remainder of this study.

3.4.1 Server Processing Times

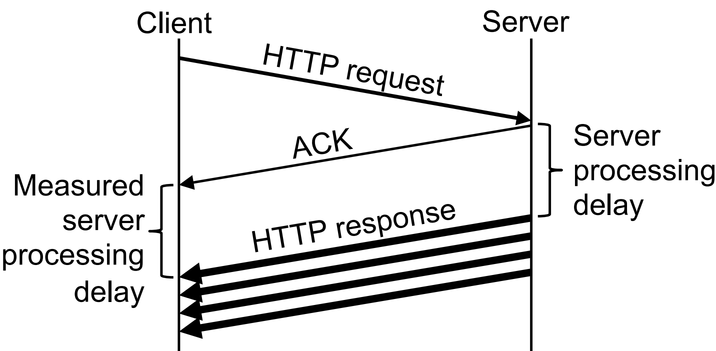


Figure 3.2: Measurement of server processing delay from *tcpdump* traces at the client.

To faithfully reproduce a page download, we also need to emulate any delays caused by the web server. We observed that these can sometimes dominate page download times when network latency is low and bandwidth is high. The browser sits above TCP, so does not have enough information to distinguish transport latency from server delays. To infer server processing delays we capture a full packet trace during profiling using *tcpdump*. For this we use HTTP/1.1, as pipelining with HTTP/2 would mask the effect we wish to measure. Figure 3.2 illustrates how we measure the server processing delay. As the HTTP request arrives with the TCP PUSH flag set, it is immediately ACKed by the server. After a short delay, the server sends the first data packet of the response. This difference can be measured at the client and the server processing delay inferred.

There are several caveats to using this method for estimating server processing delays. For

one, queuing delay could add noise to this measurement. However, we observe that the latencies seen by incoming traffic to our university network are orders of magnitude lower than the server processing delays we measure. Additionally, because we infer server processing delays from HTTP/1.1 exchanges, these delays may not accurately reflect HTTP/2 server processing delays. With HTTP/2, the server may apply different code for prioritisation and header compression for example, which takes a different amount of time. Despite this limitation, we believe that HTTP/1.1 server processing times are a lower bound for those experienced by HTTP/2, and function as a reasonable approximation.

We use the ***dpkt*** Python package [108] to process the *.pcap* files generated using *tcpdump* in order to pair observed HTTP requests with their responses, and compute their corresponding server processing times using the mechanism outlined above. We discard estimates for resources with non-successful HTTP status codes, *e.g.* errors and redirects. One the other hand, we give special care to HTTP responses with `HTTP 304 Not Modified` status. A response of this kind carries no content, and indicates that the browser should use the corresponding resource from its cache since the server has not modified it since the last access. Although we record the measured server processing times for these particular resources, we mark them with a special "cached" flag.

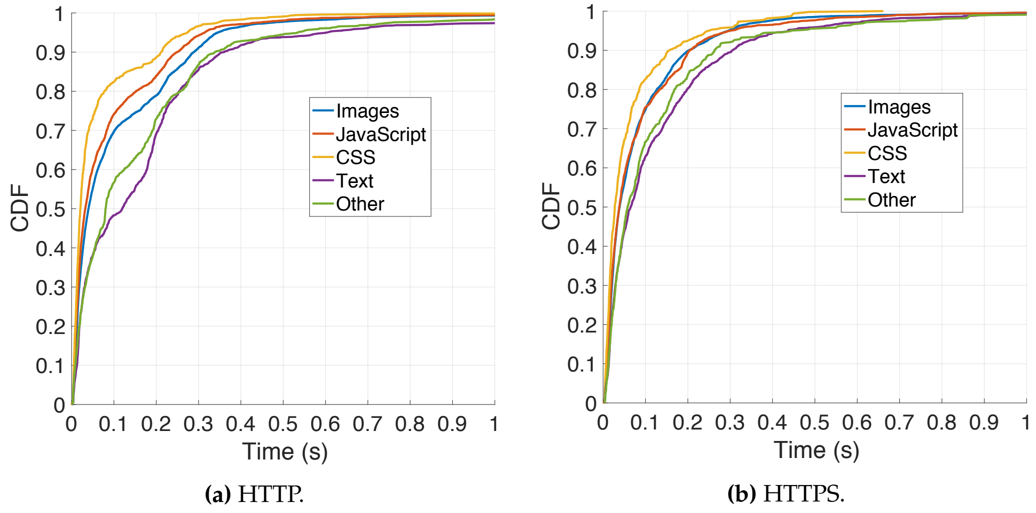
Deducing server delays for HTTP connections is relatively straightforward. HTTPS connections, on the other hand, encrypt requests and responses (including the HTTP headers) using TLS. As such, *tcpdump* can only record an exchange of opaque packets, and without visible HTTP headers we cannot discern the resource URL of each request and response. Although we can compute the server processing delay for an exchange, we can't tell which resource in particular experienced this delay.

Are we out of luck for HTTPS resources? Not necessarily. Since the Profiler records information about all the downloaded resources, we can use the Profiler logs in conjunction with the packet traces to our advantage. From the logs, we generate a list of resources per domain, sorted by their receive times, *i.e.* the time the very first packet of the response arrives at the browser. Therefore, for each domain which uses HTTPS connections, we match the series of recorded server delays with the corresponding resources in order. But since the only visible information of use in an HTTPS exchange is the TCP connection's four-tuple, we must also maintain a mapping between IP addresses and domain names, which we construct from parsing DNS queries and answers in the packet trace. We correctly handle multi-homed domains, and resolve CNAMES to produce an accurate mapping.

Given that TLS exchanges are opaque, the above mechanism may attribute the wrong server delay to a particular resource. It produces an approximation of a resource's server delay at best, especially since we cannot read the HTTP status code of the response. Nevertheless, we observed that the server delays for resources downloaded from the same domain correlate highly with each other, and will most likely be centred closely around the mean.

For the 300 top Alexa web sites we profiled, we downloaded approximately 35K resources, and applied the mechanisms above to infer their server processing times. Unfortunately, we still could not determine server times for around 4600 resources from the recorded *pcap* traces. The resources with missing server delays fall into two categories. The first consists of resources downloaded from domains for which we have already computed a set of server delays for other resources. The missing server delays represent cases where we still could not disambiguate TLS responses correctly, or cases where we have malformed HTTP requests or responses. We assign each resource of this kind a server delay drawn from the list of server delays associated with the resource's domain, with the same MIME type. If the resource has zero size, it most likely received an `HTTP 204 No Content` response, so we clamp its server delay time at 10ms.

The second category consists of resources downloaded using TLS from domains whose names were already in the DNS cache. Since we observed no DNS queries for these domains, we could not disambiguate the IPs of encrypted requests and responses for their resources. With a sufficient sample of empirical server processing times, we can estimate the cumulative distribution function of the server processing times and use it to probabilistically generate times for these missing resources (we found 313 in our sample). Because a server will consume a different amount of time when it handles resources of different types, we need to generate separate CDFs for resources with different MIME types and HTTP methods (*i.e.* GET vs POST).



(a) HTTP. **(b)** HTTPS.

Figure 3.3: Server processing delays for content served via the specified protocol.

Figures 3.3(a) and 3.3(b) show CDFs of the measured server processing delays captured in September 2016 for HTTP and HTTPS content respectively. Servers spend more time generating HTML than most other MIME types, likely because some of this content is dynamically generated. Conversely, static objects such as CSS files, which are likely to reside in the server's cache (RAM), require small server processing times (less than 250 ms). There is also a distinct excess around 200ms with HTTP. In results from December 2015 (shown for JavaScript in Figure 3.4), this excess was rather large, and predominantly affected JavaScript and images. This

was almost certainly caused by a 200ms bidding timeout in advertisement networks. The effect is somewhat smaller in more recent traces, indicating a likely change in the operation of one or more ad networks, and not visible in content served over HTTPS.

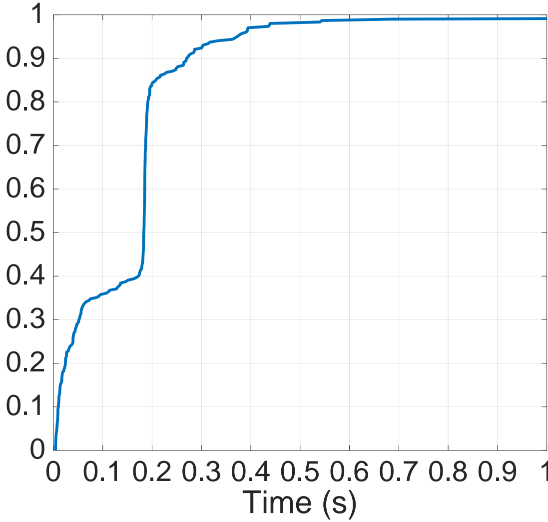


Figure 3.4: JavaScript server processing delays from 2015.

As we can see, a server typically returns an HTTP response within one second, therefore we can safely remove any outliers with greater values.[1] Note that we also exclude the resources marked as "cached" from the samples used to generate the CDFs, since they correspond to responses which do not contain data. Nevertheless, the empirical response time for a cached resource is still used for the corresponding resource in the emulator, because it represents a lower bound for how fast the server can turn around a response for this particular resource.

To apply server processing delays, the emulator operates as follows: when a server receives an HTTP request for a particular resource, it checks whether an empirical measure for this resource's processing time exists. If so, it uses the measure. Otherwise it generates one from the measured CDFs based on the resource's MIME type, and applies it according to the chosen server design.

3.5 Graph Builder

The Graph Builder parses a raw Profiler trace and generates a web dependency graph (in JSON format) for the corresponding web site. To do this, it embeds a set of rules that capture the semantics of how the browser learns about resources to download, how the parsing thread processes embedded resources such as JavaScript and CSS, and how timers and event handlers work.

[1]Our results found only 233 resources with server load times greater than 1s, from a total of about 35K resources.

3.5.1 Dependency Activities

To build a web page's dependency graph, we must first define the web activities that form its nodes, and their relationships with one another. A web page *activity* denotes a particular action a browser takes to download and display a web page. We summarise the types of activities we will use and their dependency relationships in Table 3.1, which provides a brief overview before we examine these activities in more detail in the following sections. We use $a \to b$ to indicate that activity b depends on the completion of activity a. Similarly, $a \wedge b \to c$ indicates that c depends on the completion of both a and b.

Rule	Parent Dependency	Activity
1	ParseChunk	→ Resource
2	Computation	→ Resource
3	$\text{ParseChunk}_{n-1} \wedge \text{DownloadChunk}_n$	→ ParseChunk_n
4	$\text{ScriptEval} \wedge \text{ParseChunk}_{n-1} \wedge \text{DownloadChunk}_n$	→ ParseChunk_n
5	ParseChunk	→ DownloadChunk_0
6	ScriptEval	→ DownloadChunk_0
7	Event	→ DownloadChunk_0
8	Timer	→ DownloadChunk_0
9	$\text{DownloadChunk}_{n-1}$	→ DownloadChunk_n
10	Resource_{script}	→ ScriptEval
11	$\text{CSSEval} \wedge \text{Resource}_{script}$	→ ScriptEval
12	ParseChunk	→ ScriptEval
13	CSSEval ∧ ParseChunk	→ ScriptEval
14	ScriptEval	→ ScriptEval
15	Event	→ ScriptEval
16	Timer	→ ScriptEval
17	$\text{Resource}_{script} \wedge \text{ParseChunk}_{last}$	→ $\text{ScriptEval}_{defer}$
18	Resource_{css}	→ CSSEval
19	ScriptEval	→ Timer
20	Event	→ Timer
21	Timer	→ Timer
22	$\text{DownloadChunk}_{(frame,\, last)} \wedge$ ScriptEval Parent	→ $\text{ScriptEval}_{frame}$
23	$\text{DownloadChunk}_{(frame,\, last)} \wedge$ Event Parent	→ Event_{frame}
24	$\text{DownloadChunk}_{(frame,\, last)} \wedge$ Timer Parent	→ Timer_{frame}
25	*Event specific*	→ Event

Table 3.1: Summary of dependency graph activities and their relationships. Event activity parents are described in more detail in Appendix A.

Broadly speaking, a web page dependency graph consists of four main types of activity nodes:

- A **resource** is a network download activity, where the browser sends a request for a particular MIME object to a remote server, and waits for the complete response. A resource download commonly *depends* on parsing a specific HTML tag to obtain the resource's URL, but a resource may also depend on executing a computation (CSS, JavaScript, timer or event evaluation), which programmatically triggers the download. Images, CSS and JavaScript downloads are examples of resources, but HTML pages are not, as HTML can be incrementally parsed as it arrives, triggering new dependencies, so it needs special

handling. In our example from Figure 3.1, the JavaScript in Line 7 creates an image element, which in turn causes the browser to request the image resource for download. Evaluating CSS style rules may trigger font or image downloads as well. Consequently, a resource may have either a parse chunk or computation as a parent, as captured by Rules 1 and 2 in Table 3.1.

- A **parse chunk** is a unit of computation with duration equal to the parse time until the next resource download or inline script declaration. A browser's parser does not usually consume an HTML frame in one continuous pass, but rather pauses to evaluate inline scripts, or yields to allow other scripts to execute. Therefore, we must split the parsing of a contiguous HTML file into multiple logical chunks, whose boundaries occur at HTML tags that trigger downloads or inline script evaluations. For example, a simple HTML page with two embedded images would be split into three parse chunks, separated by the HTML tags for the images. After the first parse chunk has completed, the download of the first image resource can begin, so this image resource depends only on the first parse chunk, not on downloading or parsing the second and third chunks of HTML. Figure 3.5 shows our original example with the parse chunks marked on the left hand side. In order to execute a parse chunk, the browser must download the corresponding HTML download chunk first, and execute the previous parse chunk (Rule 3). Any synchronous JavaScript evaluated as the result of the prior parse chunk will also block parsing, and is therefore a parent (Rule 4).

- A **download chunk** represents the download of the chunk of an HTML file corresponding to a particular parse chunk. A download chunk depends on the completion of the previous download chunk (Rule 9). The first download chunk of the main HTML frame forms the root of the web page's dependency graph. As for iframes, their first chunk is either downloaded as a result of parsing an HTML `<iframe>` tag, or evaluating a script that creates the iframe programmatically (Rules 5-8).

- A **computation** is a JavaScript or CSS evaluation with the ability to modify the DOM elements dynamically. Like parse chunks, we will subdivide computations along resource download boundaries to allow for easier simulation. Computations also include JavaScript timers and events, which fire when a certain time elapses or certain DOM-related events occur. When evaluated, their handlers may trigger further activities, such as resource downloads. Some JavaScripts are re-entrant: they can be interrupted by the execution of an event or another script before resuming, *e.g.* an event handler for the `DOMNodeInserted` event. To model these re-entrant scripts, we subdivide the parent computation into chunks around the child's boundaries for simulation purposes. A script, event or timer invoked within a particular frame may manipulate DOM elements from a different frame. These computations will, in consequence, depend on the complete

download of the frame in question up till its last chunk, in addition to their regular parents (Rules 22-24).

3.5.1.1 Computation Activities

We further classify computation activities into four different types:

- **JavaScript evaluation**: or script eval for short. JavaScript can either be declared inline within an HTML file, or can require the download of an external *.js* file. Inline JavaScript executes synchronously, blocking the next parse chunk since the HTML parser must wait for the JavaScript engine to complete (Rule 12). External JavaScript, on the other hand, cannot execute until its resource completes downloading (Rule 10). Developers may set one of three attributes to specify when an external script should execute: *sync, async* or *defer*. The default *sync* attribute indicates that the JavaScript file should complete downloading and executing before the parser can continue onto the next tag. JavaScript marked with the *async* attribute, allows the parser to continue processing tokens while the resource downloads. Once the download completes, the browser can evaluate the script asynchronously when it sees fit. Finally, a developer may choose to use the *defer* attribute to delay the evaluation of the JavaScript until the main frame has finished parsing (Rule 17).

  Because both the CSS and JavaScript evaluations modify the DOM, most browsers will prevent any future JavaScript from executing until previous CSS files have completed downloading and have been executed. Since we do not have enough information from the Profiler to selectively block JavaScripts that explicitly attempt to access style attributes, we conform to FireFox's model, where a CSS evaluation is the parent of the next synchronous JavaScript evaluation (Rules 11 and 13). Script elements may be created by other JavaScript functions. Therefore timers, events and other scripts can be parents of a script eval (Rules 14-16).

- **CSS evaluation**: represents parsing a *.css* file, resolving the enclosed style rules, and recalculating the style of all the nodes in the DOM tree. CSS evaluation depends on downloading the CSS resource first (Rule 18). CSS evaluation may block JavaScripts from executing (as described above). Because the browser performs CSS style recalculations on the main thread, CSS will block any other parsing or JavaScript evaluation. But these scheduling dependencies are dynamic, and cannot be expressed explicitly in a graph. Instead, we model them in the emulator, as described in Section 3.7.2.

- **Timers**: are declared within JavaScript. They schedule a function for execution at a specified interval in the future. When a timer fires, its handler may trigger further activities, like resource downloads. A timer activity depends on evaluating the JavaScript that registered it, i.e. either a script eval, event or another timer activity (Rules 19-21). Timers

may be one-off, declared in JavaScript using `setTimeout()`, or recurring declared using `setInterval()`, *i.e.* firing every specified number of milliseconds. Interval timers may form long timer chains in the dependency graphs, especially if they are used to animate part of a page, or to periodically check the status of a variable in order to trigger further actions. Because these timer chains usually have minimal impact on the critical path completion of the page, we only include timer chains which eventually download a resource in our dependency graphs.

- **Events**: are activities which represent the evaluation of JavaScript event handler functions. JavaScript allows developers to register event handlers that fire whenever the event occurs in a web page. Events can be generated as a result of the user interacting with the UI, for example clicking a button on the page, or scrolling. Others can pertain to a particular DOM element or even the DOM Window itself where the web page is rendered. For example, an image `load` event fires when the file referenced by the image element completes downloading. An event therefore has a particular name and a DOM target to which the event is dispatched.

  When handling events, we must take nesting of DOM elements into consideration. Suppose an event fires with a particular DOM element as the target, which is in turn nested within two other DOM elements. Since each of the elements may have implemented its own handler for the event, the browser can execute the handlers in two possible orders. If the browser executes the handlers in a top down manner, we say that the event is *captured*. The evaluation starts with the outermost DOM element, and the browser executes the nested handlers in turn until it reaches the target element. Conversely, if an event *bubbles*, then the browser calls the event handler of the target first, and then the event handler of its parent and so forth in nesting order. Whether an event exhibits bubbling or capture behaviour is particular to the type of event itself. As the parent node of all enclosed DOM elements, bubbled or captured events eventually trigger the Document object's event handlers if they are specified, and in turn the DOM Window's handlers. To construct the dependency graph, PCP correctly assigns the parent activities of captured or bubbled events.

  JavaScript exposes numerous event types with different semantics. Consequently, we can deal with event activities only on a case by case basis, where the type of event and its target dictate which parents it depends on. Since attempting to account for all possible JavaScript events is infeasible, we have decided to narrow our focus to the most common events encountered in our analysis of popular web sites. We immediately discount any events which may fire as a result of user interaction with the page, especially since we are only interested in the page load process. Tables A.1, A.2, A.4 and A.3 in Appendix A summarise the event activities our model will focus on, and their dependency information. The events target the DOM Document, Element, Window, XMLHTTPRequest and

MessagePort objects respectively.

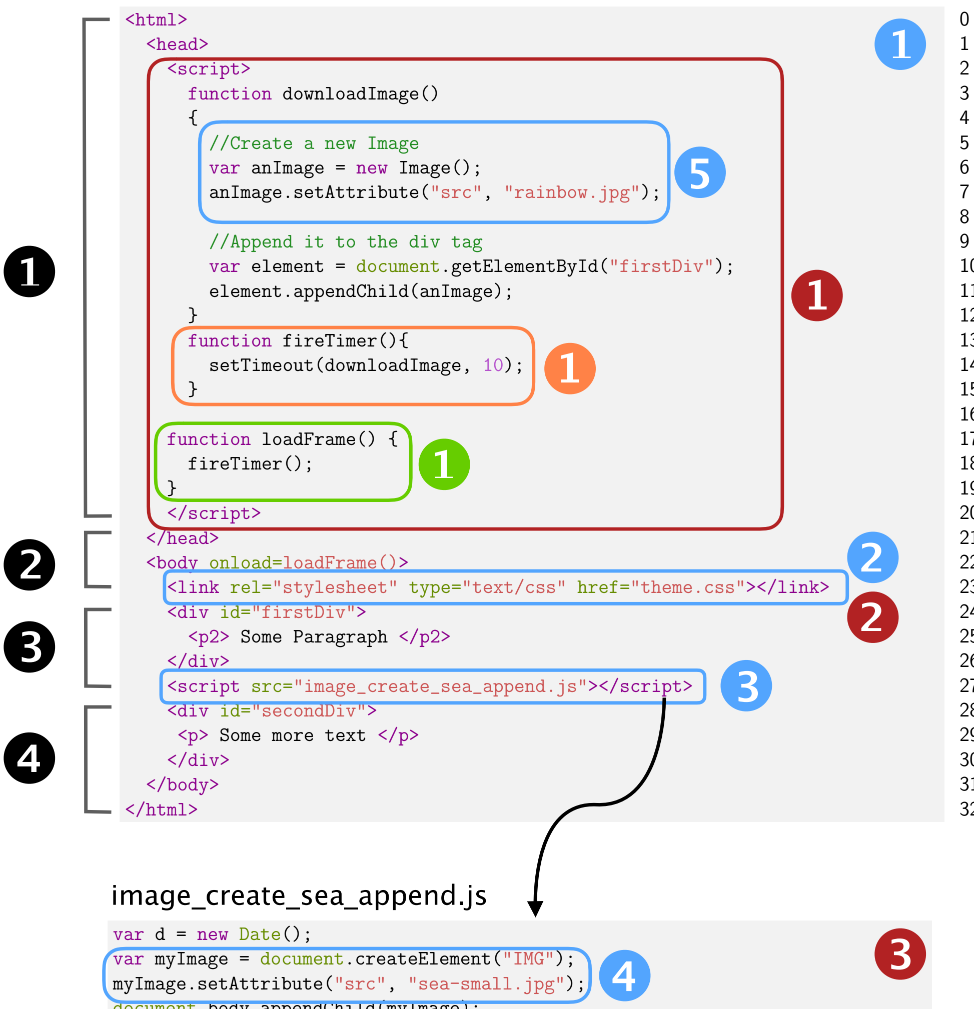


Figure 3.5: Our example HTML file, reproduced here with the relevant annotations. We mark the parse chunks in black in the left hand margin. We further mark the resources in blue, computations in red, events in green, and timers in orange. The `<script>` tag on Line 27 triggers the download of the *.js* file shown at the bottom. Refer to Figure 3.6 for the corresponding dependency graph.

With these dependency rules in mind, recall our example web page in Figure 3.1. We reproduce it here in Figure 3.5, and annotate the dependency graph activities we extracted after running the Profiler. We show in Figure 3.6 the dependency graph constructed after profiling the web page, and annotating the corresponding dependency graph activity nodes. Each box represents one of the activity node types we've described above. The graph in the figure enforces the dependency rules we have outlined. For example, the inline JavaScript (marked with a one inside a red circle) depends on the parse chunk containing its `<script>`, and blocks the next parse chunk till it completes. The *.js* and *.css* resources depend on their respective parse chunks, and they trigger the evaluation of their content (the red circles with two and three). The `DOMContentLoaded` event depends on the completion of parsing, and, in addition

to the resources downloaded by the page, is a parent of the Window `load` event. Finally, the timer declared in the `load` event handler depends on its execution, and downloads an image resource.

3.5.2 Frame Completion Events

When constructing a web page's dependency graph, we must give particular attention to events that signal the page's transition through its different completion states. Consider, for example, the web page shown by Figure 3.7. The main HTML document (or main frame) contains two resources R1 and R2 (*e.g.* images), in addition to two subframes: iframes 1 and 2. Each iframe is an embedded HTML document with its own resources, R3 and R4 respectively, and iframe 1 has its own inline frame, iframe 3.

Developers can declare event handlers that listen to completion events on any of the frames or the main frame itself. Generally speaking, a web frame has three states of completion, as shown in Figure 3.8:

1. **Loading**: where the browser is still receiving the frame's chunks over the network, and the parser has not completed processing the HTML tokens from the frame. The DOM Document's `readyState` attribute is set to "loading". After the frame completes downloading, and the browser has parsed all of its HTML tags, the frame transitions to the next state. The DOM Document's `readyState` property changes to "interactive", after which the `DOMContentLoaded` event fires.

2. **Interactive**: indicates that the browser has completed loading and parsing a frame's HTML. Nevertheless, some issued resource requests may not have completed, and continue to load during this state. Similarly, the browser may continue to download child iframes, parse them, and download their respective resources even after the parent frame finishes parsing. When the browser finishes downloading these outstanding requests, and completes its subframes recursively, it fires the Window `load` event. At this stage, the Document's `readyState` property changes to "complete".

   Returning to our example from Figure 3.7, the Window `load` event for iframe 1 fires when the browser completes loading R3 and iframe 3. On the other hand, the main frame's Window `load` event will fire after all of the following complete: R1, R2, iframe 1, iframe 2, R3, R4 and iframe 3. The main frame's subframes must *all* go into the "complete" state before their parent can undergo that transition. If the main frame has completed parsing its HTML, it will remain in the "interactive" state while its child frames complete.

3. **Complete**: represents a frame's state after its Window `load` event fires. Any activities that occur after the Window `load` event fall under this state. The Window `load` event handler, when evaluated, may trigger more web page activities, in the form of computations and resource downloads. Similarly, any timers registered during the course of

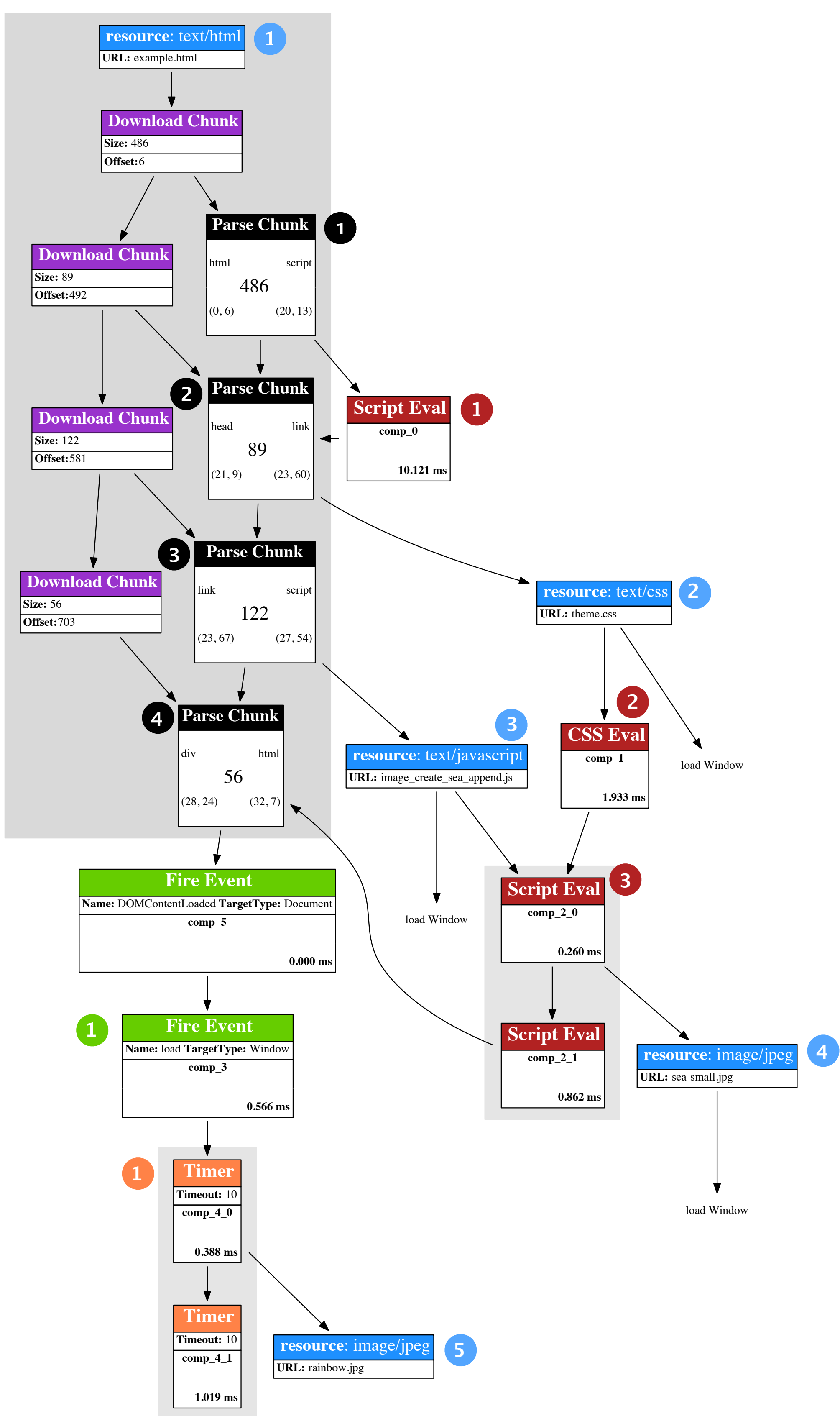


Figure 3.6: Dependency graph of the example HTML page shown in Figure 3.5. Parse chunks are in black, and are labelled with their size, and *(row, column)* pairs indicating the start and end of the chunk. Computations are in red, marked with their identifier and their duration. Resources are blue, timers orange and events are in green. Finally, grey boxes group the parse and download activities of a single frame, or are used to mark sub-computations belonging to the same overall computation. We artificially inserted the `DOMContentLoaded` event to mark the end of parsing.

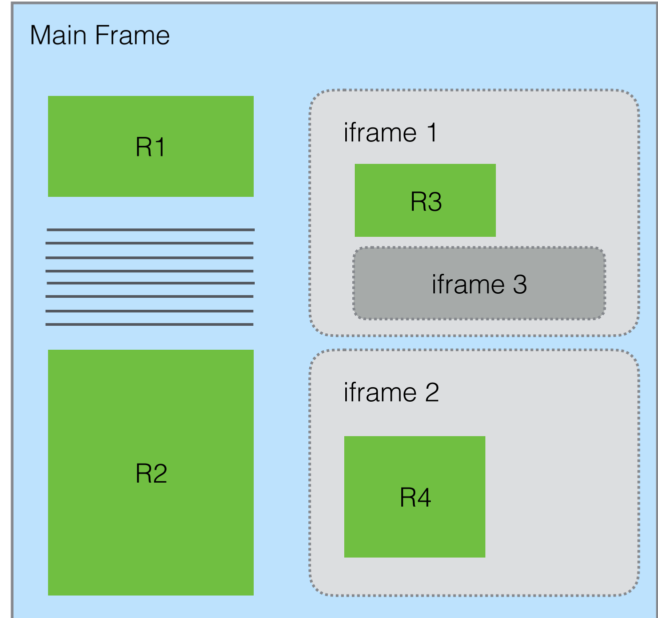


Figure 3.7: An example web page with subframes, to highlight page completion events.

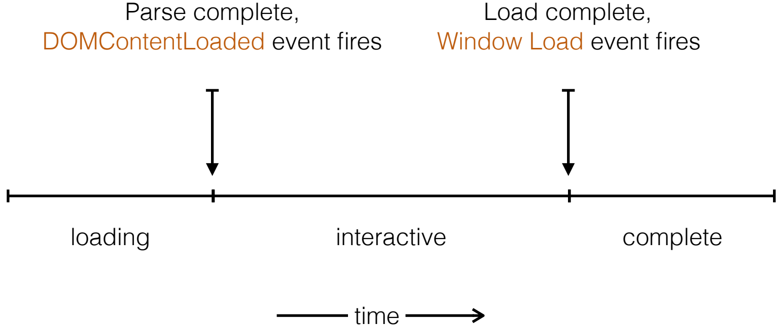


Figure 3.8: Frame load completion states.

the page download may fire after the Window `load` event completes, and may perform various actions that can trigger further downloads.

JavaScript allows developers to register event handlers that fire when any of the frame completion events occurs. Since there are multiple equivalent events that fire on a state transition (*e.g.* `DOMContentLoaded` and Document `readystatechange`), we need to apply some definitive rules with respect to the ordering of these event activities in the dependency graph. Figure 3.9 shows the order in which these completion events occur in the Chrome browser, and we will adopt this order for our dependency representation.

When the browser processes the last parse chunk in the frame, or the last synchronous JavaScript that generates DOM elements, the frame goes into the "interactive" state. The Document `readystatechange`, Document `DOMContentLoaded` and Window `DOMContentLoaded` (bubbled from the Document) events fire in succession. After all its resources and subframes com-

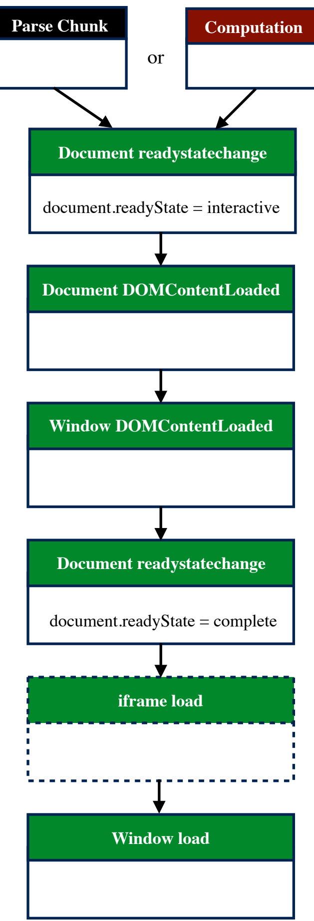


Figure 3.9: Frame completion events chain.

plete, the frame goes into the "complete" state, and the Document `readystatechage`, iframe `load` and Window `load` events fire in succession. The iframe `load` event only fires against a subframe (never the main frame) in response to an `<iframe onload=loadFunction()>` declaration.

Although a completion event may fire, the event handler that deals with it may not have been registered by the web page. As a result, any one of the green boxes may be missing from Figure 3.9. If, for example, the web page does not declare a handler for the Document `readystatechange = interactive` event, the next event in the completion chain whose handler is declared, *i.e.*the `DOMContentLoaded` event, takes its place in the chain and becomes the immediate child of the parse chunk.

In the Emulator, we use the `DOMContentLoaded`, and Window `load` events to signal that a page has completed (see Section 3.7.3). Unfortunately, these events only fire when they are explicitly registered from JavaScript, and developers may not have implemented either of these event handlers in the web pages that we profiled. To simulate these pages correctly, the Graph Builder inserts the completion events artificially at the appropriate points in the dependency graph. In particular, we insert the `DOMContentLoaded` event after the last parse chunk in the

page. As for the load event, we insert it after parsing has completed, and the last resource requested before the end of parsing has completed downloading. These artificial events have zero duration, and only act as markers for the simulation.

3.5.3 Resource Prioritisation

Not all web resources are created equal. Recall, for example, that a CSS file may block any subsequent JavaScript execution until it fully downloads and runs, especially if the script requires attributes specified by the CSS file. The CSS file in this scenario is much more important to the page than, say, an image file on which no other resource depends.

HTTP/2 allows the browser to attach stream priorities to the header of a requested stream. Using the priorities, a client can communicate to the server the relative importance of the requested streams in the form of a "prioritisation tree", indicating how it would like the server to allocate resources like CPU and memory when managing concurrent streams. Additionally, priorities inform the server which stream it should deliver first, in order to improve overall PLT. Note that these priorities are advisory: the server will not block a stream if it cannot make progress on a higher priority stream.

Two fields communicate priority information in HTTP/2's HEADERS frame (Figure 2.3): the stream dependency, and the stream weight. The stream weight is a number between 1 and 256, whereas the dependency contains the identifier of the stream considered to be the requested stream's parent. Combined, these two fields are used to implement either weight-based or dependency-based prioritisation. For weight-based prioritisation, the dependency is set to zero by default, implying that the prioritisation tree is flat with nodes ordered by the stream weights; 256 corresponds to the highest priority. On the other hand, if a stream's parent is specified, then the stream explicitly depends on the parent's completion. On receiving dependency-based priority information, the server will prefer allocating resources to the parent stream rather than its dependent. For streams that share the same parent, the weight field represents the proportion of resources a server should allocate the stream with respect to its siblings.

Browsers can use weight-based and/or dependency-based schemes to implement their own custom prioritisation policies. Firefox, for example, uses a dependency-based scheme [109]. It sets iframes and images to be dependent on CSS and JavaScript files declared in the <head> section. Alternatively, it uses the stream weights to give iframes double the priority of image files, and JavaScript files within <body> a lower priority than earlier JavaScript files.

Previous versions of Chrome, on the other hand used simple weight-based prioritisation based on an object's MIME type [109]. Chrome allocated the stream weights according to the following order:

Type	Weight
HTML	256
CSS	220
JavaScript	183
Image	110

Profiling HTTP/2 enabled web sites using chrome://net-internals shows that more recent versions of Chrome (*e.g.* Chrome 60) use dependency-based prioritisation.

Despite the proliferation of different resource prioritisation schemes across browsers, this study will examine two schemes in particular, described in more detail in the following sections. We compute the resource priorities corresponding to each scheme in the graph building stage, and assign each resource a priority metric in the resulting JSON file for that web site.

3.5.3.1 Priority Based on MIME-Type

The simplest scheme is to have static priorities based on the MIME types of objects—CSS and JavaScript are higher priority than JPEG and MP4 for example. MIME-type prioritisation mimics the weight-based prioritisation previously used in Chrome. Servers like H2O also use this type of prioritisation when the client fails to provide any priority information in its HTTP requests [110]. Under MIME-type prioritisation, we assign the priorities to the resources according to the following rule: HTML $>$ CSS $>$ JavaScript $>$ Images $>$ other MIME types.

3.5.3.2 Priority Based on Critical Path Analysis

In addition to MIME-type prioritisation, we implemented an omniscient dependency-based policy that uses critical path analysis to allocate priorities. These priorities represent a global ordering of all the resources in the web site, such that no two resources share the same priority. The dependency-based policy knows the future, so should outperform any real browser dependency policy.

The critical path of a dependency graph is the set of activities that form the longest path through the graph in terms of total duration. As a result, the duration of a critical path represents the minimum time required for completing a web page. Other activities not on the critical path can be performed in parallel, and reducing their duration will not modify the critical path, or allow the page to complete faster.

Critical path analysis allocates node priorities as follows: the first activity on the graph's critical path will naturally receive the highest priority, since executing it will reduce the remaining time required to download the web page. If we remove this activity from the graph, *i.e.* we execute it, then we must recompute the critical path of the remaining graph to the find the new longest path. The first activity on the new critical path now has the next highest priority, and so forth.

To apply this logic, we first compute the weight of each activity node in the graph. Recall that the Profiler records a corresponding duration for each node. For a resource node, the duration is the time required to request and download it, whereas a computation's duration

Algorithm 1 Assign weights to activities in dependency graph.

```
1: Q ← [all a where GETCHILDREN(a) is ∅]            ▷ Q is a set of activities with no remaining unexamined children;
2: ∀a ∈ Q SETWEIGHT(a, GETDURATION(a))              ▷ Initialize the weights of the leaf nodes to be equal to their duration;
3: while LENGTH(Q) >0 do
4:     a ← POP(Q)
5:     w_a ← GETWEIGHT(a)
6:     for p in GETPARENTS(a) do                    ▷ Loop through all of a's parents;
7:         w_p ← GETDURATION(p)
8:         REMOVECHILD(p, a)
9:         if w_p + w_a > GETWEIGHT(p) then         ▷ Set the parent weight to the duration of the longest path through its children;
10:            SETWEIGHT(p, w_p + w_a)
11:        if GETCHILDREN(p) == ∅ then              ▷ Add parent to Q if all of its children examined;
12:            Q ← Q ∪ p
```

measures to how long the browser took to execute it. The weight of a node represents the duration of the longest path from the node to a leaf in its sub-graph. Algorithm 1 shows how we compute node weights, by applying a mechanism similar to a reverse topological sort and walking the graph backwards starting from the leaves. We update a node's weight as we walk backwards until we finally hit the root node.

Algorithm 2 Assign resource priorities.

```
1: E ← MAXHEAP( )                                   ▷ Create a max-heap sorted by node weights;
2: PUSHHEAP(E, root)                                ▷ Initialize heap with root of graph;
3: priority ← 0
4: visited ← ∅
5: while LENGTH(E) >0 do
6:     a ← POPHEAP(E)                               ▷ Get next node with largest weight;
7:     if ISRESOURCE(a) then
8:         SETPRIORITY(a, priority)                 ▷ If it is a resource, assign it the next highest priority;
9:         priority ← priotity + 1
10:    for c in GETCHILDREN(a) do
11:        if c not in visited then
12:            PUSHHEAP(E, c)                       ▷ Add the node's children to the heap if not visited yet.
13:            visited ← visited ∪ c
```

Using the node weights, we assign resources their appropriate priorities by applying Algorithm 2. The algorithm uses a priority queue to walk forward through the graph and assign incremental priorities to the resource nodes based on their weights. Using this method, the resource with the largest weight (*i.e.* longest path to completion) will have the highest priority, followed by the resource with the second longest path and so forth.

The astute reader will note a limitation of applying critical path analysis in this manner: resource durations depend on the network bandwidth and latency. The network conditions under which we recorded resource durations using the Profiler do not match those chosen for our experiments. If the network changes, the critical path will change. For example, if the network is infinitely fast, fewer resource downloads will appear on the critical path, and computations will become dominant. Given the shortcomings of a static analysis, a possible alternative would be to compute the critical path dynamically [111], based on estimates of the network bandwidth and latency during a page download. But even this method is wrought

with pitfalls, since the estimated duration of a resource may be incorrect if there is congestion on the path. Another mechanism involves using node-depth prioritisation, where rather than using measured node durations, we set the durations of all resource nodes to a constant and other node durations to zero [51]. In this case, the longest path in terms of depth forms the critical path. We briefly experimented with this type of prioritisation and found that, although most times it produced values very similar to those generated by regular critical path analysis, it fails to attribute higher priorities to large resources , or resources that trigger long computations that block page loads. In the end, we have found that critical path priorities computed statically provide a good measure of the relative importance of resources to the completion of a particular web page and generally help reduce PLT, as we shall show in Section 4.4.4.

3.6 Graph Viewer

The Graph Viewer can be used to visualise the dependency graphs of the 300 top Alexa web sites we have profiled. These visualisation and dependency graphs can be found at the link provided in [3]. We have used the the Graph Viewer earlier to build the graph for Figure 3.6.

Figure 3.10 shows the dependency graph generated for amazon.com, as an illustration of the complexity and number of dependencies that comprise a fairly typical web page. It is not necessary to read the text in the boxes, but the curious reader will note that it is possible to zoom in to the PDF version. Amazon relies heavily on events and timers to download the images displayed in the web site, which appear in the figure as the side graph branching off from the vertical main page.

3.7 Emulator

Our emulator is built as an extension to the ns-3 network simulator [35]. It models three main components that, when combined, simulate a typical web page download: the client's browser, the web servers, and the underlying network. The browser loads a web dependency graph and requests resources and download chunks as their dependencies are satisfied. The server responds to the requests after applying the server processing delays either serially, in parallel, or out-of-order. Finally, network downloads use ns-3's TCP implementation, extended to support several features that improve TCP performance, as we describe in Section 3.7.7. We augment ns-3 with our implementation of various web protocols, ranging from the very basic HTTP/1.0 through HTTP/1.1's variants and finally to HTTP/2. The following sections discuss the three components in more detail.

3.7.1 Dimensionality of Problem Space

We would be remiss if we did not preface this section with some insight about the scope of the parameters we must examine for a thorough analysis of web downloads. Table 3.2 summarises some of the dimensions we can explore, and the choices we made with respect to the parameters we modelled and emulated *vs.* those we didn't.

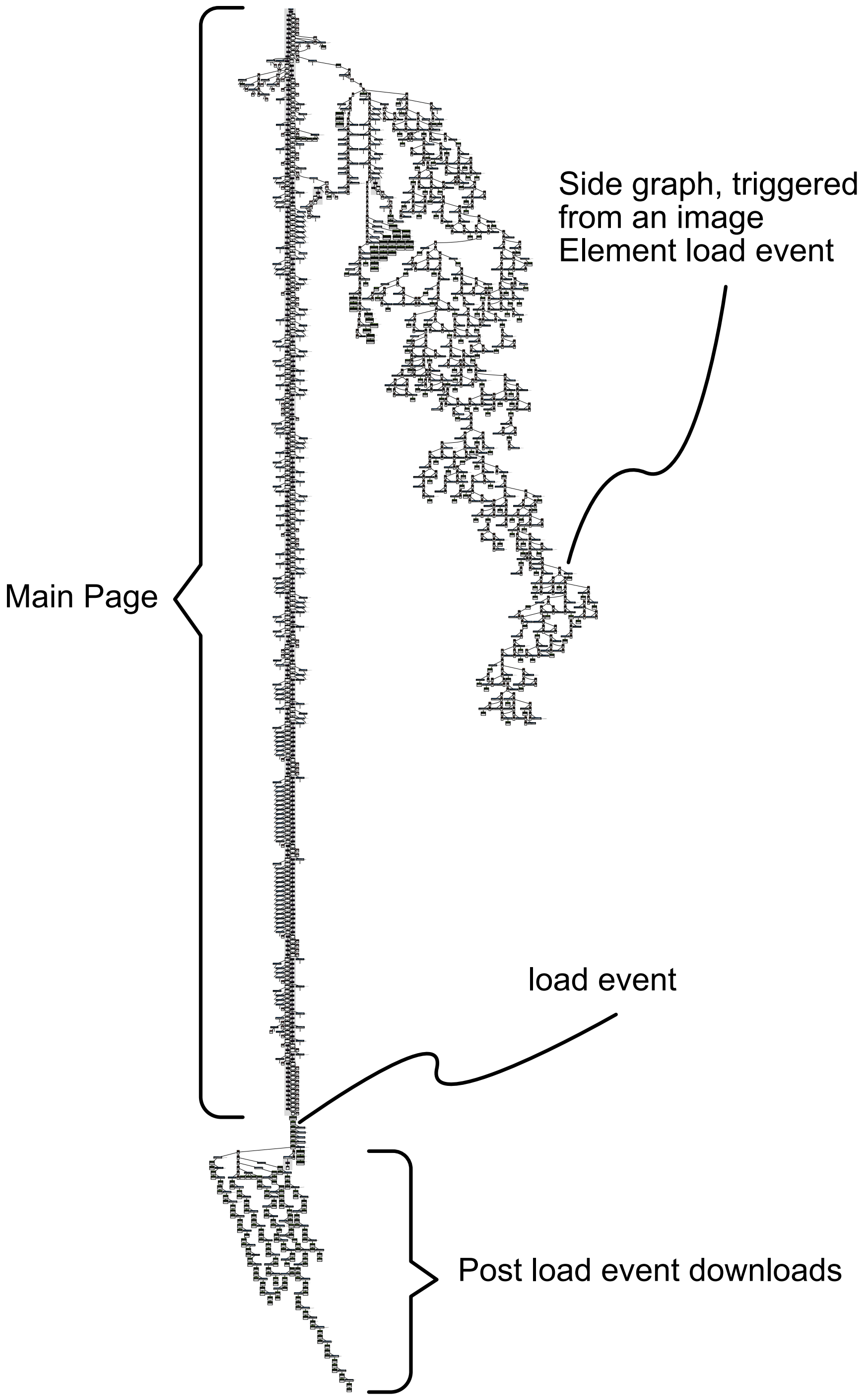


Figure 3.10: Zoomed out view of dependency graph of amazon.com.

Dimension	Values	Emulated
Bandwidth	Several in the range of 500Kbps - 1Gbps.	✓
RTT	Several in the range 8ms - 400 ms.	✓
Web Protocol	HTTP/1.0, HTTP/1.1, HTTP/1.1 Pipelined, HTTP/2, QUIC.	✓
Connections	1 - 12	✓
Server architecture	Serial, parallel and out-of-order (Section 3.7.6).	✓
Prioritisation	MIME-type, critical path (Section 3.5.3).	✓
Web site	300 web sites profiled from Alexa 500	✓
Caching	Hot or cold. (Section 3.7.5).	Cold
TLS	On or off (Section 3.7.2).	Off
DNS	Cached or uncached (Section 3.7.2).	Cached
Completion Metric	`DOMContentLoaded`, `load`, first-paint, and full graph (Section 3.7.3).	`load`
Link Layer	WiFi, 3G/LTE, and point-to-point (Section 3.7.7.1).	Point-to-point
Buffering	Drop tail, RED, or FQ-CoDel (Section 3.7.7.3).	RED
Transport Protocol	TCP New Reno (+ SACK), TCP CUBIC, QUIC, and BBR (Section 3.7.7).	TCP New Reno + NACK
Initial cwnd	1, 4, 10 (Section 3.7.7.2).	4
Background traffic	On or off.	Off
Sharding	Sharded or unsharded (Chapter 5).	✓
MPTCP scheduler	RTT or Stream (Section 6.4.1).	✓
MPTCP topologies	Equal paths, RTT mismatch, bandwidth mismatch, joint (Section 6.3).	✓

Table 3.2: Dimensions for the web download experiments, and our selected parameters for emulation.

From the table, we can conclude that the dimensionality of our analysis is quite large, with each dimension assuming several possible values. As the number of dimensions grows, it becomes more difficult to draw any meaningful conclusions about the effect of each dimension on the PLT. We need to render the space sufficiently simple to reason about. As a result, we picked the parameters we deemed most important in terms of their effect on PLT. Although we excluded some options, we have retained enough complexity to allow us to understand how page loads behave for a large set of conditions, and the intricate interactions between the chosen parameters and page loads. Note that despite our choice to fix some parameters, PCP's adaptable design allows us to experiment with the full range should we see fit in the future.

Within the set of chosen dimensions, we do not emulate the entire space of possibilities. For a particular analysis, we narrow down the parameters and choose sensible ones that are particularly interesting for the analysis in question. In the coming sections, we explain as we go along why we excluded some choices and why it is reasonable to do so.

3.7.2 Browser Model

The browser model consumes a profiled web page's dependency graph, and performs the correct actions: it waits the recorded time for each computation or parse chunk, and issues the requests for resource activities as necessary. Algorithm 3 shows the core of the browser's op-

eration. Each time an activity node in the graph completes, the browser attempts to execute all of its children. A child is only executed when all of its parents have completed. To execute resource nodes, the emulated browser issues the necessary HTTP requests over the network, using configurable web and network protocols. These requests can either be GET requests that result in a resource download, or POST requests that push data onto the server. We can therefore run experiments with different web and transport protocols, link rates and RTTs with ease. Computation activities, which consist of parsing and script evaluations, are handled differently: the browser pushes them onto a scheduling queue, which we will describe in more detail below.

Algorithm 3 Executing the activity nodes in the dependency graph.

```
1:
2:  function EXECUTE(activity)                          ▷ Attempt to execute an activity node.
3:      parents ← GETPARENTS(activity)
4:      isReady ← True

5:      for activity p in parents do   ▷ If all the activity's parents have completed, the activity is ready for execution.
6:          isReady ← isReady and ISCOMPLETE(p)

7:      if isReady then
8:          if ISEVAL(activity) then
9:              PUSH(scheduler, activity)       ▷ If the activity is an evaluation, push onto scheduler queue.
10:             EVALUATEPENDING(scheduler, currentFrame)
11:         else
12:             SENDHTTPREQUEST(activity)     ▷ If the activity is a resource, send corresponding HTTP request.
13:             WAIT( ) for HTTP response.
14:             COMPLETION(activity)

15: function COMPLETION(activity)        ▷ Function that executes at the completion of the last activity.
16:     SETCOMPLETE(activity)
17:     children ← GETCHILDREN(activity)
18:     for activity c in children do
19:         if  not ISCOMPLETE(c) then EXECUTE(c)

20: EXECUTE(rootActivity)                 ▷ Begin the dependency graph by executing the root activity.
```

The browser does not model DNS exchanges for resolving domain names, or TLS handshakes for HTTPS connections. In terms of DNS, our experiments operate as though all the HTTP requests result in a DNS cache hit. For a particular network configuration, DNS resolution will consistently add an equivalent fixed overhead to the PLT regardless of the web protocol examined, therefore will have no effect on protocol ordering. Reducing this overhead would require mechanisms for prefetching or optimising DNS itself, which are outside the scope of this study. The TLS handshake also adds a fixed overhead to each newly established connection. The client and server exchange several messages for cipher suite negotiation, server authentication and session key exchange, which adds at least 2 RTTs of delay to the setup of every new connection. The TLS handshake forms a large proportion of download time for HTTP/1.1 flows that open several connections per domain, but is amortised for HTTP/2, which opens one connection. Studies evaluating QUIC show that the handshake latency (which includes both TCP and TLS handshakes) increases linearly with RTT [21], and in consequence reducing the time spent on the TLS handshake is pivotal for reducing PLT.

Reducing the overhead of TLS requires employing new mechanisms for establishing the cryptographic handshake, and these are not the focus of this thesis. Instead, we decouple TLS from our analysis. We leverage the fact that QUIC and TLS 1.3 have reduced the number of RTTs required for connection establishment to 1-RTT from a client to a new domains, and 0-RTTs for repeat connections to the same domain [61]. Since techniques exist for shrinking the TLS handshake, we choose to discount TLS from our analysis, and focus only on the core exchange of data in a web download.

Enabling TLS would skew results in favour of web protocols that use fewer connections to each domain server. We expect, for example, that HTTP/1.0 would produce larger PLTs with TLS enabled than disabled as it is in our experiments. HTTP/1.0 is the worst-performing protocol, even without the additional TLS overhead, which implies that enabling TLS would not change its overall ranking. As for the other web protocols we examine, if they establish same number of connections to each server, then the TLS overhead is constant. Otherwise, we expect protocols which establish more connections to perform relatively worse in terms of PLT.

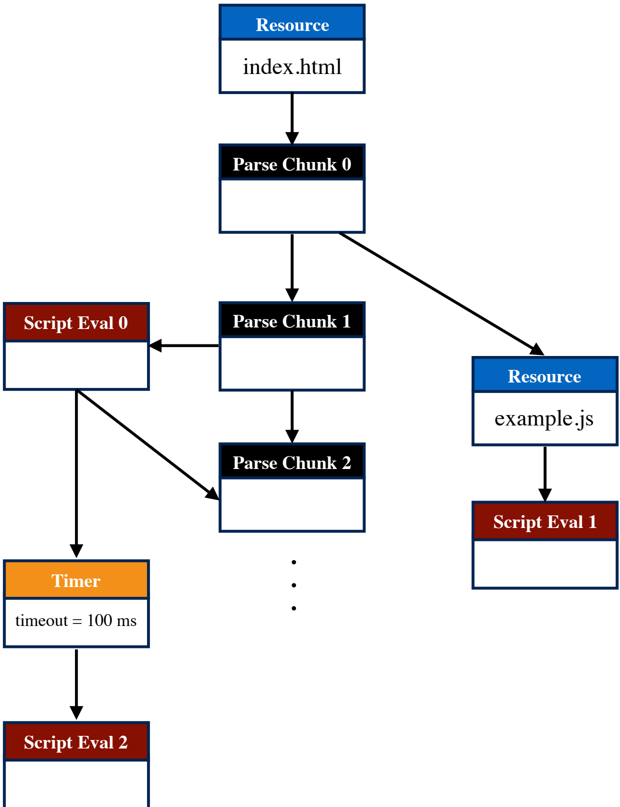


Figure 3.11: Example web page dependency graph with ambiguity regarding the order of executing Script Eval 1 and Script Eval 2.

While most of the browser behaviour is determined by the web dependency graph and by the arrival of data over the simulated TCP connections, some key properties of web browsers remain to be defined. The web dependency graph, by definition, constructs the dependency relationships between the different web activities without any knowledge of the underlying

network conditions and protocols employed. As a result, scheduling dependencies that precipitate due to the single-threaded execution of parsing, JavaScript and CSS are not captured by the Profiler trace, since they depend on the underlying network properties.

To elaborate, consider the dependency graph shown in Figure 3.11, in particular the activities denoted by Script Eval 1 and Script Eval 2. Script Eval 1 represents an asynchronous JavaScript evaluation as a result of downloading the *example.js* file, whereas Script Eval 2 is the result of evaluating a programmatically defined script declared within a timeout function. The Graph Builder does not know when the browser will complete downloading *example.js* at the point when it constructs the graph. If *example.js* is a small file, and the network has low delay, Script Eval 1 may complete execution well before the Timer fires and triggers the execution of Script Eval 2.

On the other hand, if the network experiences long delays, the *example.js* file may arrive half-way through executing Script Eval 2. Because the browser must execute JavaScripts serially on a single thread, it will wait for Script Eval 2 to complete before executing Script Eval 1. Consequently, there exists an implicit dependency between the two activities, such that Script Eval 2 must precede Script Eval 1. This dependency is dynamic, and cannot be determined in advance due to the variability of the network and transport protocols. Such scheduling dependencies can only be determined at runtime, and a few more examples include:

- The order in which the parser processes tags from an iframe versus its parent frame depends on when the iframe completes downloading. The parser is single-threaded, therefore tags from either frame cannot be processed in parallel. In Chrome, the parser operates on the main frame first, only switching to an iframe if the main frame parsing is blocked, for example by a synchronous JavaScript file download.

- Asynchronous JavaScript must execute when the parser has yielded, and will block subsequent parse chunks. To determine which particular parse chunk must wait, we must know when the JavaScript file completes downloading. Note that a JavaScript file programmatically downloaded by an inline script is treated as asynchronous by the browser as well.

- Timer and event handler functions may fire while the main thread is parsing the main HTML file, or any of its enclosed iframes.

We have implemented a *scheduler* in the Emulator's client browser to enforce an ordering on the various serial activities. Only resource downloads can occur in parallel. The scheduler must consequently handle all the evaluation activities: parse chunks, timers, events, script and CSS. Closely modelling browser behaviour, it processes activities pertaining to a single frame first, and only move on to the next frame if it has run out of work to do.

To this end, the scheduler maintains one evaluation queue for each frame displayed in the web page. It also owns a global priority queue for timer and event activities. As the Emulator

walks the dependency graph and executes the activity nodes, if it encounters an evaluation activity it does not execute it right away. Instead, the scheduler enqueues the activity onto its appropriate frame queue, as we've seen in Algorithm 3, Line 9. For these frame queues, parsing activities take precedence over JavaScript and CSS evaluation. Since a browser executes timers and events preemptively, these activities are appended to the priority queue.

Nested timers can hog the execution queue with this scheduling model, especially if a web site is designed poorly and has long timer chains with zero timeout values. Since they will always have a higher priority, these nested timers will block other computation activities from executing. In reality, Chrome clamps the minimum timeout to 4ms (for nested timeouts with more than five levels) in order to prevent the CPU from spinning. We adopt this design choice for our purposes as well, since it allows parsing and computations to interrupt the execution of long timer chains.

Algorithm 4 Scheduler operation.

```
 1: function EVALUATEPENDING(scheduler, currentFrame)
 2:
 3:     if ISEVALUATING(scheduler) or not currentFrame then
 4:         return          ▷ The scheduler is already evaluating an activity, or has no more work to do.
 5:
 6:     frameQueue ← GETQUEUE(currentFrame)
 7:     priorityQueue ← GETPRIORITYQUEUE()
 8:     if not ISEMPTY(priorityQueue) then     ▷ Try to evaluate any timer or event activities first if they exist.
 9:         activity ← POPFRONT(priorityQueue)
10:         Schedule COMPLETION(activity) after GETDURATION(activity)
11:
12:     else if not ISEMPTY(frameQueue) then
13:         activity ← POPFRONT(frameQueue)            ▷ Evaluate the frame's activities.
14:         Schedule COMPLETION(activity) after GETDURATION(activity)
15:
16:     else            ▷ No work to be done in this frame, switch to the next one.
17:         nextFrame ← SELECTNEXTFRAME(currentFrame)
18:         EVALUATEPENDING(scheduler, nextFrame)
19:
```

Each time a new activity is pushed to a queue, or an activity completes, the scheduler calls EVALUATEPENDING(), whose definition is shown in Algorithm 4. Beginning with a pointer to the page's main frame, the scheduler will execute the activities at the front of the main frame's queue in order (but prioritising any event or timer activities). If the main frame's queue becomes empty, for example because we are waiting for a synchronous JavaScript file to download, or we have completed executing all its computations, the scheduler attempts to switch to the next viable frame. The mechanism for switching between frames is shown in Algorithm 5. First, the algorithm checks if any of the current frame's subframes have any pending computations, then it checks the current frame's parent. Finally, the algorithm returns the next possible frame from the list of remaining unprocessed frames.

The scheduler therefore models the serial execution of activities in the browser, and the transition between processing activities from one frame to those of another frame until the entire web page activities have been processed.

Algorithm 5 Selecting the next frame for the scheduler.

```
1:
2:  function SELECTNEXTFRAME(currentFrame)
3:
4:      remainingFrames ← all web frames
5:      subframes ← GETSUBFRAMES(currentFrame)
6:
7:      for Frame f in subframes do                ▷ Attempt to switch to one of the subframes.
8:          queue ← GETQUEUE(f)
9:          remainingFrames ← remainingFrames − f
10:         if not ISEMPTY(queue) then
11:             return f
12:
13:     parentFrame ← GETPARENT(currentFrame)
14:     queue ← GETQUEUE(parentFrame)
15:     remainingFrames ← remainingFrames − parentFrame
16:
17:     if not ISEMPTY(queue) then                 ▷ Attempt to switch to the frame's parent.
18:         return parentFrame
19:
20:     for Frame f in remainingFrames do          ▷ Otherwise find the next available frame with work enqueued.
21:         queue ← GETQUEUE(f)
22:         if not ISEMPTY(queue) then
23:             return f
24:
25:     return NULL                                ▷ No more evaluations enqueued for any for the frames.
```

3.7.3 Measuring Completion

What metric should we use to measure when a page load is complete? One metric often used is "first paint": the time when the first part of the web page is rendered. Another measure of page readiness involves determining the page's visible completeness score in the style of Google's Speed Index [112]. Overall, this score measures how quickly the page's content is painted in the visible viewport. At the other extreme, we could measure when the very last resource associated with a page is processed. However, many web pages use event handlers and timer chains to load additional resources after the main page has finished loading, and sometimes these timer chains continue indefinitely. We need a completion time that is observable as an event in the web dependency graph, or we cannot perform an apples-to-apples comparison between two protocols. First paint and visual completeness fail this requirement—we do not emulate rendering, so cannot tell when a browser would first render part of the page.

Two events in the traces are correlated with page load time and always occur: `DOMContentLoaded` and the page `load` event. Neither is perfect; the former can fire even before first paint in the case of iframes, and the latter might wait until off-screen inlined images have loaded. The effects we observe affect both metrics, but to different degrees. We configure the Emulator to measure the time it takes to reach both of these completion points. We mostly show results using the Window `load` metric for completion, but also discuss `DOMContentLoaded` in a few of the contexts we examine. The `load` event correlates well with when the user believes the page to be ready to interact with [12], but also gives a more consistent metric, as described in the next section.

3.7.4 Ensuring Consistency

For an apples-to-apples comparison, two downloads of the same page should measure the same amount of work. However, changing the download order of some resources can alter the moment when timers and events fire, which in turn could change the amount of work done before the `load` event fires.

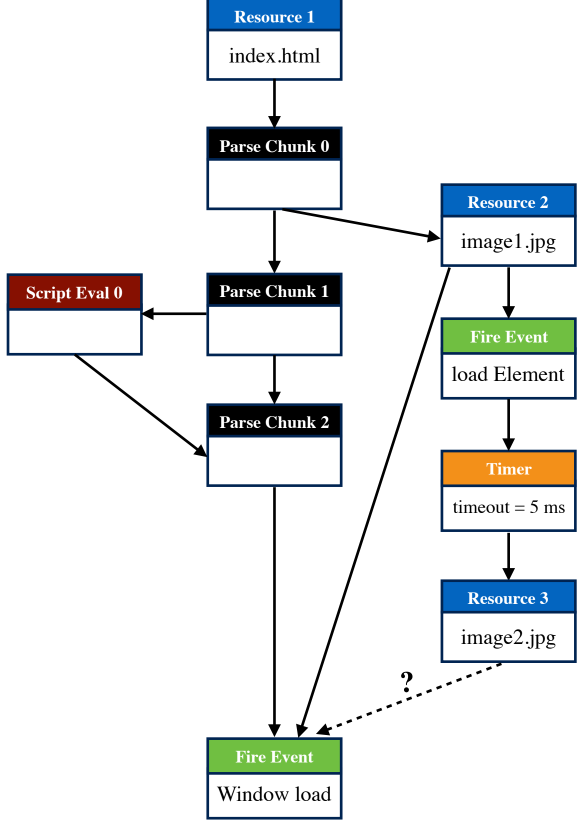


Figure 3.12: Example web dependency graph where the Window `load` event may have different parents depending on the network conditions.

As an example, consider the dependency graph Figure 3.12. When the browser has finished parsing Parse Chunk 2 for the web page shown, it must wait for Resource 2 (*image1.jpg*) to complete before it can fire the Window `load` event. If the network is fast enough, Resource 2 will complete before the browser completes parsing, which will prompt the execution of the following activities in order: the Element `load` event, the Timer and the request for Resource 3. Because the browser issues the request for Resource 3 before it completes parsing, it will delay firing the Window `load` event till Resource 3 has completed. In this scenario, the Window `load` event has three parents: Parse Chunk 2, Resource 2 and Resource 3.

On the other hand, if the network suffers from long delays, the browser will complete downloading Resource 2 after parsing the entire page. The `load` event will fire as soon as Resource 2 has been downloaded, since the timer has not fired yet, and there is no pending request for Resource 3. The `load` event as a result only has two parents: Parse Chunk 2 and

Resource 2. The dependency graph has changed in response to the network parameters, and this discrepancy is based on whether the event and timer handlers are executed before or after the `load` event.

We do not want this behaviour. In fact, we would like to maintain consistent dependency graphs in the light of varying network conditions so that we can generate comparable and reproducible experiments. To ensure the work done is the same in all runs, we *normalise* the trace. All timers in the original profiled trace that occur *after* the page `load` event are marked accordingly by the Graph Builder. We consequently disable them in the dependency graph before we emulate the page download. Timers with timeout values larger than one second are trimmed from the emulated graph by default, since it is reasonable to assume web page designers intended for them to run in the background, and to fire after parsing of the main HTML has completed. The Graph Builder similarly marks all computations to indicate whether they were evaluated before or after the `load` event. As a design choice, if any part of an iframe is downloaded before the `load` event, we mark all of its resources and computations as parents of the `load` event, so that the full iframe download is emulated in our experiments. By emulating only timers and computations that occur before the `load` event, we ensure that the amount of data downloaded is the same across all protocols and parameters, and the browser performs the same amount of computations, allowing meaningful comparisons.

We cannot reasonably normalise the traces for `DOMContentLoaded`, as this event typically occurs while important resources such as images and sometimes iframes are being downloaded. Normalisation here would result in a web page that behaves very unlike the original. Therefore, for experiments where we want to measure the completion of a page based on `DOMContentLoaded`, we emulate a web page download to the point where it fires the load event, but record the time the `DOMContentLoaded` event fires.

3.7.5 Browser Optimisations

Modern browsers apply several optimisations in attempt to speed up web page load times. The following section lists some important optimisations and discusses which of them we have chosen to implement in our browser model, and why.

Caching

Browsers usually maintain a cache of resources they have recently requested and downloaded in the past. Any future requests for the same resource on the same page, or even another new web page, do not need to access the network, but are rather retrieved from the local cache, allowing page loads to complete faster. The browser cache provides a large performance win for periodic accesses to the same web site. Common JavaScript libraries, analytic scripts, even embedded Twitter and Facebook widgets, can recur across multiple web sites, and will rarely require a network download, since they will most likely reside in the browser's cache.

The browser stores an expiration date for each object in its cache, in addition to the last time it was modified, if that information was returned in the HTTP response. A server can use

one of several header fields in the HTTP response to specify caching behaviour. The simplest of these is the `Pragma:No-cache` header field that was originally used with HTTP/1.0 to indicate that a resource should not be cached.

Alternatively, if the server uses the `Last-Modified:<date>` header field with no other caching directive, it alerts the client to the date the resource was last modified. The client will subsequently send future requests to the same resource with the `If-Modified-Since:<date>` field, to check if the server has a new version of the resource.

The server can use the `Expires:<date>` header field to tell a client when it should invalidate a resource in its cache. Finally, the `Cache-Control` header field overrides the previous fields, and offers more fine grained cache directives. The most pertinent are:

1. No-Store: notifies the browser that it must not cache the current response, but initiate a new HTTP request for any future references to this resource. The same effect is achieved by the use of the No-Cache and/or Must-Revalidate directives.

2. Max-Age: indicates that the browser can store the current response for the specified time in seconds. After this time, the cached response is considered stale. The server also appends the resource's ETag to the header, an identifier for its specific version of the resource, typically a hash of some contents of the file.

Once a resource expires in the browser's cache, the browser must request it again from the server. The server could, at this point, respond by transmitting the full resource over the network. However, if the resource hasn't changed on the server, resending the entire resource is wasteful since the browser already has it in its cache. The client uses `If-Modified-Since` header field, and the resource's Etag in the HTTP request header to check whether the server's version of the resource has changed in the interim. If the resource has not been modified by the server, it can reply with an `HTTP 304 Not Modified` response, allowing the client to avoid downloading the resource again. If the requested object has in fact changed, the download will proceed as usual.

To understand the amount of caching a typical web site undergoes, we recorded the maximum age of resources from the packet traces captured by the Profiler. We inspected the cache related fields in the HTTP response headers for non-TLS connections. First we checked the `Cache-Control` field, and in its absence, we checked for the `Expires` and `Pragma:No-cache` header fields in order, to record the maximum age of a resource in seconds. We found around 15K resources with the aforementioned cache control directives in our traces (which had around 20K resources downloaded using HTTP). There were 1371 resources with no cache directives at all. A browser will typically not cache resources that lack any cache directives, implying that they effectively have an age of zero. But since we do not know for certain if the web site designers intentionally omitted cache control, or whether this is in fact a bug, we do not include these resources in the data used to examine caching behaviour.

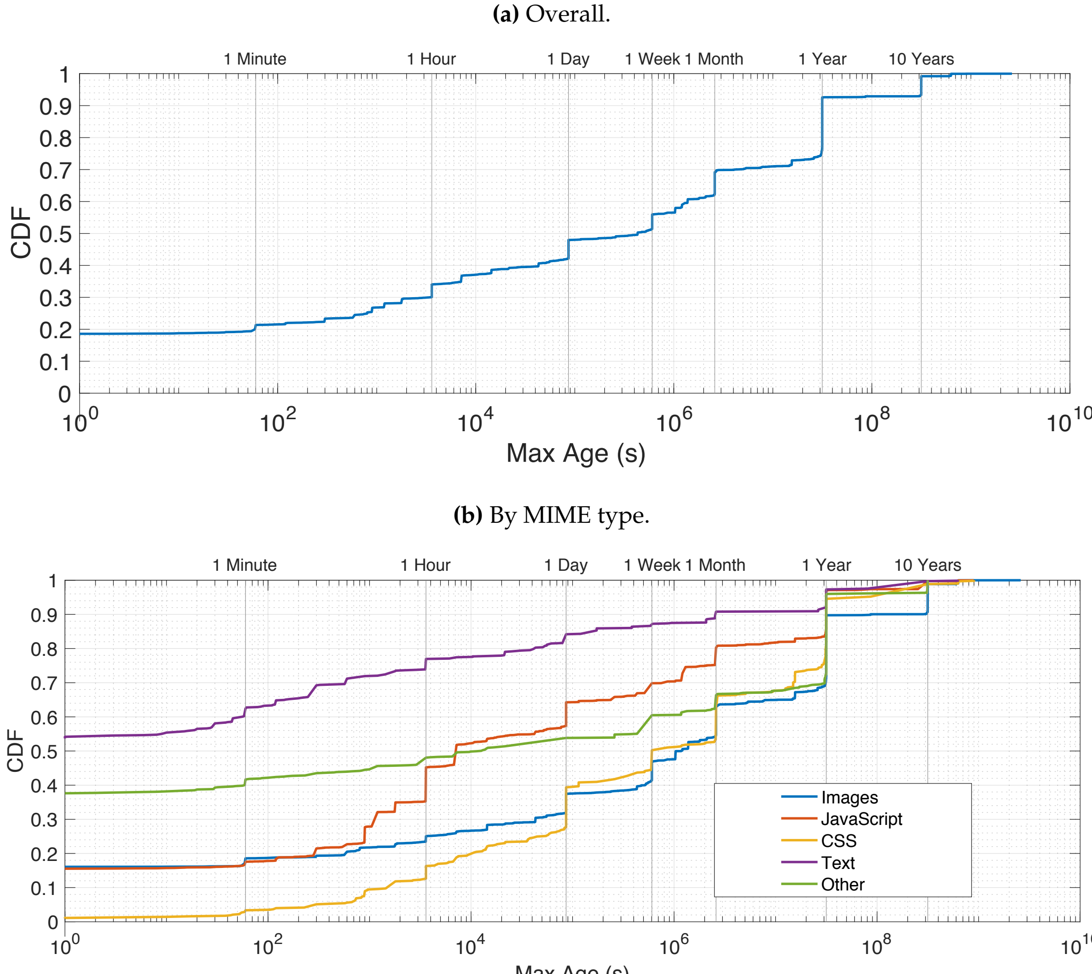


Figure 3.13: CDF of the resource maximum age.

Figure 3.13(a) shows the CDF of the maximum age of resources over all the web sites we profiled. For about 20% of the resources downloaded, the servers specified the No-Store cache control (or a Max-Age of 0), explicitly asking the browser not to cache those particular resources. Instead, the browser will always issue a fresh request for these resources in the future. Although nearly 30% of the resources had an expiration date of over one year, 50% of the resources downloaded expired after one day or less. Figure 3.13(b) shows the breakdown of resource maximum age by MIME type. Images and CSS files are cached for the longest durations, since they are unlikely to change. JavaScript shows more of a spread, where some (probably dynamically generated) scripts expire after one day, whereas others, typically libraries, exhibit longer refresh cycles of over one week. Finally, HTML files must be refreshed much sooner, with nearly 80% of text resources expiring after one day.

Another interesting metric to examine is the fraction of the total resources in a web site which must be refreshed after a certain period of time has elapsed. Given the maximum age of resources in a page and a particular time in the future, we compute the fraction of the total resources which must be refreshed for each web site, and show the median, lower and upper

quartiles across web sites in Figure 3.14. On average, nearly 60% of a web site's resources must be refreshed after one day.

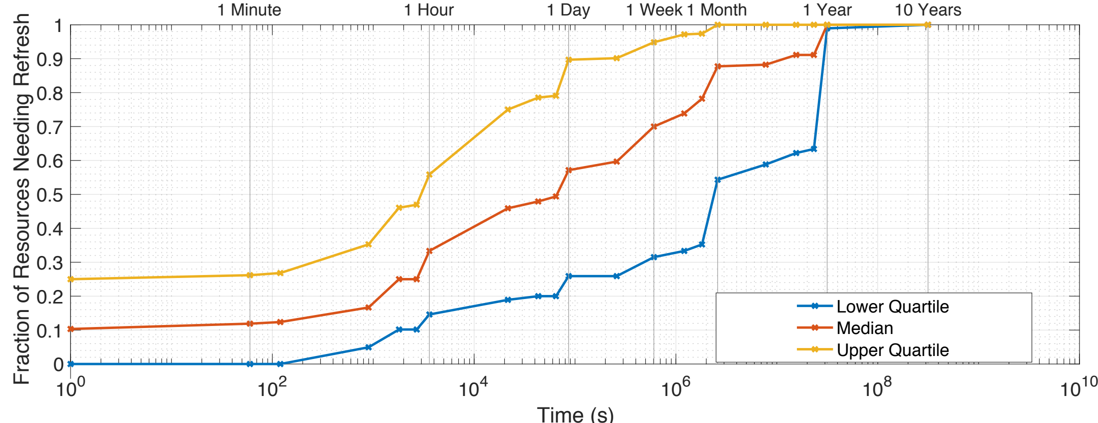


Figure 3.14: Fraction of resources which require refresh after elapsed time.

With the exception of using a local session cache, we do not model the browser cache in the Emulator. Our intention is to model and investigate cold cache web downloads. There are two reasons behind this choice.

First, cold page loads are a typical use case. A user may be accessing the web site for the first time, may have cleared the cache, or may be operating with browser settings that limit the cache size. According to a study done by at Yahoo! in 2007, 40%-60% of users arriving at their web page did not have a primed cache [113]. Additionally, results in Figure 3.14 show that on average a large proportion of a web site's resources require refresh after one day, and for web sites in the upper quartile this proportion grows to 90%. Note that these results only apply to the cases where the examined web site remains static from day to day. Popular home pages, like the news or shopping sites we've examined, change daily and require the download of new images, HTML, JavaScript and ad iframes, although some CSS and script files may not be modified as often (evidenced by Figure 3.13(b)). As such, our model, which operates with an unpopulated cache, approximates a common scenario.

Second, although absolute page load times would improve with a warm cache, the goal of this study is to emphasise differences between protocols and reveal lessons about their behaviour and efficacy, which may be hidden by the cache. We aim to examine and improve PLT specifically for cases when the network is the bottleneck. Cold page loads represent the worse case PLT a user can experience. Protocols that improve PLT for this case will usually also produce improvements for cases with a warm cache where a fraction of requests still need to access the network, but the effect with be smaller.

Wang *et al.*'s WProf work shows that web sites refreshed immediately after they are loaded have 90% of their objects cached, and experience a PLT reduction of around 40%.[2] Note that JavaScript execution time is also reduced with caching. Modern browsers compile JavaScript

[2]PLT reduction is not proportional to number of cached objects, since a lot of them are not on critical path.

and cache intermediate computation steps. For example, Chrome's V8 JavaScript engine stores a local copy of the compiled code using a hash of the source, which persists across page reloads, and allows Chrome to reduce of JavaScript parse and compile time by 40% [114].

If the cache is primed for a particular web page, a fraction of its resources will not require downloading over the network, which effectively alters its dependency graph structure. Cached resource durations will consume zero time and computations and server delays will consume a larger proportion of the critical path. In consequence, the relative ordering of web protocols may change when comparing cached and uncached PLT for the same web site.

Nevertheless, if cached web pages still send a reasonable subset of their requests over the network, and if we consider the relative ordering of web protocols across *all* the pages, then the relative protocol rankings might not differ when the web pages are cached *vs.* uncached. Statistically, although an individual web page's performance may change when cached, overall the trends across all web sites will most likely be maintained. Since fewer transactions occur over the network for cached web pages, the relative win of one protocol over the other will be diminished. Caching will also reduce the benefits of resource prioritisation, since high priority objects like CSS files will likely reside in the browser's cache.

Although we simulate web downloads with a cold cache, we have implemented a very rudimentary session cache, without entry expiration and HTTP cache directives. If a resource is requested multiple times on the same page, *e.g.*there are several `<image>` tags with the same URL, then subsequent requests for the image will hit the local cache and forgo the network entirely.

Preloading

To minimise page completion times, a browser must ultimately download a web site's required resources in parallel with parsing its HTML and executing JavaScript and CSS. However, any synchronous JavaScript object encountered by the parser will block further parsing until the script downloads and completes execution. Only then can the browser resume building the DOM tree.

Modern browsers improve parallelism by implementing a crucial optimisation: the browser *preloads* some resources embedded later in an HTML page while its primary thread is blocked. By activating a second lightweight parser, the browser scans the rest of the HTML tags without constructing any DOM nodes or executing scripts, and triggers the download of images, JavaScript and CSS resources that it encounters as it goes. Iframes, on the other hand, are not preloaded. With preloading, the browser hopes that by the time the main HTML parser resumes, most of the resources it will need to request will have already been downloaded and stored in the cache.

Originally, preloaders downloaded objects in the order they have been referenced in the HTML, but newer browser versions assign priorities to the preloaded resources. For example, Chrome prioritises images that are visible in the viewport over an occluded image, and

blocking JavaScripts over asynchronous ones.

We have implemented preloading in our emulator, modelled after Chrome's preloader policy but excluding request prioritisation. In addition to the main dependency graph, whenever a frame's download chunk completes, the emulator creates a clone of the associated parse chunk. The cloned chunk retains a pointer to all the original chunk's child activities that are CSS, image or JavaScript resources. The clone is then added to the frame's special preload queue. If a frame becomes blocked on downloading a synchronous JavaScript, the browser runs the preloader and drains the preload queue for that frame, executing the associated parse chunks and requesting any resources that are referenced. The browser pauses the preloader when the blocking JavaScript downloads and completes execution. As such, the preloader will only pre-parse HTML from the current frame that is blocked, and will not begin pre-parsing other iframes which have been downloaded.

We can enable or disable the preloader to observe its effects. Unless explicitly stated, our experiments are performed with the preloader enabled.

Lazy Binding

Lazy binding is a technique browsers primarily use when evaluating CSS files. If a CSS file contains an image which will used as a background or to form CSS sprites, this image is not downloaded immediately when the file is evaluated. Instead, the browser waits until it encounters an HTML tag decorated by that particular image before issuing the resource request. Lazy binding ensures that potential style resources are only downloaded when the browser will in fact use them.

We model lazy binding in our web page dependency graphs. In fact, we attribute resources which may be downloaded lazily to the parse chunk which has in fact triggered the download, rather than CSS evaluation activity.

3.7.6 Server Model

A web server must perform several operations to generate and transmit a response to a particular HTTP request. The server listens on a particular socket and reads any incoming requests on that socket. On receiving a request, the server must perform some processing to parse the request header and body. It then generates the corresponding response header and content, *e.g.*by reading a static resource from disk or assembling the content from several backend servers. The server can also generate some dynamic content; for example, it can construct the page's HTML on the fly using PHP or Node.js. Finally, it writes the response on the outgoing socket. These I/O and CPU operations do not consume negligible time, and depending on whether the cache is hit, may last several hundreds of milliseconds.

As such, we cannot discount the contribution of a web server's processing time from the total page load time of a particular web site. In fact, these processing times may even form the bottleneck in terms of page completion, especially when network speeds increase. Furthermore, they may be a source of head-of-line blocking depending on the server design itself, as

we shall demonstrate.

In order to model server processing times correctly, one must first understand the architecture of a typical HTTP server, and the different ways it can process incoming requests. A web server must be capable of supporting thousands of incoming connections, therefore it must have a multi-processing architecture, which can typically be classified into two types [115]:

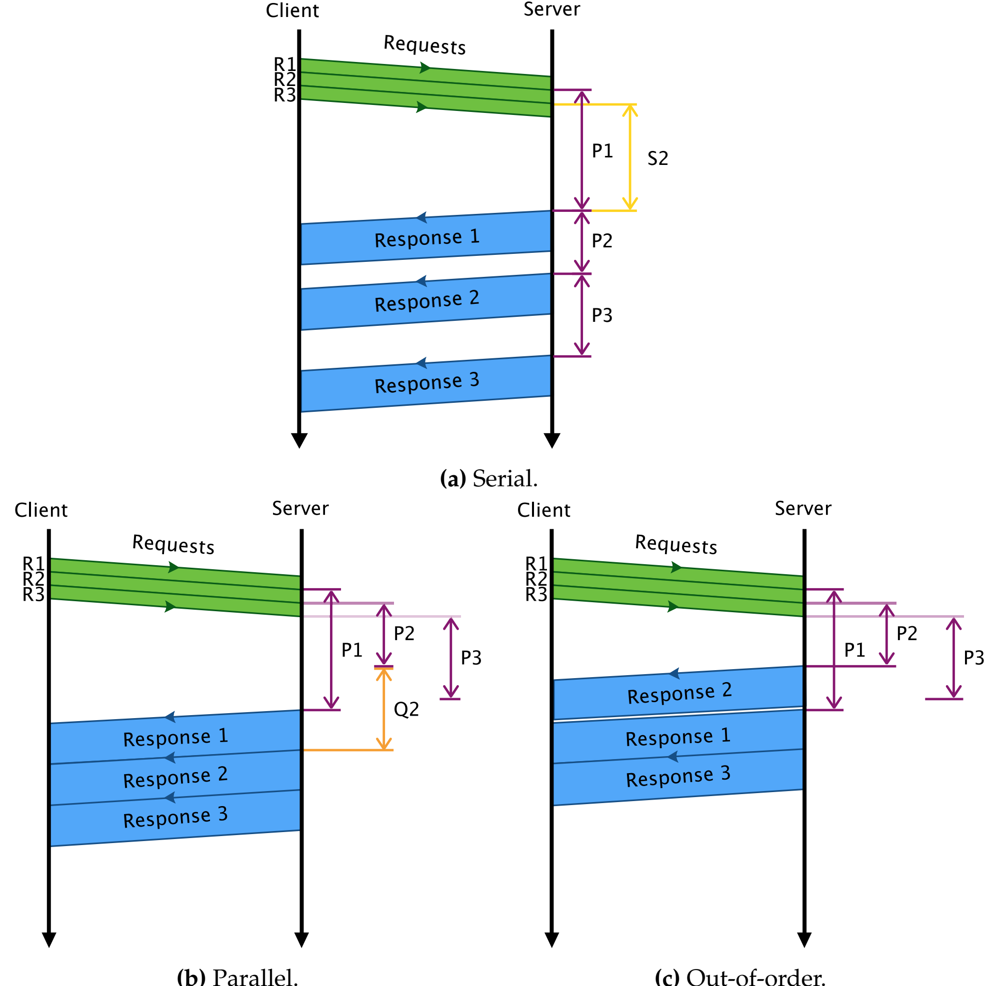
Multi-process or multi-threaded Traditionally, multi-processing HTTP server architectures have used separate processes to serve incoming connections. A master process listens on port 80 for incoming connections, and when a new connection is established, calls *fork()* to spawn a new worker process that will handle this connection. But, since forking a new process every time a connection arrives is rather time consuming, a server with this design usually requires the master process to maintain a pool of pre-forked worker processes. On receiving a new request, the master selects an idle worker from the worker pool, which in turn serves the request.

Modern server architectures use threads to serve requests, rather than distinct processes, since they are more lightweight and consume less memory. A main thread listens for incoming connections, and is responsible for launching worker threads to serve these connections. Once again, a pool of idle worker threads stand ready to serve any incoming HTTP requests, saving the extra overhead of spawning a new thread upon the arrival of each new request.

Finally, hybrid multi-process and multi-threaded architectures combine the scalability of using multiple threads to serve a large number of concurrent requests whilst consuming minimal system resources with the stability of a process-based server. Servers with this architecture have a master process which maintains a pool of child processes, each with its own set of threads. Each process has a listening thread, which listens for incoming connections and assigns these connections to an available idle worker thread. Apache web servers [116, 117] employ these multi-process/multi-threaded architectures, and have separate pluggable modules that implement the three configurations above.

Event-based Instead of creating several threads, where each thread responds to a particular connection, an event based server recognises that these threads spend most of their time blocked on I/O or waiting for new requests to arrive from the client. Event-based designs attempt to maximise the amount of time a thread spends doing useful work by creating a fixed number of single-threaded worker processes in a pool (usually equivalent to the number of CPU's on the machine). A worker process accepts new requests from a shared "listen" socket. Each worker maintains its own runloop which processes events from an event queue. Therefore, when a connection is initiated, an event is triggered on the listen socket and processed via the runloop, which in turn creates a connection socket. A connection socket listens for further events from the client, *e.g.*the arrival of the next request on a Keep-Alive connection. Connections are added to the runloop and processed asynchronously until completion. In other words, by using non-blocking I/O, a connection may generate events in the runloop which are handled using callbacks, timers, and event notifications. NGINX servers [118, 119] and Apache's

mod_http2 [107] use this event-based design, where each worker can handle hundreds of thousands of concurrent connections.



(a) Serial.

(b) Parallel.

(c) Out-of-order.

Figure 3.15: Server models in PCP, demonstrating what will happen on receiving three pipelined requests R1, R2 and R3. The server may apply the processing times corresponding to each request (P1, P2 and P3) serially, in parallel, or out-of-order.

Understanding how real servers operate allows PCP to model how they may process different incoming HTTP requests. The model is straightforward for the basic request/response semantics of HTTP/1.0 and 1.1, but becomes more nuanced when we consider pipelined HTTP/1.1 and HTTP/2, where a server must handle several successive requests on one connection.

Suppose a server receives three pipelined HTTP requests on a particular connection, as shown in Figure 3.15. Given the server architectures described above, a server can process and respond to incoming requests using one of the three mechanisms described below:

- **Serially**: In the case of servers which employ multi-threaded architectures (*e.g.*Apache), a single thread handles all incoming requests on a particular connection, and can only process each each of them in order, as shown in Figure 3.15(a). Because the processing is sequential, the server cannot begin processing request R2 immediately upon receiving

it, but must rather wait a serialisation delay S2 before it can handle the request, since it must complete processing the previous request R1. Serialisation delays are additive and detrimentally affect a web page's completion time. It therefore comes as no surprise that browsers commonly disable pipelining: they are trying to avoid encountering large serialisation delays exhibited by this typical type of server.

- **In Parallel**: Servers with event-based architectures do not bind a single thread to a particular connection, but rather allow the processing of several incoming requests to occur in parallel, as seen in Figure 3.15(b). In the absence of clear framing, the semantics of HTTP/1.1 dictate that the responses to particular requests must arrive in the order the requests were sent (*i.e.* FIFO order), to allow browsers to correctly pair requests and responses together. Consequently, slow responses actually impose head-of-line blocking on responses which have shorter processing times, as is the case for R1 blocking R2. Request R2 experiences a response queuing delay Q2, where it must wait for the previous response to be transmitted before it before it can be transmitted itself. For this type of scenario, the server would also need to maintain a queue of completed responses in memory before they can be transmitted (the second and third responses in this case).

- **Out-of-order**: In essence a type of parallel server, this server supports HTTP/2 (or any protocol with a framing mechanism) and therefore allows responses to be transmitted back to the client out of order (Figure 3.15(c)). Since an HTTP/2 stream is marked with a corresponding identifier, the server can process requests in parallel, and send the response to R2 back before R1, therefore eliminating any inherent head-of-line blocking.

PCP incorporates all three of these server architectures in its simulation framework, allowing us to fully examine the impact of server design on a web page's completion time. The Emulator uses server processing delays collected from the Profiler, as described in Section 3.4.1.

3.7.7 Network Model

We define the properties of the underlying network for delivering web requests and responses in this section. Although TCP CUBIC is the de facto protocol used by many operating systems today, we use TCP New Reno as transport in our experiments. In later chapters, we compare single-path TCP's page load performance with MPTCP, and therefore must use a comparable congestion control algorithm. Unfortunately, a thoroughly examined load-balancing CUBIC protocol for MPTCP does not exist. Additionally, TCP CUBIC generally does not achieve a large performance improvement over New Reno for web downloads; it works best for large transfers over high RTTs. Web downloads, on the other hand, are short flows that normally operate with smaller congestion windows. A comparison of between TCP Reno and TCP CUBIC downloading web pages using HTTP and SPDY as performed by Erman *et al.* [56]. It shows little difference in average PLT across the two TCP variants. More importantly, the choice of TCP protocol had no effect on the relative ordering between SPDY and HTTP with respect to PLT.

Another possible transport which could improve PLT is BBR, but BBR has emerged towards the conclusion of this work. We do not expect it to change the relative ranking of web protocols and or their behaviours. However, BBR is better than CUBIC or New Reno at maintaining short queues, so should reduce latency for flows that are large enough to leave slow-start. This feature will particularly help when a high priority object from one connection competes with a low priority object on another connection.

Finally, we present results of downloading a web page without any competing background traffic. We performed preliminary experiments using a long lived background flow representing a large file download. As expected, protocols that are more aggressive and initiate multiple connections per domain like HTTP/1.1 received a larger proportion of the bottleneck bandwidth, resulting in faster page loads. Unfortunately, this is a selfish choice that starves the background flow and is consequently not the right solution to the problem. If background traffic is added to the mix, it is difficult to arrive at meaningful conclusions with respect the web download itself, especially since web downloads create multiple flows that compete with each other, resulting in self congestion. Additionally, we'd have to examine all background traffic types to understand their impact.

Ultimately, downloading a web page with minimal background traffic is a common use case for cellular networks. Celluar networks usually use dedicated per-host queues, indicating that hosts have little effect on each other's traffic. Additionally, if a host uses more connections, that doesn't necessarily increase its share of the overall capacity, which implies some measure of isolation between hosts. Of course, the single host itself might be the source of the background traffic, for example by browsing the web and installing a software update.

We choose not to emulate background traffic for the purposes of simplicity and reproducibility. Nevertheless, it is not uncommon to find use cases in the wild where background traffic has a minimal impact on web page loads.

3.7.7.1 Topology

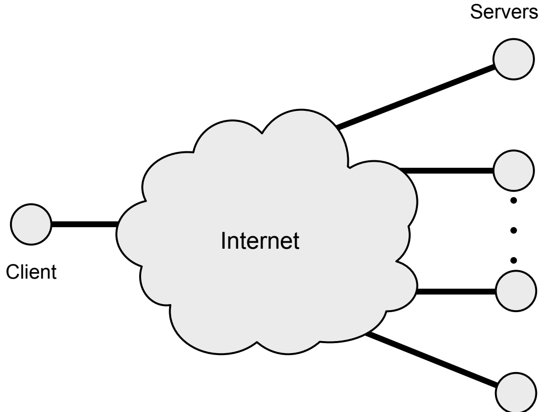


Figure 3.16: Overview of network topology for downloading a web page.

We would like to model the topology displayed in Figure 3.16, where a client connects to

multiple servers over the internet in order to download a web page. The internet backbone is unlikely to suffer from congestion, although delays on the backbone will increase web page load times and must be modelled correctly. Discounting the backbone leaves two other possible locations for congestion: the server tail or the client tail networks. Since we expect servers to be adequately provisioned, the bottleneck will more commonly occur in the client tail. A client typically accesses the web from a home network using WiFi, which indicates the bottleneck will probably be the DSL link, on the return path from the server to the client.

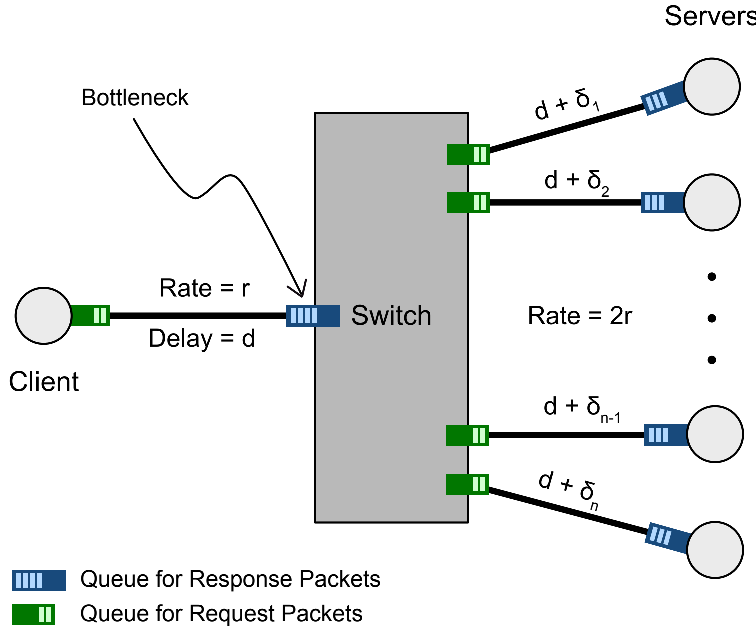


Figure 3.17: Topology for the emulated web downloads.

Taking these factors into consideration, we arrive at the model network topology shown in Figure 3.17 as a reasonable approximation of the topology above. To request and download a particular web site, a single client connects to multiple servers corresponding to each unique domain name the Profiler has recorded in the web site's trace. The client will establish either one or multiple connections to each domain, prompting us to model the correct number of distinct servers with slightly different delays on their incoming links.

In a network of heterogeneous link speeds, the bottleneck determines a connection's maximum data delivery rate. From TCP's viewpoint, a complex path can be accordingly modelled by a single bottleneck link fed by a link of larger capacity, regardless of how many intermediate links exist on the path. Since the client tail network will likely suffer congestion, we model the bottleneck in the switch as shown in the figure. We set the bandwidth on the bottleneck link to be r, and the one way delay to d, and perform experiments where we vary their values to investigate web downloads under different bandwidths and RTTs (where the RTT is approximately $4d$). Each server link capacity is double the bottleneck, in order to keep the bottleneck busy. To account for the variance in results due to transport protocol artefacts and an overly

deterministic simulator environment, we jitter the RTT for each run. We add a random δ to the server link delays, where δ is sampled from the range $\pm 0.1d$.

For the link-layer we use basic point-to-point connections. This is an over-simplification, since link-layer artefacts may influence the behaviour of transport and web protocols running in the layers above. Because we will ultimately examine MPTCP's behaviour for the common use case of downloading a web site on a mobile device, the WiFi and 3G/LTE link-layers are more realistic choices to model. The extent to which our model resembles the real world depends on how predictable the bottleneck bandwidth and the overall RTT are.

For the case of WiFi, the bandwidth between a device and the access point (AP) is unpredictable and time varying, since the bit rate between the client and AP depends on the effective signal to noise ratio (SNR), which in turn depends on path attenuation, interference from other sources, and mobility. The delay is between client and AP is also unpredictable. In light of a poor channel, a transmitted packet can be dropped at the link layer and retransmitted, causing varying delays.

The WiFi link is usually not the bottleneck in typical deployments — the backhaul DSL link is. Although WiFi bit rates may vary, the maximum bit rate of 802.11n is 600Mbps and 802.11ac is around 1Gbps. DSL links on the other hand are optimistically on the order of 20–30Mbps, with fibre to the cabinet typically around 40–80Mbps in the UK [120]. As for delays incurred due to WiFi MAC retransmissions, these will remain on the order of a few milliseconds at most.[3] These short delays will likely be dwarfed by the overall RTT between the client and an upstream server. In short, PCP's simple link-layer will suffice as a model for WiFi in most cases since WiFi is not the single limiting factor that dominates PLT. It is an open question whether BBR interacts benevolently with the WiFi MAC; TCP New Reno generally coexists with WiFi with no negative effects.

In contrast to WiFi, 3G links (and to a lesser extent LTE) will commonly form the bottleneck of a web download. Cellular networks also exhibit time varying bit rates, which may fluctuate due to variations in the wireless channel and mobility, and may be asymmetric. These networks typically use time division multiplexing to assign each user a fixed time slice in which to transmit, and usually cycle through users in a round-robin fashion. Most cellular wireless networks enqueue traffic for each user in its own separate queue. The resulting isolation between users implies that self interference has a more dominant effect on the transport than cross traffic, a feature our model exhibits because it does not include background traffic.

Since the bandwidth may differ significantly (two orders of magnitude) in a short amount of time for cellular networks, overbuffering is common to protect against bursts, causing extra delays and bufferbloat. Winstein *et al.* show an example of how large swings in LTE capacity cause the formation of large standing queues which dramatically increase the delay for

[3] For 802.11n, we arrive at this estimate by recalling that an ACK timeout is around a hundred microseconds, the WiFi contention window has 9μs slots and a maximum size of 1024 slots, and the maximum number of packet delivery attempts is 8.

time sensitive flows [121]. Our model covers cases with these buffering delays (as long as they don't change rapidly) by configuring several experiments to run with high latency. Even overbuffered base stations can experience high losses when a sufficiently provisioned connection suddenly jumps to a high transmission rate. As such, outages on cellular networks are common, are often asymmetric, and will have a large impact on congestion control. Although MPTCP should offer a measurable advantage over single path TCP by shifting traffic to WiFi in cases of outage, we will not examine outages as part of this thesis, since they are out of scope.

Finally, device radios usually spend most of their time in the `IDLE` state in order to conserve energy. The time it takes to promote a device from `IDLE` by powering up the radio is non-trivial, especially for 3G which needs 2s to reach the dedicated channel state (`DHC`) and begin transmitting data [56]. LTE promotion times are better: it takes a radio 400ms to transition to the `CONNECTED` state. A device is demoted if there is no network activity for a certain period of time (3G and LTE go through intermediate lower power states before returning to idle after 12-15s). Browsing habits of users show they pause between navigating between web pages. If a user browses to a web site after a large period of inactivity, its PLT will experience a large one off delay for 3G state promotion (and to a lesser extent LTE). When using multipath with two interfaces, the second interface may not be instantaneously available. In fact, it may be more beneficial to transmit an entire web page over WiFi if it will complete before the time it takes to power up the 3G radio. We do not model state promotion and its impact on PLT, but leave it for future work.

Despite its limitations, our simple topology captures enough real world features to allow us to make useful conclusions about the web and transport protocols we mean to examine. It assumes a bottleneck bandwidth which is stable and does not vary significantly within a short period of time, and no outages. We do not know the border of applicability of our conclusions, and it is unclear to what extent our results will continue to apply. But if we operate in a regime reasonably close to our assumptions, our conclusions should remain valid. In the end, even with the simplest link layer, the effects of using multipath for web downloads are not well understood, and present a good avenue for us to explore. By simplifying the link-layer, we can establish a baseline for the behaviour of transport and web protocols, allowing us to increase the complexity and add more realistic link-layer features in the future.

3.7.7.2 TCP New Reno Parameters

Table 3.3 shows the TCP parameter settings we have configured in the emulator. Most notably, we set the initial congestion window (icwnd) to 4 segments to allow TCP to ramp up more quickly in slow-start. Although modern operating systems like Linux and Windows use an icwnd of 10 segments, we have found this value to be too high since it causes a large amount of drops when operating within lower bandwidth regimes. We observed better performance at low bandwidths using an icwnd of 4, and therefore continue to use this value for our experiments.

Parameter	Value	Description
Minimum RTO	200 ms	The minimum value for the retransmission timeout.
Delayed ACK Timeout	100 ms	How long to wait before returning an ACK when delayed ACKs are enabled.
Delayed ACK Count	2	Number of packets received before an ACK is returned [122].
Connection Timeout	1s if $RTT < 200ms$ or 2s if $RTT \geq 200ms$	TCP retransmission timeout for opening a connection (*i.e.* if SYNs are lost).
Connection retries	20	Number of connection attempts (SYN retransmissions) before returning failure.
Data retries	20	Number of data retransmission attempts before returning failure.
Duplicate ACK Threshold	3	Number of duplicate ACKs required to trigger fast retransmit.
Initial Slow-Start Threshold	unsigned int max	Let the congestion window grow exponentially in slow-start till we hit the first drop.
Transmit Socket Buffer Size	2^{30}	Maximum transmit buffer size.
Receive Socket Buffer Size	2^{30}	Maximum receive window, our simulations should not be receive window limited
icwnd	4 segments	Initial congestion window size.
MSS	1448	Maximum segment size. Computed by assuming an MTU of 1500 minus IP, TCP and point-to-point header sizes and excluding timestamp option size.
Limited transmit	Enabled	Send a new data segment in response to the first two duplicate ACKs for better loss recovery [123].

Table 3.3: TCP New Reno parameters in the ns-3 emulator.

We do not place a cap on the receive buffer size so that our experiments are not receive buffer limited. We want the window available to TCP to reflect the congestion on the network rather than the memory resources available to the client. Most clients will usually have enough buffering to fit small web objects anyway. To accommodate for receive buffer sizes larger than 2^{16}, we enable the Window Scale option [124].

We added an implementation of TCP's PUSH flag to ns-3, to allow a client to send an ACK immediately at the end of a web resource. Otherwise, in the scenarios where a download is application limited, waiting for a delayed ACK time out to fire at the end of objects causes extra delays.

To prevent spurious retransmits, we enable the timestamp option [124], which allows a TCP sender to obtain more accurate estimates of the RTT. As an additional improvement to RTO accuracy, we augmented ns-3 with the following modification. If a retransmission timeout expires, it may have been scheduled in the past using what might have become a stale RTT estimate, especially if the RTT has increased due to buffer growth at the bottleneck. If the most

recent RTT estimate is larger than the one used to compute the expired RTO, we rearm the retransmission timer, so that it fires after the remaining time elapses.

3.7.7.3 Buffer Configuration

The convention for buffer sizing when dealing with TCP flows is to allow for one bandwidth delay product (BDP) worth of buffering on the bottleneck link [69]. Here, the bandwidth is the bottleneck bandwidth and the delay is the RTT. BDP sized buffers work well because they can hold a flight size of bytes if they arrive in a burst, and yet are small enough to minimise bufferbloat. Additionally, if a TCP flow halves its window due to a loss, the buffer still maintains a BDP worth of packets in flight. These fill the pipe to keep link utilisation high while TCP recovers, and keep the ACK clock going.

Which type of buffer should we use for the bottleneck? The simplest is a drop tail queue, which, as its name implies, drops the last packets arriving at the queue if they may cause it to overflow. Drop tail queues represent the most common buffers deployed in real networks. But they can be biased against bursty traffic, and may exhibit global synchronisation with respect to TCP flows. Competing flows may experience drops at the same time on a drop tail queue. In response, they will halve their congestion window jointly, and display oscillating TCP behaviour as a result [125, 126]. These phase effects may be mitigated in the real world due to the randomness of arriving traffic, but in our deterministic simulation environment they will be much more pronounced.

We use Random Early Detection (RED) queues to mitigate phase effects in the simulator, and to inject a measure of randomness into the drops experienced by TCP across multiple runs [127]. RED queues were designed to detect congestion with respect to an average queue size, computed using an exponentially weighted moving average (EWMA). They allow for traffic bursts, but keep the average queue size low. If the smoothed queue size is large for several RTTs, RED queues drop packets as a signal to the transport protocol to trigger congestion avoidance.

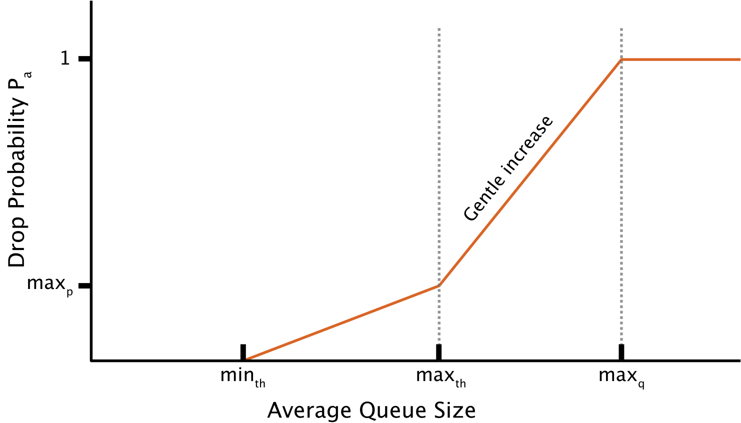


Figure 3.18: RED queue drop probability P_a as a function of the average queue size.

To determine when a packet should be dropped, RED compares the average size with a minimum and maximum size threshold, min_{th} and max_{th} respectively. When the average queue size is less than min_{th}, no packets are dropped. On the other hand, if the average queue size is between min_{th} and max_{th}, then packets are dropped with a probability P_a which is proportional to the queue size, and is less than a fixed maximum probability max_p. Finally, if the average queue size exceeds max_{th}, the RED algorithm can adopt one of two schemes. It can either drop all packets when its size exceeds max_{th}. Or, it can adopt a "gentle" approach, and grow the drop probability linearly till its size reaches the queue maximum size max_q, after which it drops all incoming packets. Figure 3.18 shows how the drop probability P_a changes as the average RED queue size grows.

Our intention isn't to use RED queues to signal incipient congestion or to keep buffer sizes low. Rather, we want our queues to behave like drop tail queues with added randomness. To achieve this goal, we set the maximum queue size to 1.5 BDP, and the RED parameters to the values shown in Table 3.4. We set min_{th} equivalent to 1 BDP to keep the link utilisation high, after which the queue begins to experience drops. We pick max_{th} to be midway between min_{th} and the maximum queue size. Finally, we set w_a, the queue sample weight for EWMA, to one. This setting allows us to use the instantaneous queue size as opposed to the average when determining when to drop a packet, allowing the queue to behave like a drop tail queue.

Parameter	Value	Description
max_q	$1.5\times$ BDP	Maximum queue size.
min_{th}	$0.66 \times max_q$	Minimum average queue size threshold.
max_{th}	$0.8 \times max_q$	Maximum average queue size threshold.
max_p	10%	Maximum loss probability.
w_a	1	Weight given to the current queue sample when computing the average queue size using EWMA.
Gentle	True	Set to true to increase dropping probability slowly when ave queue exceeds max_{th}.
Wait	False	Set to true to wait between dropped packets.

Table 3.4: RED queue parameters.

In real deployments, RED queues have proven difficult to configure correctly due to their many parameters, especially if link rates are dynamic. Fair queuing mechanisms like FQ-CoDel [128–130] have emerged as an alternative. Because they have a dedicated queue for each flow whose 4-tuple hashes to the same value, the effects of cross traffic are minimised, and are much smaller than self interference. FQ-CoDel queues aim to minimise the delay a packet encounters, which may reduce queuing delays across the board independently of the web and transport protocols involved. We use RED queues (really drop tail queues with randomness) because they are easier to reason about, and remove an extra level of complexity from a problem which already has too many dimensions.

3.7.7.4 Random Numbers

In our experiments, there are three main sources of randomness: the RED queue drop probabilities, the RTT jitter on each connection from switch to server, and the generation of server processing delays from the empirical CDFs described in Section 3.4.1. One of the benefits of using a simulator for measuring PLT is the reproducibility of the results. Using ns-3's random number generator [131], we picked a fixed seed for all our experiments, but varied the run number for each replication of the same experiment. Fixing the seed allows the simulator to produce the same long sequence of random numbers each time, and the run number chooses disjoint partitions of this sequence. As a result, our experiments are deterministic, and experiments with the same run number use the same substream of random numbers each time.

3.7.7.5 Loss Recovery

When we performed this study, the available ns-3 version used the loss recovery mechanism defined by TCP New Reno [132], without selective ACKs (SACK) enabled [64]. Without the SACK option enabled, TCP New Reno can only recover from one loss per RTT. In consequence, if there are a burst of losses in a single window, TCP New Reno may take a relatively long time to fully recover. We demonstrate this behaviour in Appendix B.

In contrast, the SACK option allows the sender to resolve ambiguities regarding which sequence numbers have been lost much faster, facilitating loss recovery without resorting to New Reno's one retransmission per RTT. Real world TCP implementations use some variant of SACK, for example in Linux and Windows. SACK would improve TCP's performance by decreasing the time spent in loss episodes, and would consequently reduce the large variance in PLT when random bursts of losses occur across different runs in our experiments. Faster loss recovery is kinder to protocols like HTTP/2, which only use one connection per domain and thus suffer from head-of-line blocking until all lost packets are retransmitted.

Our simulations do not suffer from packet reordering or ACK loss. By virtue of our chosen topology, all packets will arrive at their destination via the same path, and will therefore arrive in order. As for ACK loss, the buffers on the client-to-server path will not drop ACKs because there is no congestion on this path. On the other hand, ACK loss may possibly occur on the return path from the server to the client, for ACKs in response to HTTP requests. Because ACKs are small, and because requests are sparse in frequency, it is highly unlikely that these ACKs will experience any drops. In fact, we did not observe any ACK drops in our collection of experiments.

Because our simulator doesn't reorder packets or drop ACKs, we implemented a lightweight version of SACK in ns-3 which behaves similarly with respect to loss recovery. As opposed to acknowledging received sequences, the receiver instead notifies the sender when it determines a sequence has been lost using negative acknowledgements, or NACKs. The details of this scheme can be found in Appendix B. In the absence of packet reordering and ACK loss, we can use NACKs to reproduce SACK's behaviour in our simulator, and allow

flows to recover from losses much faster.

3.7.7.6 Congestion Window Validation

By virtue of the inherent dependencies between web resources, and the delays incurred for computation, parsing and server processing, web traffic may experience several periods of idleness during a download. When interspersed HTTP requests arrive at the server, it cannot respond with a continuous stream of data but rather with bursts of traffic corresponding to each individual response, which can often be smaller than a congestion window. As such, the TCP flows are not network limited, where the congestion window limits the send rate. Instead, they are application limited, *i.e.* the sender transmits less data than is allowed by the congestion or receiver windows.

Flows punctuated with idle periods and application limited episodes pose a problem for TCP, since they may cause the invalidation of a flow's congestion window. During these periods, the window no longer reflects the current state of congestion in the network. If the flow has been idle for a certain duration, then it hasn't obtained any feedback about the network in the form of returning ACKs. In the absence of feedback, the network could have become more congested since the last transmission without the sender ever knowing. If the flow is application limited, the congestion window may have increased even if the previous window had not been fully utilised.

If a TCP flow has been idle or application limited for some time, it encounters problems when it must send a new resource arriving from the application layer. With an invalid congestion window that doesn't accurately express the state of congestion in the network, TCP will attempt to send a large window of packets only to encounter large losses, causing an increase in page completion times.

To avoid unnecessary losses, and improve TCP performance for application limited web traffic, we implemented congestion window validation in ns-3 as defined in [133]. The basic idea behind congestion window validation is to decay the congestion window proportional to the duration of the idle period. Additionally, it prevents the TCP sender from increasing its congestion window when it is application limited. Implementation details for congestion window validation can be found in Appendix C.

Enabling congestion window validation helps improve PLT when the bottleneck bandwidth is low, and we expect losses on the path. As for higher bandwidths, congestion window validation can limit performance, and prevent a link with a high capacity from sending its full window when there are no losses. The benefit of congestion window validation for high loss regimes is large enough in terms of PLT that we accept the penalty of under-utilisation for higher bandwidths. Since halving the cwnd for each idle RTO worked well for lossy regimes, but is too conservative when we have enough capacity, a better scheme is to use the updated validation mechanisms defined in [134]. Linux implements congestion window validation according to the original RFC [135,136]; other operating systems are even more aggressive, opting

to reduce the window to icwnd after an idle period. As such, our congestion window validation algorithm reflects real deployments, and will sufficiently prevent losses in light of invalid congestion windows, thereby improving PLT.

3.7.7.7 Prioritisation

Since resource prioritisation will require a server to re-order HTTP responses, prioritisation only makes sense for servers with an out-of-order architecture using either HTTP/2 or QUIC. After experiencing the correct server processing delays, if several responses are ready to transmit, the server will deliver them to the client in order of decreasing priority. To achieve this ordering, we replaced TCP's regular socket transmit buffer with a priority queue, which orders packets based on their resource's priority. If TCP has already begun transmitting packets from particular a resource, there are two distinct ways our emulator can deal with enqueuing a higher priority resource at the transmit buffer in that instant. In the first case, the higher priority resource will not preempt the resource currently in progress, but will rather wait until the resource completes before it gets its turn. As for the second case, the higher priority resource will indeed preempt the remaining untransmitted bytes of the resource in progress, pushing them further back in the queue.

We are not suggesting a change in TCP to support this feature, but prioritisation in the socket layer (if we're not using UDP like QUIC) achieves the best results and is easy to implement in the emulator. A real server, on the other hand, may instead choose to keep resources in an application layer priority queue, and write resources to the socket in chunks. In fact, the H2O server implements prioritisation in this fashion: it uses a scheduler to decide which HTTP response should be sent to the client, per 16KB chunk [110].

We do not prioritise HTTP requests in the same way on the client side. We discovered that doing so actually increases the PLT, because the client has no knowledge of the server delays. It may delay requests of lower priority that would have been ready to transmit sooner on the server side. Only the server can decide how to apply priorities based on which resources are available to transmit, once they have been subject to their processing delays.

3.7.8 Conclusion

In this chapter, we presented PCP, a trace-driven framework for modelling and emulating web page downloads. We defined the most common web page activities and their relationships, in order to build dependency graphs that allow us to simulate popular web pages. Additionally, we've stated our assumptions and design choices behind the browser, server and network models we use for our simulations. The server model incorporates processing delays to approximate server behaviour more accurately. Simiarly, the web page dependency graphs incorporate JavaScript timers and events into the page load process.

We may have a potential bias towards home and login pages of popular domains (*e.g.* facebook.com), as opposed to web pages users may navigate to within these domains. Although we have profiled 300 web sites, and captured a good enough spread to cover many types of

web pages, in the future it may be prudent to examine traces from web sites that are not landing pages as well.

Chapter 4

Web Baseline Evaluation

4.1 Introduction

In the previous chapter, we presented PCP, an emulation framework which simulates the download of web pages using real traces and measures their page load performance under a varied set of network conditions and web protocols. It is a tool that will allow us to investigate the benefit of using multiple paths for web downloads, and evaluate possible design solutions to combat any limitations that may arise. But before we embark on an in depth analysis of web downloads over multiple paths, we need to understand some interim questions with regards to web downloads on a single path, thereby establishing a viable baseline. Additionally, we have profiled a substantially large set of web sites, and we'd like to extract better insights into their nature and structure.

At its core, the goal of reducing a web page's load time translates to increasing the parallelism of the page's latency contributing components. Increasing parallelism, in turn, implies minimising the amount of time the network is idle. For example, the page load time would diminish if a browser can perform its parsing and script evaluation activities *at the same time* as downloading the page's critical resources. Similarly, on the server end it is better to parallelise the processing of incoming requests with the transmission of available responses over the network. On the other hand, if the download of a JavaScript file blocks the parsing of the remaining HTML, for example, the network will remain idle while the browser evaluates the script, parses the rest of the page, and issues the next HTTP request. Hopefully, this particular scenario will be avoided with the use of the preloader, which requests the rest of the web page's objects while the main parser is blocked waiting for the JavaScript.

It is not enough to merely keep the network busy. When we only have one path to contend with, maximising network utilisation is key:[1] resources must be transmitted as quickly as possible using the network's available capacity. If connections remain in slow-start, for example, the network will remain underutilised, delaying the transmission of each resource and leaving much room for improvement. In the end, the more we coalesce several small resources so

[1]The issue becomes more complicated when multiple paths with heterogeneous bandwidths and RTTs are used. We discuss these in more detail in Chapter 6.

that they appear like one large download (that correctly consumes all of the network's offered capacity), the smaller the PLT.

Often, the factors that facilitate parallelism interact unexpectedly and counter to their original intention. For example, in the absence of pipelining, HTTP/1.1 achieves parallelisation at the server by using multiple connections per domain and web page sharding. These mechanisms are workarounds for HTTP/1.1's FIFO semantics, aimed at preventing the network from falling idle for too long. Unfortunately, increasing the number of connections causes self-congestion at lower bandwidths, thereby increasing packet drops and ultimately the PLT.

If we look to more recent web protocols, HTTP/2 keeps the network busy by multiplexing several responses onto one connection and allowing the server to return these responses out of order. In theory, this design should reduce PLT even further, but for cases where connections suffer from high loss, it ends up increasing PLT due to HOL in the TCP stack (Section 4.4.3).

For both these cases, the blame lies squarely on the interplay between components in the different layers of the stack. As a result, we must understand in greater detail the subtle interactions between the application and transport layer, especially with regards to possible web protocols, server architectures, number of connections per domain, bandwidth and RTT. In particular, we want to use PCP to answer the following questions:

- For a particular web protocol, what number of connections per domain should we use in order to minimise PLT?
- As the network speeds up, can we achieve a comparable decrease in PLT?
- How do different web protocols behave under a representative spectrum of bandwidths and RTTs?
- What is the best way to apply resource prioritisation, and does it produce a tangible effect on PLT?
- How do existing server architectures influence PLT?

Although the answers to these questions and the subsequent analysis presented in this chapter are not entirely novel, we focus in particular on leveraging PCP to provide an in depth and detailed investigation of the scenarios in question. It will allow us to the choose the most beneficial set of parameters and protocols moving forward. Furthermore, a deep analysis will uncover crucial lessons about single-path web page downloads, and build the necessary background for our main event: multipath web browsing.

4.2 Web Page Characterisation

The structure of a web page imposes constraints on its interaction with the transport protocol and the underlying network. In later chapters, we will examine in particular the impact of this interaction on page load performance if web pages are unsharded or transmitted over multiple

paths. As a result, before we emulate the PLT of web pages under different network conditions and web protocols, we must first understand some characteristics of typical web sites. We use the traces collected from 300 top Alexa web sites to answer the following questions:

- How accurate are our assumptions about web page sizes, resource sizes and number of objects per web page? Using our collection of web page traces, we can confirm that web pages have tens to hundreds of small resources per page, which motivates the reduction of the number of established connections in modern web protocols in order to minimise the overhead of transferring a large number of small objects. MPTCP must also correctly handle transmitting small web resources on heterogeneous paths which may delay the completion of these resources.

- How spread out are web page resources? Are they concentrated mostly in one domain, or are they distributed across multiple domains? The larger the number of domains, the smaller we expect the benefit of using HTTP/2 or QUIC over their precursors. Web pages may require the download of resources from several different domains for three reasons. First, we expect some measure of content sharding to apply to the origin server, in order to cope with the inefficiencies of HTTP/1.1. Second, web pages often choose to display advertisements in iframes, downloaded from ad servers. Finally, web pages may contain third party content as well, for example analytic JavaScript libraries, or social media widgets. We will examine the amount of sharding web pages typically experience, and the proportion of iframes and ads they download. In addition to reducing the benefit of HTTP/2, establishing multiple connections to many servers may also affect adversely affect MPTCP, since MPTCP suffers from the additional overhead of setting up *two* subflows on each path for each corresponding connection.

- What are the typical delays experienced by web domains? If most of a domain's content is downloaded from caching servers owned by content delivery networks (CDNs) and located next the the client, we expect shorter latencies when compared to content downloaded from a server located further away geographically. We can use the proportion of CDN domains and empirical RTTs from our traces to inform our selection of RTTs to use in our experiments.

- How important are timers and events to the page load process? Are we justified in modelling them in our emulator? Previous work [10,95] considers their contribution to critical path delays minimal and discounts their effect. We use the traces to examine their impact, and also to confirm how much network downloads make up of the critical path.

4.2.1 Resource Counts and Sizes

Let us begin by investigating web site and resource sizes. Figure 4.1(a) shows the CDF of resource sizes in bytes over all the web sites we have traced. The median resource size is about

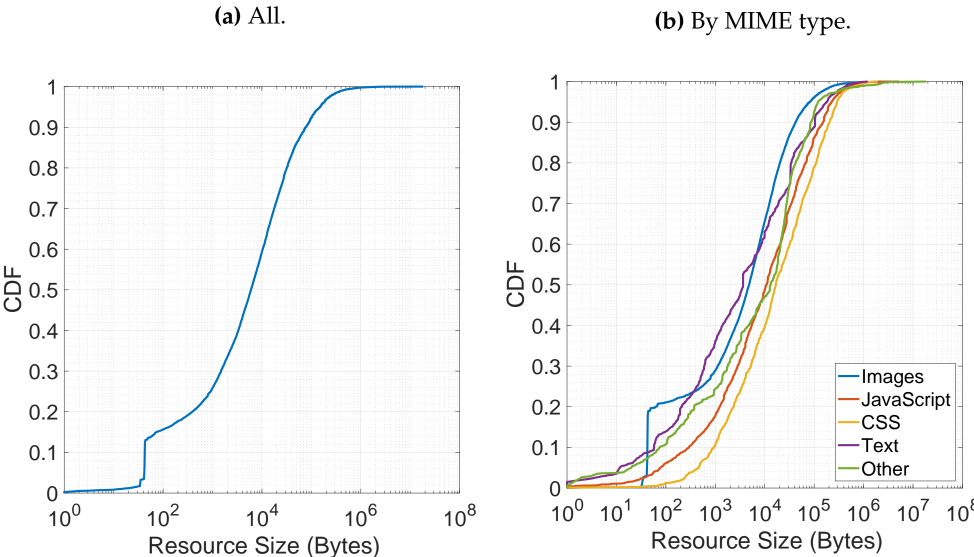


Figure 4.1: CDF of resource sizes across 300 top Alexa web sites.

6KB, and nearly 90% of the resources have a size less than 80 KB. There is an excess at 40-44 bytes, which corresponds to images of size 1×1 pixels, frequently used by web sites as ad tracking pixels, or as placeholders for sprites. The excess is clear in Figure 4.1(b) which divides the resource sizes by MIME type. CSS files represent the largest resources, followed by JavaScript files, images and finally text files. Other types of files, which includes resources like fonts (*e.g.* .woff files), video, flash, JSON files and plugins fall in the middle. While text files mostly correspond to HTML, they can also include XML and plain text files, and they are typically smallest files in the web page.

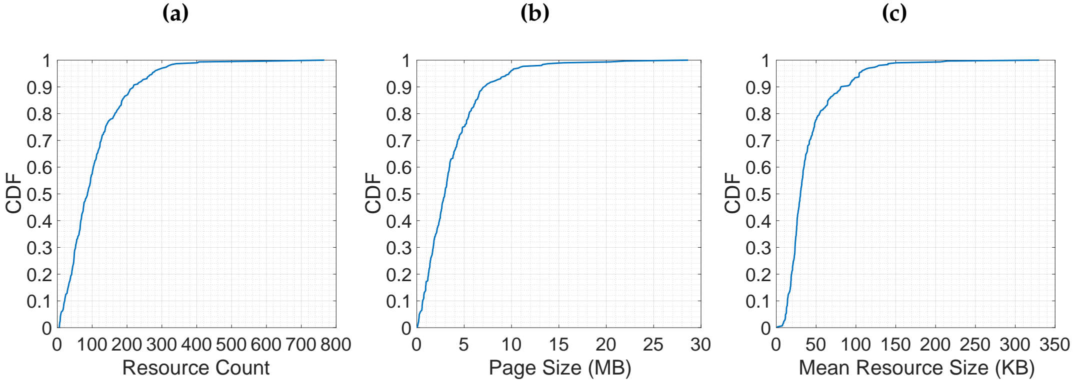


Figure 4.2: CDFs for resource count, page size and average resource size across web sites.

.

Approximately half of the web sites we've profiled contain 100 or less resources a client must download, as seen in Figure 4.2(a), with 90% of the web sites containing 200 or less resources. The remaining 10% of the web sites are large and contain hundreds of resources with a maximum of 766 resources downloaded for the web site www.rakuten.co.jp, a Japanese shop-

ping web site, with hundreds of product images. For each web site, the CDF of the mean resource size is shown in Figure 4.2(c), which indicates that about 95% of sites have an average resource size of 100KB or less. The resource count and average size CDFs suggest that web pages are generally small, an observation confirmed by Figure 4.2(b) which shows that nearly half of our profiled web pages have a total size of 3MB or less, and about 95% of them have a size of 10MB or less. The remaining web sites are outliers, and can reach up to 30MB in size.

4.2.2 Resource Distribution Across Domains

What about the number of domains in a web page? Resources from web sites are distributed across multiple domains, as shown in Figure 4.3(a). The figure reveals that web sites download content from tens of domains, with a median of 20 domains per web site. These domains can correspond to servers for the main page content, third party servers or ad servers. On the opposite ends of the spectrum, a web site can have as little as one domain representing the origin server, or, as is the case of uk.businessinsider.com, a total of 127 domains. A significant proportion of these domains correspond to advertisements, others correspond to third party web sites, and finally, uk.businessinsider.com shards its contents across multiple domains as well.

Given that a web site may download content from as many as a hundred domains, we'd like to know the proportion of bytes downloaded from the largest domain. A web site which allows a client to download most of the resources from one domain are beneficial to protocols like HTTP/2 and QUIC, and allows them to maximise their win over previous HTTP iterations by reducing self-interference and promoting prioritisation. Web sites which spread their resources across a large number of domains, on the other hand, prevent HTTP/2 from attaining a large improvement in PLT. Figure 4.3(b) shows the CDF of the fraction of resources from the total which were downloaded from the largest sized domain across the profiled web sites.[2]

As we can see from the figures, resources are typically spread out across multiple web sites, instead of originating mainly from the largest domain. For approximately half the web pages examined, less than 5% of the total resources and downloaded bytes originated from the largest domain. At most, 20% or less of a web site's resources are downloaded from the largest domain. Sharding web objects across multiple domains is the main culprit behind this trend, and Figure 4.3(c) shows a CDF of the proportion of domains which correspond to shards of the web site, and the proportion of external web domains. Shard domains form a small proportion of the total number of domains in a web page,[3] for example, only six domains in uk.businessinsider.com are actual shards, whereas the rest correspond to ad and third party domains. Nevertheless, Figure 4.3(d) demonstrates that although the are a few shards, and although the largest domain in a web site may contains only a small fraction of the total resources, the collection of a web page's shards as a whole are responsible for delivering most of the web resources.

We expect domain sharding to diminish as more web sites adopt HTTP/2 and implement

[2]We obtain a similar CDF if we instead use the total bytes downloaded from the largest domain.
[3]Chapter 5 describes our method for determining which domains are shards.

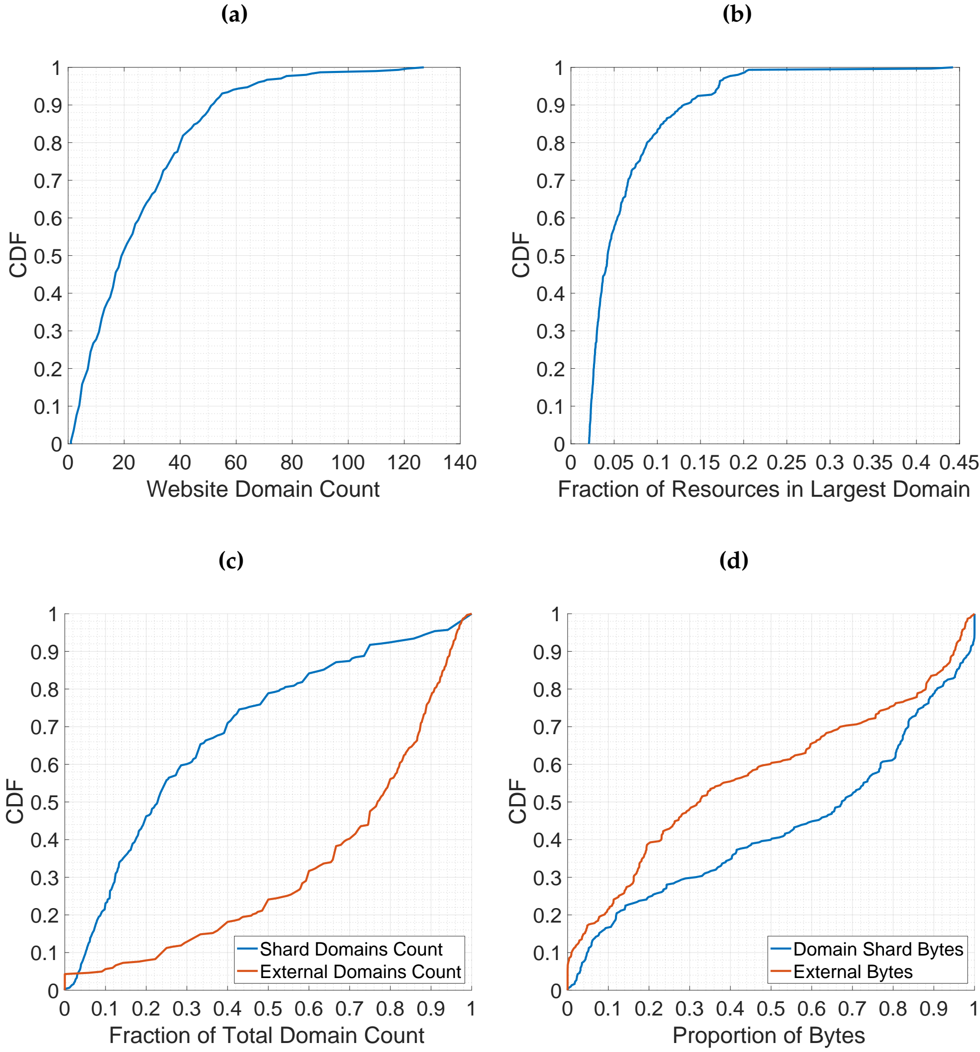


Figure 4.3: CDFs of the number of domains per web site, the proportion of resources downloaded from the largest domain, and the count and size of domain shards. The CDFs are computed across 300 web sites.

its best practices. Unfortunately, unsharding web pages does not remove the large number of external domains shown Figure 4.3(c). To understand the extent of which we can coalesce shards and get a win with HTTP/2, we examine the proliferation of iframes and advertisements in web pages. We used a filter list of domains which is typically used by ad blockers in browsers to mark web resources from ad servers accordingly [137]. We display the fraction of resources for each web page which correspond to ads and those which do not in Figure 4.4(a). Although 30% of the web sites we examined do not contain ad resources at all, ads can form a notable proportion of the resources downloaded by a web page. In the extreme, half of the resources in youtube.com are ads.

Another measure which approximates the number of ads and third party content present in a web page is the number of iframes the page downloads, whose CDF is shown in Figure 4.4(b). The total number of iframes in a web page varies, but half of the web pages ex-

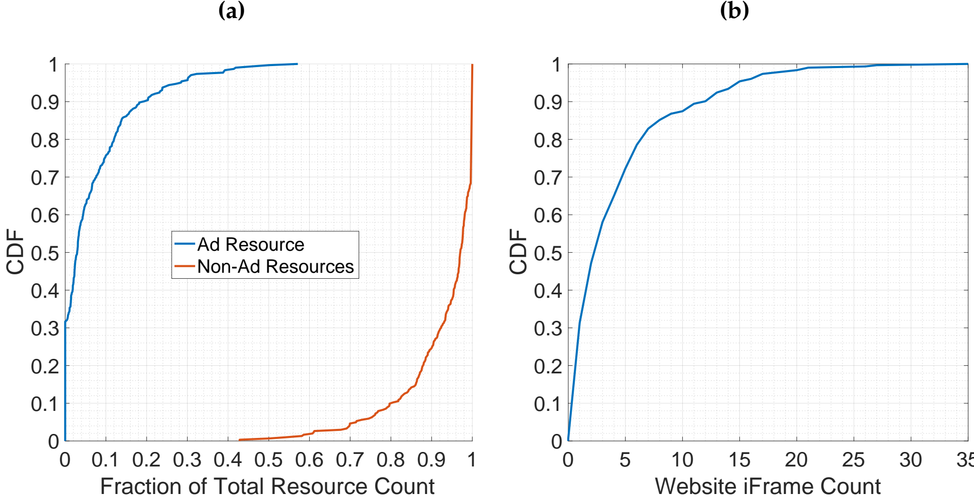


Figure 4.4: CDFs for the proportion of resources which correspond to ads, and the number of iframes per web page.

amined contain two iframes or less. The rest of the web pages can contain up to 35 iframes, which are most likely used to display ads. The presence of iframes and ad domains in a web site indicates that, when optimising a web site to operate with HTTP/2, web developers can only collapse domain shards up to a point. The larger the number of iframes in the web site, the larger the number of domains which persist even after coalescing all of the main domain's shards, which in theory would prevent PLTs from improving too much as shards are eliminated.

4.2.3 Latency to Web Servers

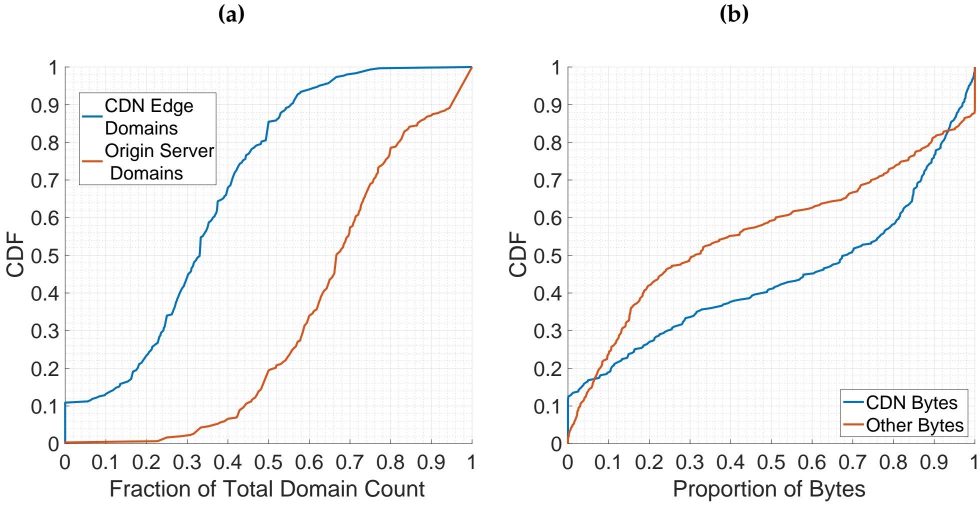


Figure 4.5: CDFs showing the proportion of domains corresponding to CDN edge servers, and the proportion of bytes downloaded from those domains.

Although web pages download content from multiple different servers, web page design-

ers usually attempt to minimise the latency of connecting to these servers by using CDNs to place the requested resources on edge servers located in close geographic proximity to the clients in question. In order to determine the proportion of resources downloaded from CDNs *vs.* those that aren't, we attempt to discern which domains are CDNs by checking their host names or DNS CNAMEs and comparing them with known CDNs, *e.g.* akamai, cloudflare, cloudfront and azure. Additionally we classify any domain which contains "cdn" in its name as a CDN edge server as well. Figure 4.5(a) shows the CDF of the proportion of domains for each web site which correspond to CDN edge servers and those that correspond to origin servers. The figure indicates that a larger proportion of a web site's domains map to origin servers. In the median, only a third or less domains in a web site correspond to CDN edge servers.

Web sites use fewer CDN edge servers than origin servers, nevertheless Figure 4.5(b) shows that most of the web page's bytes are downloaded from these CDNs. In fact, CDNs are responsible for delivering nearly 70% of a web page's content in the median. As a consequence, we expect most resources downloaded from a web page to experience low latencies and RTTs on the order of several milliseconds. Of course, although content may be downloaded from a CDN, the content may have not been pushed yet to the edge server and cached before the browser issues the request. Connection RTTs to CDN edge servers may still be high as a result.

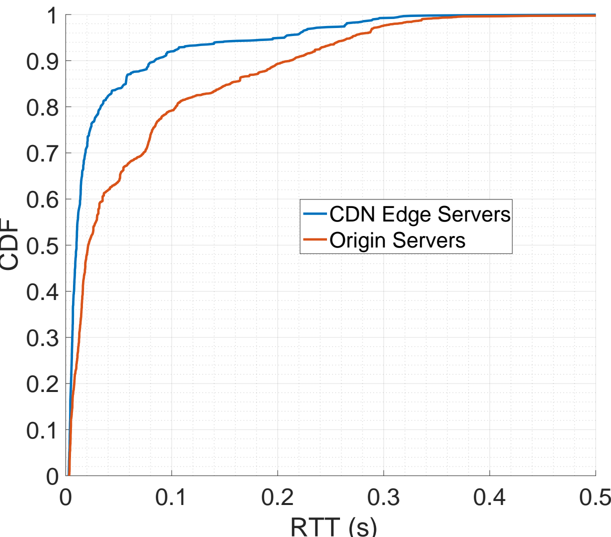


Figure 4.6: CDF of RTTs measured using the Profiler in PCP, when connection to CDN edge servers or origin servers.

To investigate domain RTTs further, we plot a CDF of the empirical RTTs for CDN and origin domains measured using our packet traces, by computing the time difference between the SYN and SYN/ACK when establishing a new connection to that domain. Generally, CDN edge servers have smaller RTTs than regular domains, with 90% of the RTTs when connecting to CDNs falling below about 50ms. Origin server domains, on the other hand, exhibit a larger spread of RTTs, with about 20% of the observed RTTs falling above 100ms. We note that some

connections to CDN domains may have resulted in a cache miss, and required the retrieval of the resource from the origin server, since the connections have long RTTs of 100ms and greater. From these graphs, we conclude that a reasonable selection of RTTs to emulate in our experiments should range from 10ms to 500ms on the upper end.

4.2.4 Resource Dependencies

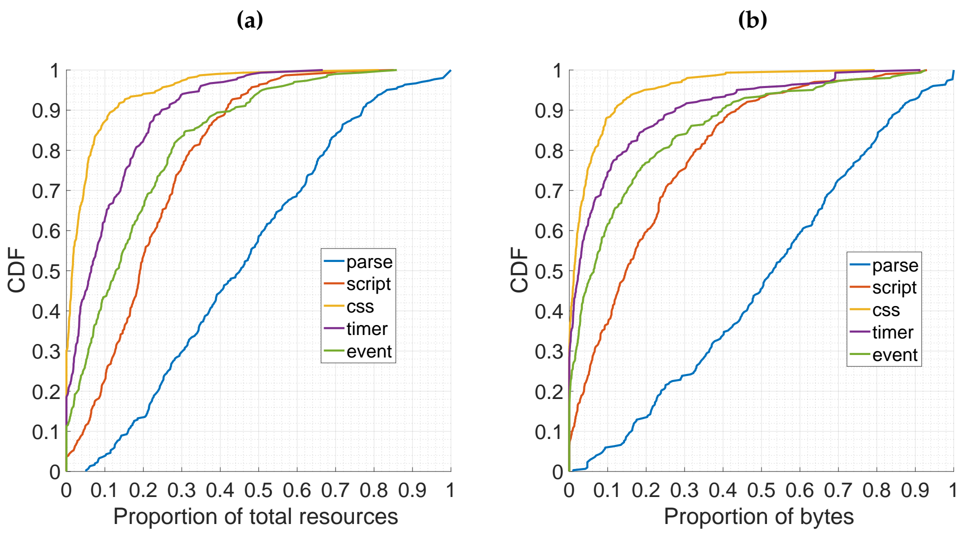


Figure 4.7: Proportion of resource bytes and counts downloaded as a result of completing different dependency graph activities.

Finally, to demonstrate which browser activities lead to the download of the most resources, we plot the proportion of resources and their bytes downloaded from parsing, evaluating scripts, timer and event handlers, and recomputing CSS in Figures 4.7(a) and 4.7(b). The CDFs show the parent type for all resources downloaded by a web page, including those downloaded after the `load` event fires. As expected, the largest proportion of resources are downloaded as a result of parsing HTML tags. Scripts and event handlers are next and are responsible for a significant proportion of resource downloads, and lastly the remaining resources are downloaded from timer handlers, and CSS style recalculations.

How do the various network activities we've encountered contribute to the total time spent on the critical path? Figure 4.8 shows a CDF which plots the proportion of time spent performing the different web activities on the critical path before the `load` event fires. The graph is for illustrative purposes only, since the activity durations we've collected greatly depend on several factors which can vary across different clients, for example the network conditions, the browser type, the client CPU's speed and finally its geographic location. The graph indicates that downloading resources over the network consumes the largest amount of time on the critical path. Computations, whilst consuming a smaller proportion of time, are also critical, with JavaScript evaluations occupying the largest time.

The graph corroborates results shown in Figures 7 and 8 of Wang *et al.*'s WProf work [10], with two discrepancies. First, we note that HTML parsing consumes a minimal proportion of

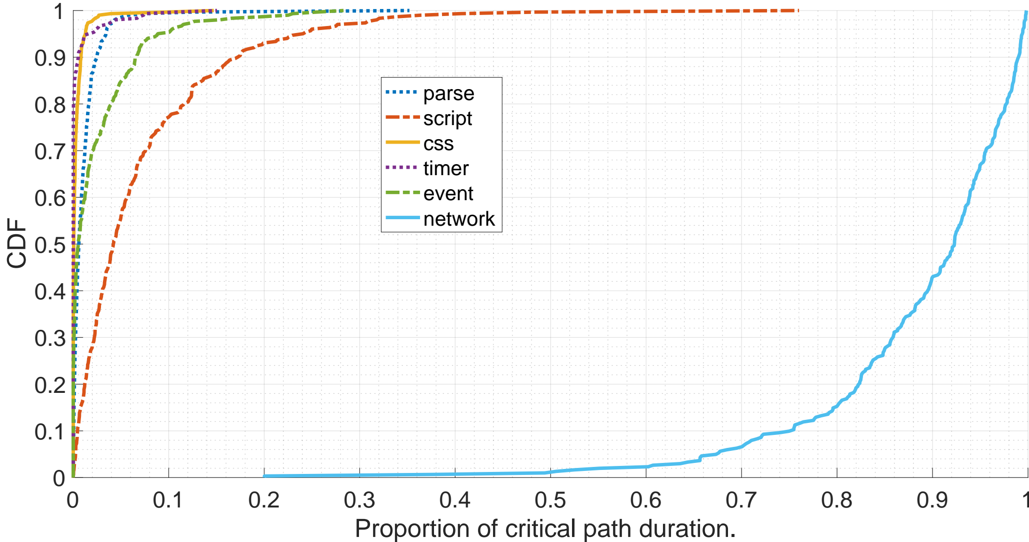


Figure 4.8: CDF of the proportion of time on the critical path spent performing the various network and browser activities.

the time on the critical path in our results, in contrast to Wang *et al.*'s results which indicate that HTML parsing is the most time consuming computation. While the original WProf placed trace points around the start and end of token parsing in the browser, we found these to be incorrectly overestimating the time spent in parsing. After fixing this problem in PCP's Profiler, we arrive at more accurate measurements of time spent in each parse chunk, which is on the order of microseconds. Second, since we have included timer and event handlers in our analysis, and we can see from the graph that their contribution to the page load time cannot be discounted: event handlers are the second most important computation on the critical path.

We end this section with some additional insights. While the graphs we've presented highlight general trends across web pages, they have also exposed another feature of web page downloads: there is enough diversity across web pages to suggest that certain web protocols which target web sites' average behaviour may not work as well for some outliers. For example, resource prioritisation in HTTP/2 may produce a win for most web pages, but fail to make much difference for web pages with resources spread across a hundred domains. Similarly, we must keep in mind that the popular Alexa 500 web sites are a biased sample, since they represent landing and home pages. The graphs we've presented would look different, for example, if we had profiled Google docs or maps.

4.3 Methodology

After understanding the general properties of the web sites we've profiled, we describe in this section the experiments we performed using our emulator. To establish a good understanding of the baseline properties of web downloads, we measured the PLTs of our collection of web sites under a wide range of network conditions. In particular, we focused on the effect of the number of connections per domain, the web protocol, and the server architecture on the com-

pletion time of these pages for a broad spectrum of bandwidths and RTTs. We also measured the effect of different prioritisation schemes on the web protocols that support them.

Because the choice of web protocol informs the server design, these two design parameters are coupled. We used PCP to emulate the following combination of web protocols and server architectures:

- **HTTP/1.0**.
- **HTTP/1.1**, without pipelining.
- **HTTP/1.1 Pipe**, which corresponds to pipelined HTTP/1.1 communicating with servers that support a parallel design.
- **HTTP/1.1 Pipe Seq**, a shorthand for pipelined HTTP/1.1 used with servers which support a serial design, and process requests sequentially.
- **HTTP/2**, with an out-of-order server design. In particular, we implemented the HTTP/2 frames which incorporate the stream identifiers and priorities. We did not implement header compression, since we would like to quantify the effect of stream multiplexing without the influence of other factors. Header compression offers only a modest benefit when compared to gains produced by stream multiplexing in any case [51, 53]. Likewise, we did not implement HTTP/2 flow control, because we assume that the server is unconstrained in memory for our simulations. We ran experiments with different types of prioritisation for HTTP/2 streams: none, MIME-type and critical path. We applied prioritisation to packets in the TCP socket buffer. But, because we approximate the best possible prioritisation in the application layer, we assume that HTTP/2 has no control of resource packets once it writes them to the socket. This fact implies that higher priority resources are not allowed to preempt a resource of lower priority if any of its packets have already been transmitted.
- **QUIC-lite**, which augments HTTP/2 with a QUIC feature, namely out-of-order delivery of packets from the TCP stack to the upper layer. Although 0-RTT handshakes are the most crucial source of PLT improvement in QUIC [21], we did not implement this feature. We focus only on the components of QUIC which were designed to tackle HOL blocking in the TCP stack. We implemented QUIC's out-of-order delivery in our simulator by allowing TCP to deliver segments to the application layer as soon as they arrive. To allow correct in-order reassembly of resources/streams in the application layer, we annotated each TCP segment with its corresponding Stream Frame Header, as described in [28] and Figure 2.5. The application can therefore use the stream identifier, offset and size specified in the header to multiplex different streams, and eliminate delays caused by head-of-line blocking. The use of the Stream Header requires knowledge of web object boundaries at the transport layer. For this reason, our implementation of QUIC-lite is just a mock-up,

intended only to examine the effects of removing HOL blocking in the transport layer. We ran experiments with QUIC-lite using the three aforementioned stream prioritisation schemes, but this time with preemption enabled. Generally, resource preemption would be easy to achieve with QUIC, since it relies on UDP for transport and keeps application level transmission buffers.

Parameter	Values
Rate	500Kbps, 1Mbps, 2.5Mbps, 5Mbps, 10Mbps, 25Mbps, 50Mbps, 100Mbps, 500Mbps, 1Gbps.
RTT	8ms, 40ms, 100ms, 200ms, 400ms.
Connections	1, 2, 4, 6, 8, 10, 12.
Web Protocol/Server Design	HTTP/1.0, HTTP/1.1, HTTP/1.1 Pipelined, HTTP/1.1 Pipelined Sequential Server, HTTP/2, QUIC-lite.
HTTP/2 Prioritisation Scheme	None, MIME-type, critical path.

Table 4.1: Simulated parameters and their values for the web baseline experiments.

The full set of parameters we simulated are shown in Table 4.1. For each combination of parameters, we simulated 50 runs for downloading each web site. We chose a large number of runs to account for the variability due to random packet drops, and to produce statistically significant results.

We chose a wide spectrum of RTT and bandwidth values to cover typical web download use cases. For example, an RTT of 8ms can correspond to responses that arrive from the ISP's cache. On the other hand, web resources downloaded from CDNs will generally experience RTTs on the order of 40ms (see Figure 4.6). A CDN cache miss and corresponding download from the origin server will result in larger RTTs of 100ms. Finally RTTs of 200 – 400ms correspond to possible delays experienced by overbuffered 3G traffic.

As for bandwidth, on the lower end of the scale speeds of 500Kbps – 1Mbps correspond to dial-up. DSL speeds will typically fall in the 2.5 – 25Mbps range, and fibre to the cabinet speeds fall within 50 – 100 Mbps. 3G speeds are on the order of 2 – 10Mbps, with LTE peak rates reaching up to 100Mbps (with around 50ms RTTs). At the higher ends of the scale, fibre to the home can achieve speeds up to 1Gbps.

According to Akamai's "State of the Internet" report from the first quarter of 2017, the global average speed for IPv4 traffic reaching their networks (and not know to originate from mobile networks) was 7.2Mbps [138]. The global average peak connection speeds were 44.6Mbps. As for traffic originating from mobile networks, average speeds ranged from 2.8Mbps – 26.8Mbps across countries. These bandwidth averages fall within the ranges we have emulated.

Although some combinations of bandwidth and RTT can be unrealistic, we emulated the full set of combinations for the purpose of performing a sensitivity analysis. We wish to explore the full space of possible performance behaviours and test the boundaries for how far certain design choices will produce an improvement in PLT.

4.4 Experiments

We present, in the following sections, the results of our baseline experiments which aim to provide a solid background of the state-of-the-art properties of web downloads for the single path. We focus on each simulated web download parameter in order, and provide an in depth discussion of its effect on PLT as we progress.

4.4.1 Connections

Browsers that primarily use HTTP/1.1 as their default web protocol initiate multiple connections to each remote server. Multiple connections parallelise the download of resources in light of HTTP/1.1's FIFO semantics, and diminish the effects of HOL blocking. For example, Chrome creates six connections per domain, a choice its designers have determined works reasonably well after some experimentation.

HTTP/2's design supports pipelining in a more organic manner, and allows it to multiplex several requests and responses on the same connection. As such, the need for establishing multiple connections to improve web download performance diminishes. But even for protocols like HTTP/2 and QUIC, the ideal number of connections to create to each domain remains unclear. Will HTTP/2 benefit from further parallelisation?

In this section, we will examine the effect of increasing the number of connections on the PLT, and the ideal number of connections which produces the lowest PLTs for each protocol as we vary the bandwidth and RTT. We do not include QUIC-lite in this analysis, because it is ultimately an extension of HTTP/2, so any conclusions we can make with respect to HTTP/2 apply to QUIC-lite as well. Our examination will enable us to select a fixed number of connections per domain to use for each protocol, so that moving forward we can fix one of this study's many parameters.

Figure 4.9 shows the best number of connections per domain for each protocol and *(Bandwidth, RTT)* regime. Here, the "best" is the smallest number of connections for each regime that produces the lowest median PLT (within a margin of 20ms) across the 300 web sites $\times$ 50 runs we've emulated. Note that since we do not simulate background traffic, we expect the best number of connections to differ if there is competing traffic. Increasing the number of connections would shrink the background flow's share of the bottleneck, thereby reducing the web download's PLT.

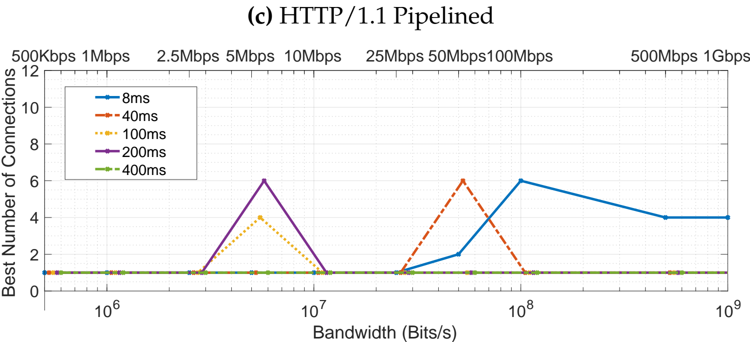
4.4.1.1 FIFO Protocols

We observe a similar trend for HTTP/1.0 and HTTP/1.1 (shown in Figures 4.9(a) and 4.9(b) respectively): web sites downloaded at low bandwidths require fewer connections to achieve the fastest PLTs, whereas the number of connections necessary for low PLTs increases as we increase the bandwidth. In fact, although we emulated a maximum of 12 connections, we expect using a larger number of connections for the higher bandwidths in our spectrum will reduce the PLTs even further. Similarly, increasing the connections' RTTs will also require a larger number of connections to attain the smallest PLTs.

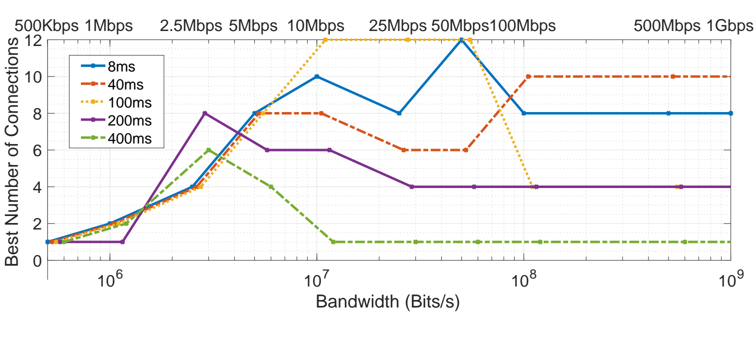
(a) HTTP/1.0

(b) HTTP/1.1

(c) HTTP/1.1 Pipelined



(d) HTTP/1.1 Pipelined Sequential



(e) HTTP/2

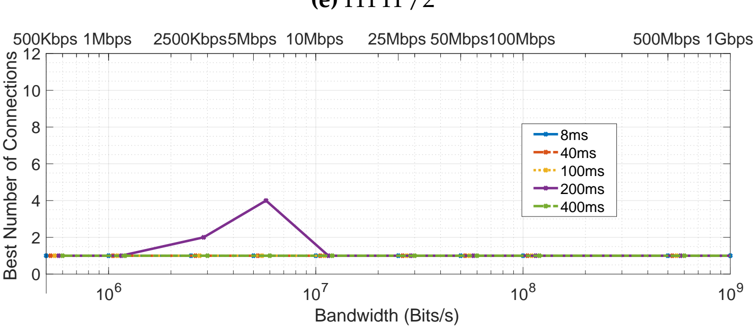


Figure 4.9: The best number of connections with respect to median PLT for each web protocol, given the bandwidth and RTT. Each line on the graph corresponds to the labelled RTT.

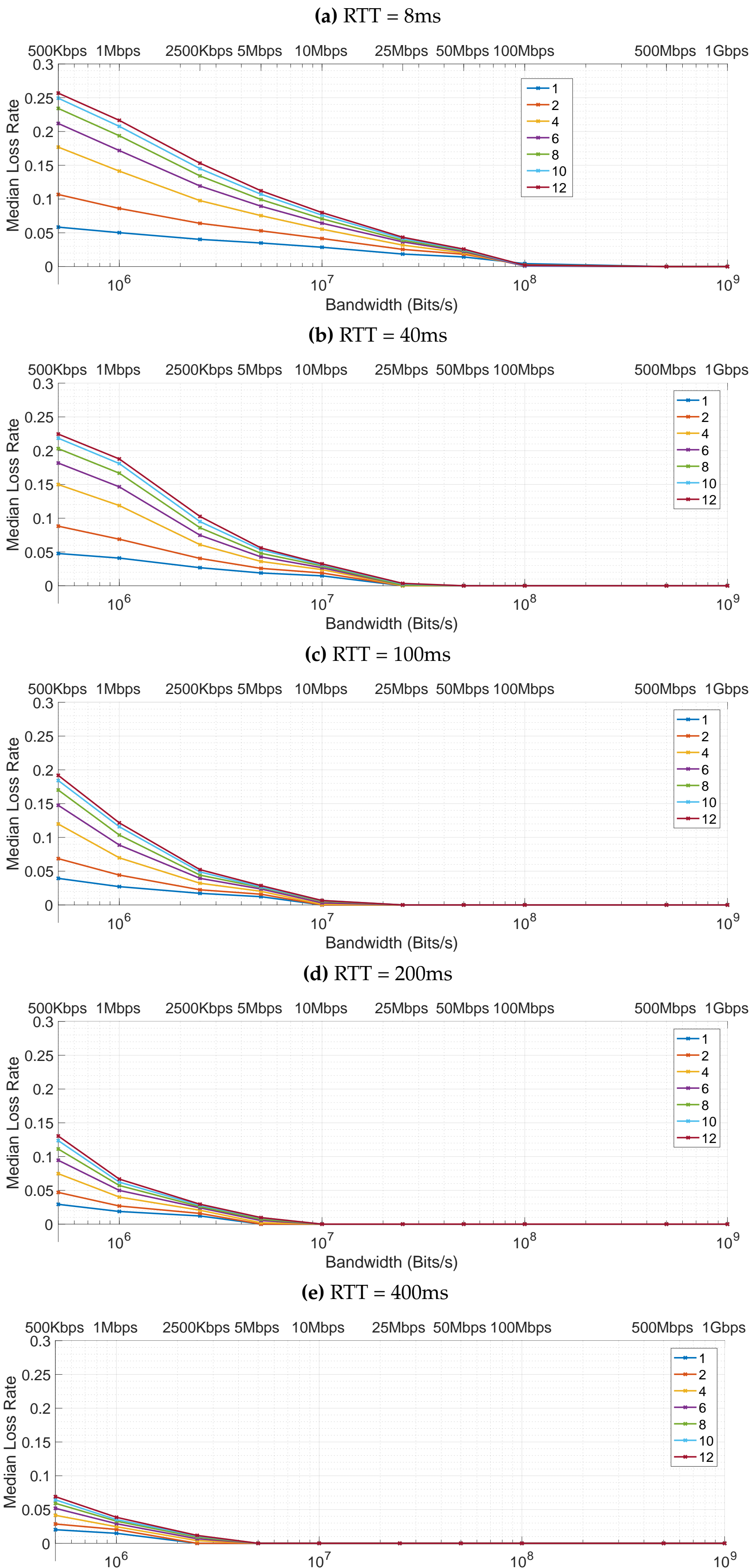


Figure 4.10: The median packet drop rate for HTTP/1.1 under various bandwidth/RTT regimes, and an increasing number of connections shown as different lines in each graph.

To explain these trends, we reference Figure 4.10, which presents the median packet drop rate for HTTP/1.1, and how this drop rate changes as we vary the bandwidth, RTT and number of connections per domain. Because we size our bottleneck queue proportional to the BDP, the figure confirms that the loss rate increases as the BDP decreases, *i.e.* HTTP/1.1 experiences a higher proportion of packet losses at lower bandwidths and RTTs. Additionally, the graphs present a more crucial observation: increasing the number of connections in the high loss regimes will increase the packet drop rate.

Choosing Figure 4.10(c) as an example, we can see that for bandwidths lower than 10Mbps, increasing the number of connections per domain increases the proportion of packet losses on those connections. As the number of connections increases, there is a corresponding increase in the self interference at the bottleneck link where these connections compete with each other for bandwidth, and cause more packet drops. As for bandwidths higher than 10Mbps, there is sufficient bandwidth and buffering to prevent any drops from occurring, as seen from the figure. Here, increasing the number of connections has no effect on the packet drops.

With this information in mind, we can return to our original discussion. In theory, and barring all other transport protocol artefacts, the larger the number of connections we use for HTTP/1.0 or HTTP/1.1, the larger the parallelisation and subsequently the lower the PLT. In reality, this strategy only applies to higher bandwidths and RTTs, in regimes where there are fewer drops. For low bandwidths, it is best to use fewer connections to reduce self interference and decrease the amount of packet drops TCP must recover from, which would increase a web page's completion time.

4.4.1.2 Pipelined Protocols

Moving on to pipelined protocols, we examine pipelined HTTP/1.1 running with a parallel server, and HTTP/2 (without resource prioritisation). Figures 4.9(c) and 4.9(e) demonstrate that using one connection per domain generally produces the lowest PLTs. A little bit of noise exists where using more then one connection appears to perform better, but here the improvement is on the order of 1-2%, and can be considered negligible. In essence, using more than one connection for HTTP/2 (and pipelined HTTP/1.1) defeats the purpose of multiplexing several streams onto one connection. By pipelining the HTTP requests and responses, the client no longer has to queue requests to ensure FIFO order. As a result, connection establishment overheads become more prohibitive. The cost of TCP's 3-way handshake for every newly established connection and slowly growing the congestion window during slow-start delay the web page completions, in contrast to using a single connection which has already expanded its congestion window to match the available bandwidth.

4.4.1.3 Serial Server

Finally, for pipelined HTTP/1.1 using sequential processing at the server, we observe in Figure 4.9(d) a combination of the trends exhibited by HTTP/1.1 and HTTP/2. For low bandwidths, the server serialisation is small compared to the time required to transmit the resources

over the network, such that resource serialisation at the server is not the dominant factor influencing PLT. Instead, the packet drop rate, which increases as the number of connections increase for low BDP regimes, is key and drives us towards favouring a smaller number of connections in order to reduce PLT.

As the bandwidth increases and the number of drops decrease, it becomes more beneficial to use multiple connections in order to increase parallelism in the face of the server serialisation required to maintain HTTP/1.1's FIFO semantics. But for cases where both the bandwidth and RTT are high, the trend we observe resembles that of HTTP/2. For high bandwidth and RTT regimes, the network delay becomes the limiting factor. Although the server serialisation delay is quite substantial, setting up new connections and going through slow-start takes even longer, especially when the alternative is to send packets on an already established connection, which has ramped up its congestion window. Combining the three components together, we arrive at the trends seen in the Figure 4.9(d): the best number of connections increases with bandwidth, only to decrease once again when the bandwidth is large enough.

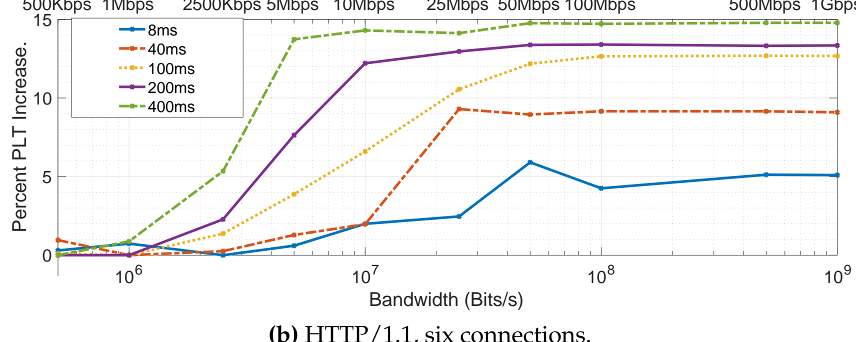
4.4.1.4 Choosing a Fixed Connection Count

We've shown that the ideal number of connections per domain varies with bandwidth and RTT. Realistically, a browser makes the decision with regards to the number of connections at the outset, independent of the network conditions. For the later sections of our analysis of baseline web protocols, we choose to fix the number of connections per domain for each of the protocols we examine, much like a browser. We choose six connections per domain for the protocols HTTP/1.0, HTTP/1.1 and HTTP/1.1 Pipe Seq. Establishing six connections per domain is a robust choice that works well for a reasonable range of bandwidths, in addition to representing the number that Chrome uses. For pipelined HTTP/1.1 and HTTP/2, we use one connection per domain since it is the best number of connections overall.

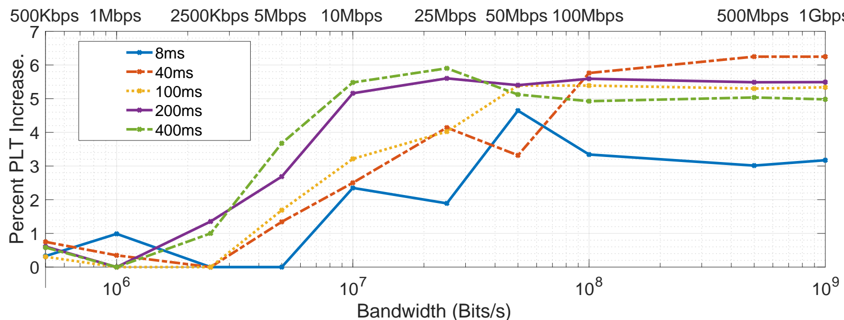
In light of our choices for the number of connections per domain for each protocol, we next measure how suboptimal these choices are and if they represent a good compromise. Figure 4.11 shows the percentage PLT increase if we use the labelled number of connections per domain as opposed to the best number of connections. For HTTP/2 and pipelined HTTP/1.1 (Figures 4.11(e) and 4.11(d) respectively) the increase in PLT is negligible as we expect.

For HTTP/1.0 and HTTP/1.1 (Figures 4.11(a) and 4.11(b)), choosing six connections has very little effect on the PLT at low bandwidths, but causes a decrease in performance at higher bandwidths where using a larger number of connections would have been a better choice. Although HTTP/1.0 may benefit from using, for example, eight connections instead, the decrease in PLT at higher bandwidths exists in tradeoff with the increase in PLT in the lower bandwidths. We argue that an increase in PLT for lower bandwidths is actually worse, since the absolute increase is in the order of several seconds. In consequence, the higher bandwidths can suffer some reduced benefit as a tradeoff for preventing an increase in PLT for the lower bandwidths.

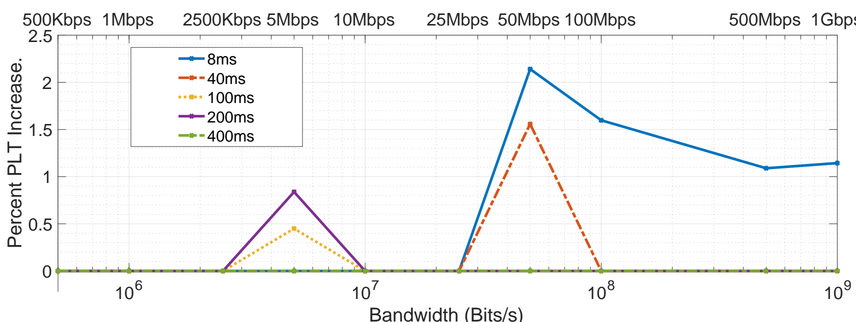
(a) HTTP/1.0, six connections.



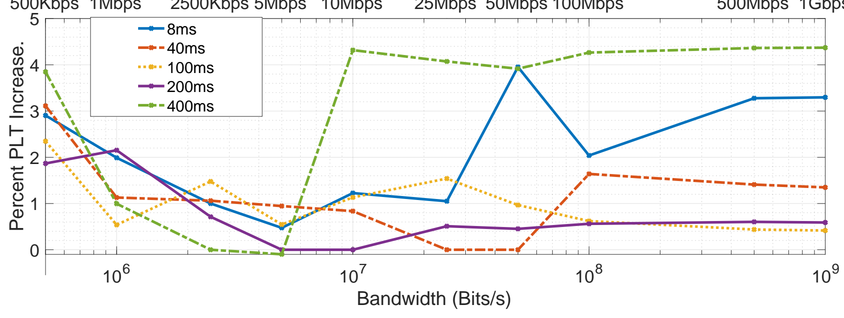
(b) HTTP/1.1, six connections.



(c) HTTP/1.1 Pipelined, one connection.



(d) HTTP/1.1 Pipelined Sequential, six connections.



(e) HTTP/2, one connection.

Figure 4.11: The percentage increase in PLT if we use the chosen number of connections instead of the best. Each line on the graph corresponds to the labelled RTT.

The percentage increase in PLT for pipelined HTTP/1.1 with a serial server (Figure 4.11(d) remains small for most *(Bandwidth, RTT)* regimes with a maximum around 4% in the worst case corresponding to the highest RTT. These are acceptable percentages, which implies that using six connections for this regime does not cause too large of a PLT increase at high RTTs. To conclude, as we continue our analysis we can be confident that our selections for the number of connections per domain are reasonable for the different protocols we wish to examine.

4.4.2 Bandwidth and RTT

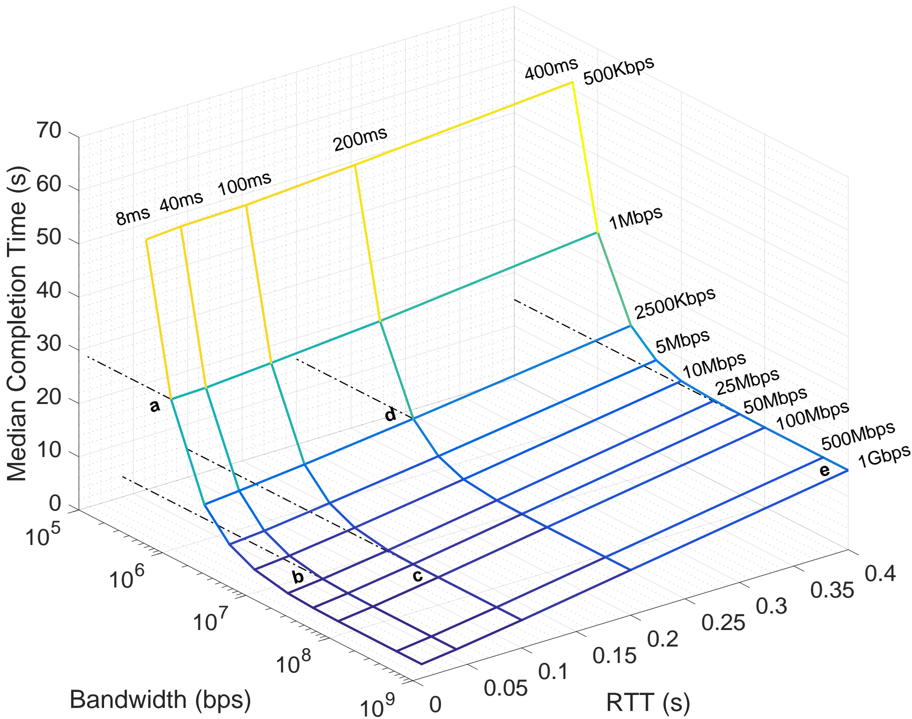


Figure 4.12: Surface plot for the median PLT as a function of the bandwidth and RTT for web pages downloaded using HTTP/1.1 with six connections and TCP New Reno.

So far, we've seen how the number of connections per domain affects the PLT of web pages downloaded using different protocols. In this section, we will answer questions about the effect of the bandwidth and RTT on page completion times. In particular, we'd like to answer the following questions: does the PLT continue to decrease as we increase the bandwidth? Similarly, how large an effect does the RTT have on the PLT across different bandwidths?

To answer these questions, we focus in particular on downloading web pages using HTTP/1.1 with six connections. We plot the median page completion time across the 300 web sites × 50 runs for each *(Bandwidth, RTT)* combination in Figure 4.12 to understand how the PLT behaves as a function of RTT and bandwidth. We label particular points on the graph (a-e) which we will examine in more detail in Section 4.4.3. Note that if we apply the same analysis to other protocols, we will obtain similar surface plots, albeit with different absolute values for

the median PLT.

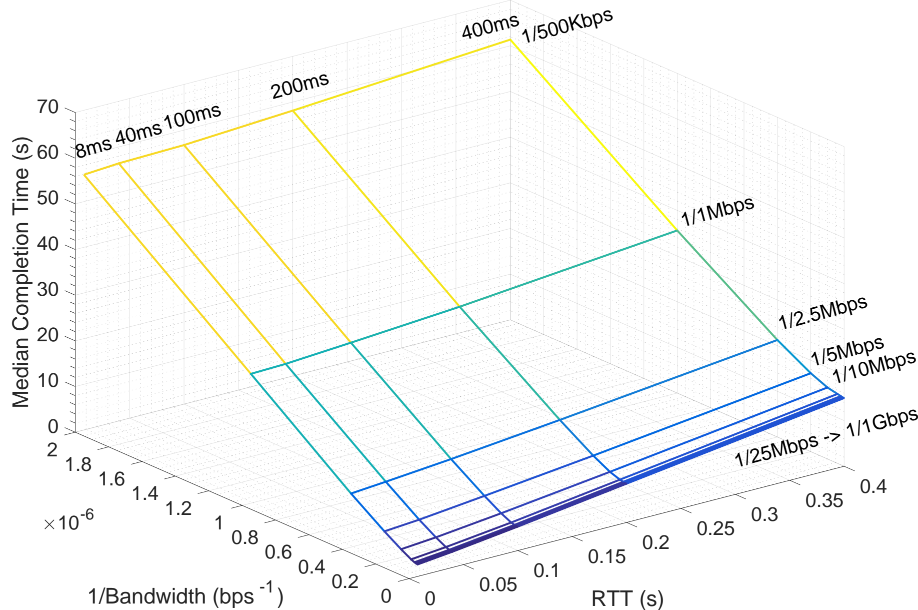


Figure 4.13: Surface plot for the median PLT as a function of the inverse bandwidth and RTT for web pages downloaded using HTTP/1.1 with six connections and TCP New Reno.

The figure confirms that both the RTT and bandwidth components of a web download influence the PLT. Furthermore, it presents several key relationships between these components that we will explain in more detail. First, it comes as no surprise that the median PLT is proportional to the RTT, such that increasing the RTT will result in a corresponding increase in PLT. Although the RTT is as important as the bandwidth most of the time, when the bandwidth is small and is therefore the dominant component affecting PLT, the relative effect of RTT is small. In contrast, when the bandwidth is large enough such that it is no longer the limiting factor, *e.g.* at 1Gbps, we see the RTT play a more crucial role in the magnitude of the PLT.

Second, the bandwidth appears to be inversely proportional to the median PLT. When the bandwidth is small, *e.g.* 500Kbps, the PLT is quite large, between 50 – 60 seconds. A small increase in bandwidth produces a large drop in PLT as we see from the graph. On the other hand, when the bandwidth is quite large, *e.g.* larger than 10Mbps, web downloads are no longer bandwidth limited. In fact, they will probably remain in slow-start and fall short of consuming all of the link's available capacity. Here, increasing the bandwidth has little effect on the PLT, and we say the web download has become latency bound. To confirm the inverse relationship between bandwidth and PLT, we produce a similar surface plot, but this time with 1/bandwidth on the y-axis, in Figure 4.13. The plot shows that the median PLT's relationship with RTT and 1/bandwidth is fairly close to a linear function of both parameters, except that the RTT's contribution differs across low and high bandwidths, as we've highlighted previously.

Lastly, we observe that despite increasing the bandwidth and decreasing the RTT to levels where the network becomes "infinitely" fast (*e.g.* at 1Gbps and 8ms), we cannot shrink the PLT to a value lower than a constant completion time. Even as network delays disappear, the constant PLT corresponds to the sum of the processing delays at the client (the browser's parsing, script execution and CSS style recalculation activities) and at the server, which cannot be eliminated by speeding up the network.

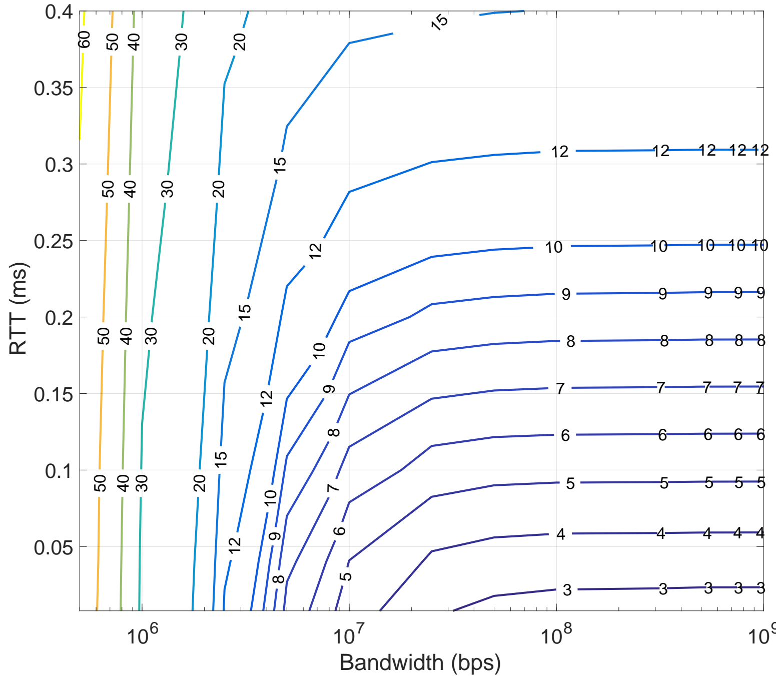


Figure 4.14: Contour plot for the median PLT for HTTP/1.1 with six connections, with respect to bandwidth and RTT.

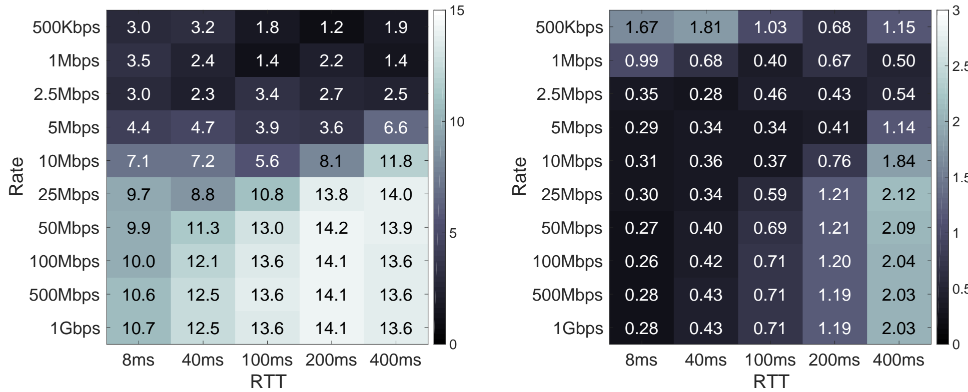
We conclude this section with perhaps a more intuitive method for presenting our data. We plot a contour graph in Figure 4.14 by slicing the surface plot across the lines of equivalent median PLT. The contour plot confirms our earlier observations. Recall that the steepest descent gradient of a surface plot is in fact perpendicular to the lines of the corresponding contour plot. As such, the plot shows two regions of operation. If we are in a low bandwidth regime (*i.e.* the leftmost regions of the graph), the web download is bandwidth limited and we must increase the bandwidth in order to decrease the PLT. Decreasing the RTT has very little effect in this case. Conversely, if we are operating in a high bandwidth regime, increasing the bandwidth has negligible effect on the PLT, as evinced by the horizontal contour lines in the rightmost region of the graph. Instead, the web download is latency bound, and we must move perpendicular to the horizontal contour lines, thereby reducing the RTT in order to obtain a sizeable reduction in PLT.

4.4.3 Web Protocols

Ultimately, before we can examine methods for improving page load times using multiple paths, a solid understanding of how existing web protocols behave for the single path case is essential, with focus on the interaction between the application and transport layers. In this section, we use PCP to compare the completion times of web pages downloaded using each of the six protocols (or more accurately the combination of web protocol and server designs) described in Section 4.3. The web protocols are in increasing order of PLT performance, ranging from the now defunct HTTP/1.0, pipelined and unpipelined forms of HTTP/1.1, HTTP/2, and finally QUIC-lite.

4.4.3.1 HTTP/2 Performance

To set the stage, let's start with a closer look at how HTTP/2 (without any resource prioritisation) performs in comparison with HTTP/1.1 with six connections, and from there springboard into a full discussion concerning the performance of our six chosen protocols. Since HTTP/2 is the standardised replacement for HTTP/1.1, comparing their performance has been a hot topic in recent years, and several past works have examined HTTP/2's behaviour in detail [10, 51–53, 56]. We aim to confirm these results and provide detailed insight into their causes.



(a) Percentage improvement. **(b)** Difference between mean PLTs in seconds.

Figure 4.15: Heatmaps comparing the mean PLT of HTTP/2 with one connection with HTTP/1.1 with six connections. Neither protocol applies resource prioritisation. The percentage improvement is computed using $\frac{\sum PLT_{http1.1} - PLT_{http2}}{\sum PLT_{http1.1}}$.

We begin by presenting the mean PLT improvement HTTP/2 achieves over HTTP/1.1. Figure 4.15(a) shows the percentage reduction in PLT HTTP/2 attains, by computing the ratio of the PLT means in the form of $\frac{\sum PLT_{http1.1} - PLT_{http2}}{\sum PLT_{http1.1}}$. The ratio conveys the average reduction in the overall time spent on page loads across all web sites. On the other hand, Figure 4.15(b) shows the absolute difference of the mean PLT in seconds, *i.e.* $\frac{\sum PLT_{http1.1} - PLT_{http2}}{300}$. The figures confirm the observations from previous work, which we have presented in Chapter 2. In particular, HTTP/2 reduces page load times on average. In the top left hand corner of the heat map, where the bandwidth is low, HTTP/2 reduces the PLT because it reduces self-competition

among multiple connections. The percentage improvement is small because we are comparing relatively large completion times at these low bandwidths. Similarly, the figures confirm that when the connection doesn't suffer any loss, HTTP/2 provides a larger win as the RTTs increase, due to the reduction of slow-start costs when HTTP/2 uses one connection instead of several. We can see this trend moving left to right in the bottom half of the heatmap.

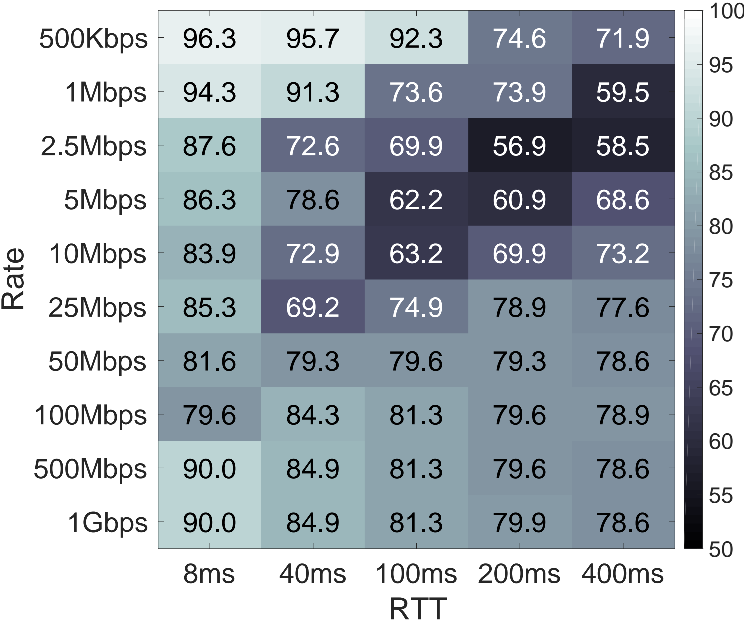


Figure 4.16: Heatmap indicating the percentage of web sites with equal or better mean PLT using HTTP/2 with one connection *vs.* HTTP/1.1 with six connections. Neither protocol applies resource prioritisation.

HTTP/2 can improve performance, but it does not *always* do so. Figure 4.16 gives a broad overview of how HTTP/2 without prioritisation performs compared to HTTP/1.1 for the 300 domains in our corpus. There is one box for each combination of bandwidth and RTT. For each combination, the corresponding box shows the percentage of web sites for which HTTP/2 either improves the mean PLT across the web site's 50 runs or makes no difference.[4] For example, in Figure 4.16, when the bottleneck bandwidth was 1Gbps and the RTT was 8ms, 90% of the 300 web sites we emulated showed the same or better average performance with HTTP/2 than with HTTP/1.1 using six connections per domain.

Although we've shown in Figure 4.15(a) that on average the percentage improvement increases as the RTT increases, more web pages perform better with HTTP/2 when the RTT is low; the heat map shows a definitive decrease in performance as we move from left to right. Without resource prioritisation, objects placed on a single connections by HTTP/2 will experience transport layer serialisation delays that become more pronounced as the RTT increases. But the figure also shows a diagonal band in the top right corner where HTTP/2 performs less reliably. For example, with a 2.5Mbps link to the client and 200ms RTT, only 56.9% of runs perform better with HTTP/2. While this is not a showstopper for HTTP/2 deployment, it does

[4] If the results are within 20ms of each other, we do not regard this as a significant difference.

raise some questions.

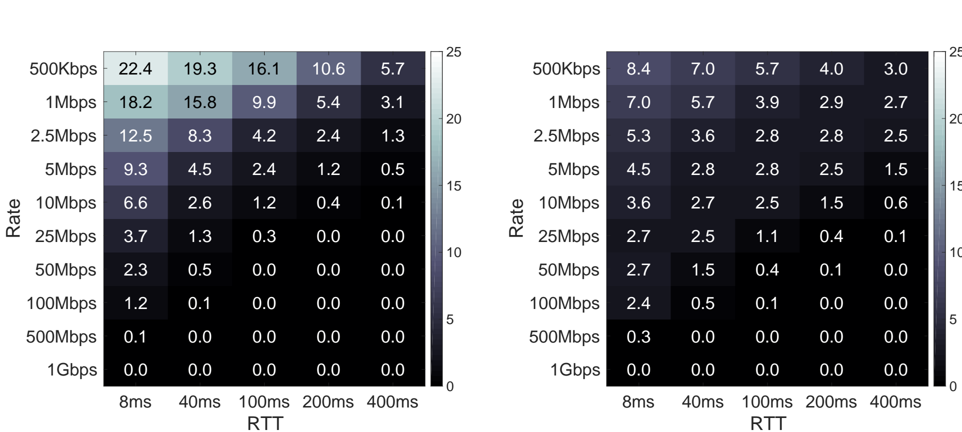


Figure 4.17: Heatmap of mean packet drop rate at different bandwidths and RTTs for HTTP/1.1 and HTTP/2.

Packet losses hurt HTTP/2 . To best explain the cause of the diagonal band, let us examine the mean packet drop rate for each combination of bandwidth and RTT, when downloading the web pages using both HTTP/1.1 with six connections and HTTP/2 with one connection (shown in Figures 4.17(a) and 4.17(b) in turn). Three distinct regions with different loss profiles exist in these figures, and they produce the trends we see in Figure 4.16.

The first region corresponds to the top left hand corner of the heatmaps. Given our discussion about connection count in Section 4.4.1, it is unsurprising that in these areas of low bandwidth and RTT, HTTP/1.1 with six connections sustains a large packet drop rate when compared to HTTP/2 with 1 connection. In these regions, HTTP/2 achieves a lower PLT than HTTP/1.1 for a large proportion of the web sites examined. The second region lies in the bottom right corner of the heatmap, and encompasses high bandwidths and RTTs. In this region where the BDP is large, both HTTP/1.1 and HTTP/2 suffer no losses at all, and once again HTTP/2 outperforms HTTP/1.1 (modulo the effects of high RTT).

The final and perhaps most interesting region corresponds to the diagonal swathe where the loss rates for HTTP/1.1 and HTTP/2 become comparable. In fact, although it may seem counterintuitive, HTTP/2 suffers a *higher* packet loss rate than HTTP/1.1 for some of these *(Bandwidth, RTT)* combinations. For example, the mean packet loss rate is 2.5% for web sites downloaded using HTTP/2 at 5Mbps and 200ms, in contrast to a 1.2% mean loss rate for HTTP/1.1. Figure 4.18, which displays the time/sequence plots for downloading bestbuy.com at 2.5Mbps and 200ms using HTTP/2 and HTTP/1.1 demonstrates the reason for this discrepancy. The chosen BDP is high enough to tolerate a burst of small resources over multiple connections when they coincide, but since the burst duration is short, these connections are safe from drops (Figure 4.18(b)). Instead, drops occur during slow-start when the resources are sent back to back on one connection using HTTP/2, which causes the congestion window

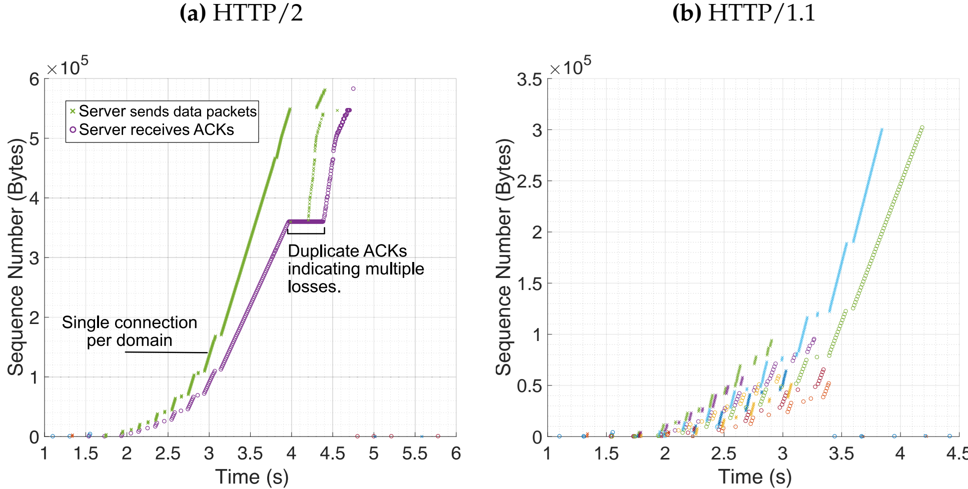


Figure 4.18: Time/sequence plots for downloading bestbuy.com at 2.5Mbps and 200ms.

to ramp up and grow beyond the buffer of the bottleneck link (Figure 4.18(a)). Although the higher loss rate might explain HTTP/2's dip in performance to some extent, it does not capture the entire picture.

When the loss rates of both protocols are nearly equivalent, the head of line blocking HTTP/2 experiences due to TCP serialisation becomes the principal cause for its comparatively lower performance. By placing all of its hopes onto one connection, when HTTP/2 ultimately suffers from losses, other resources will be stalled while the connection retransmits the lost packets. More importantly, when the connection halves its window and applies additive increase, it will remain encumbered by slow window growth even when the original congestion disappears.

Conversely, the web download can hedge its bets when the browser initiates multiple connections per domain for the typical HTTP/1.1 deployment. If any of the connections suffers from losses, only a subset of the resources will be delayed. Instead, the other connections will pick up the slack, and may continue to grow their windows. Fundamentally, using HTTP/1.1 is not the real requisite for improving performance for these scenarios. Rather, increasing the number of connections is key.

Our experiments imply that in the presence of other TCP traffic competing for the bottleneck link, HTTP/1.1 beats HTTP/2 more frequently than Figure 4.16 indicates, as many TCP connections are more aggressive than one when competing with other TCP traffic. In the extreme, this can lead to a tragedy of the commons, and HTTP/2's behaviour is generally preferable, leading to lower loss rates. In many cases, anyway, the last mile link is the bottleneck, and is otherwise uncongested.

A closer look. We present a more concrete example to solidify our understanding of HTTP/2's shortcomings. Figures 4.19(a) and 4.20 show time/sequence plots for downloading buzzhand.com using HTTP/2 and HTTP/1.1 at 2.5Mbps and 200ms, the regime in which HTTP/2 ex-

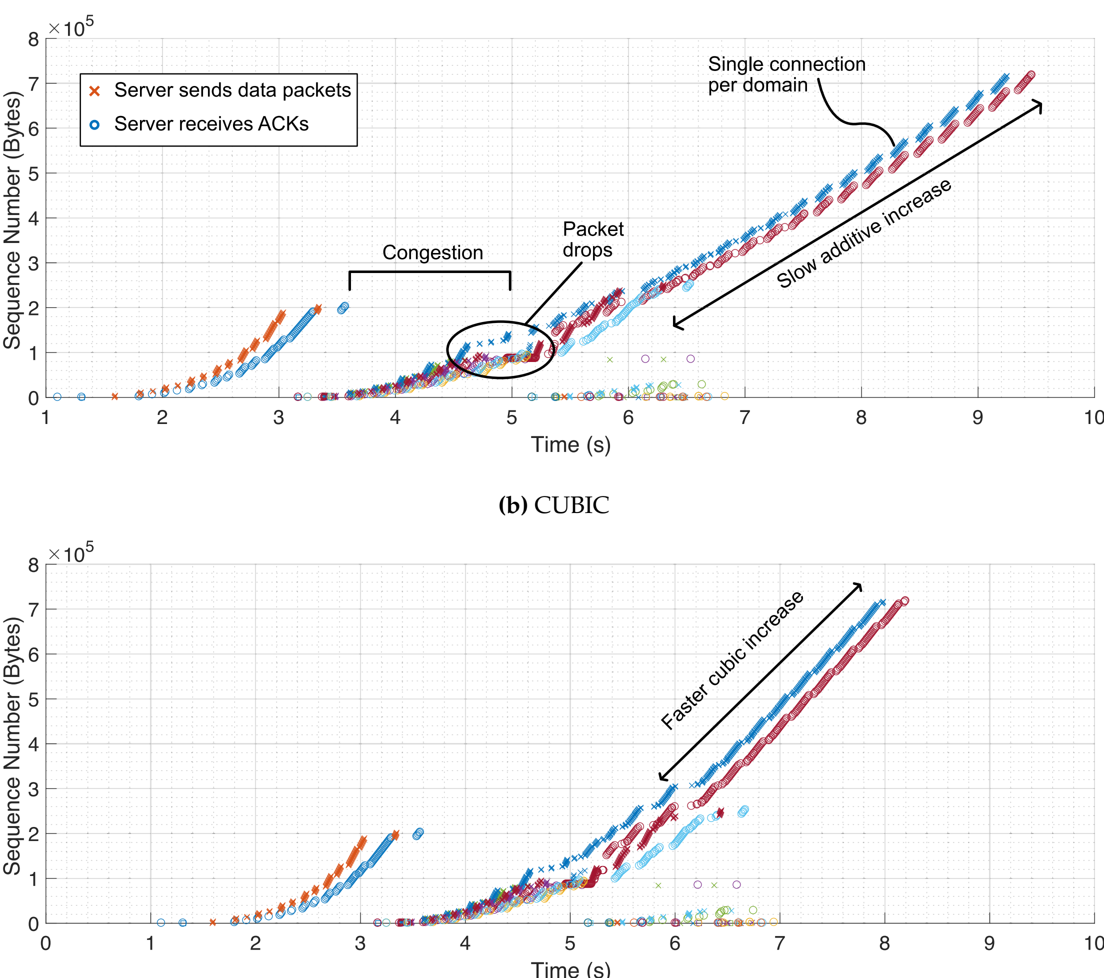


Figure 4.19: Time/sequence plot for buzzhand.com downloaded using HTTP/2 with 1 connection at a bandwidth of 2.5Mbps and RTT of 200ms, using the labelled TCP congestion control.

hibits the worse performance. For this web site, around 56 of the resources originate from the same domain, s2.buzzhand.net, and HTTP/2 creates one corresponding connection to download them all. Figure 4.19 shows that when this connection experiences packet loss due to self-competition with connections to other domains, it continues to grow its window rather slowly using additive increase, even after the other competing connections complete and there is more capacity available. The resulting page completion time is 9.36s.

The time/sequence plot for HTTP/1.1 shows a different picture. Because HTTP/1.1 uses up to six connections to s2.buzzhand.net, it is difficult to untangle these from the single overall plot. Instead, we isolate four of the connections in their own in figures 4.20(b)-4.20(e). On closer examination, we note that although connections 3 and 4 suffer from loss (and a timeout in the case of connection 4), connections 1 and 2 do not, and continue to grow their windows exponentially in slow start.[5] The link utilisation is higher than its HTTP/2 counterpart, and as a result, HTTP/1.1 achieves a faster completion time of 8.51s.

[5]Note that both connections are application limited, which accounts for the pause in the graph for connection 1.

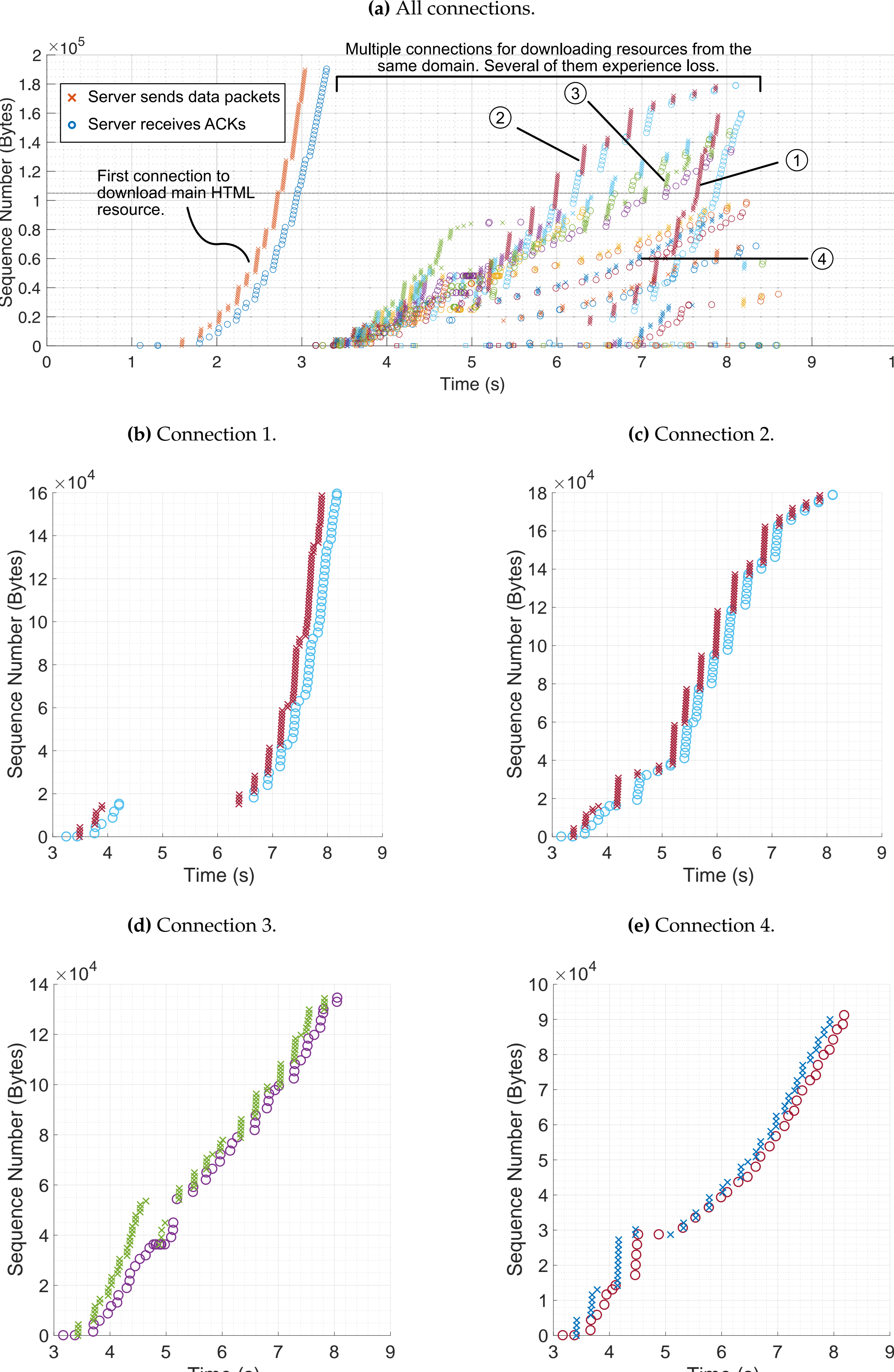


Figure 4.20: Time/sequence plot for buzzhand.com downloaded using HTTP/1.1 with six connections at a bandwidth of 2.5Mbps and RTT of 200ms.

Because TCP New Reno's linear growth function after a loss seems to be an integral cause of sub-par PLT performance, we expect better results when using faster growth functions like TCP CUBIC. We show a time/sequence plot of downloading buzzhand.com using HTTP/2 but with TCP CUBIC instead in Figure 4.19(b). Because CUBIC increases the window after a loss according to a cubic function with an inflection point centred on the window size before the loss, the congestion window ramps up to pre-loss values much faster, and the completion time improves to become 8.10s. But even TCP CUBIC cannot provide much help when losses occur very early in slow-start and the congestion window is very small. For these cases, it has been shown that TCP CUBIC's window growth remains small [139]. By minimising loss in the first place, BBR may be a better option for transport, especially since it can probe for the correct bottleneck bandwidth and increase link utilisation.

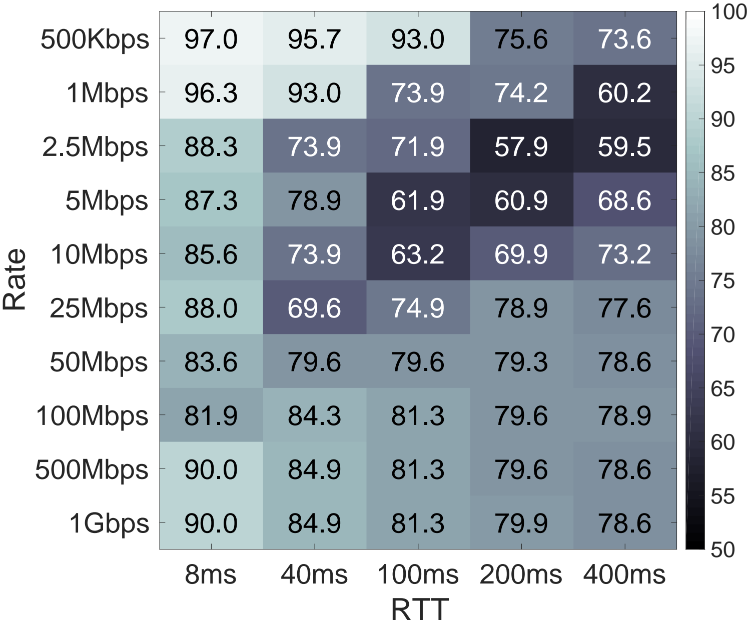


Figure 4.21: Heatmap indicating the percentage of web sites with equal or better mean PLT using QUIC-lite with 1 connection and no prioritisation compared with HTTP/1.1 with six connections.

Does QUIC-lite improve performance? Losses may cause extra delays due to HOL blocking in the TCP socket buffers. We expect better PLT performance if we use QUIC-lite, which allows for out-of-order delivery of packets to the application layer. A dropped packet therefore does not delay the arrival of packets higher in the sequence space and the completion of the corresponding resources. If we plot another heatmap with HTTP/1.1 as the baseline, but this time compare it with QUIC-lite without prioritisation to isolate the effects of only using out-of-order packet delivery, we obtain the results shown in Figure 4.21. Out-of-order delivery offers only minor reductions in PLT when compared with regular in-order TCP shown in Figure 4.16. Our results thereby confirm the minor improvements described in [21]. The largest reductions occur for *(Bandwidth, RTT)* regimes which experience high loss, *e.g.* for bandwidths lower than 10Mbps, or RTTs of 8ms. Naturally, for regimes which experience no packet loss at all, HTTP/2

and QUIC-lite are equivalent.

4.4.3.2 Protocol Comparison

After evaluating the performance of HTTP/2 and QUIC-lite, which represent the best web protocols from among the six we've examined, we now ask: how do the other protocols compare? We answer this question by choosing five representative *(Bandwidth, RTT)* regimes, corresponding to the points labelled (a–e) in Figure 4.12. We depict the PLTs for these regimes using two styles of representation. We first provide a general feel for the absolute values of the PLTs and how they change across protocols. To do so, we plot the CDFs for the full 300 $\times$ 50 PLT values we obtained across our sample web sites and runs for each given protocol in Figure 4.22. Second, we show the difference between PLTs across the protocols. For each *(Bandwidth, RTT)* combination, we compute the mean PLT across 50 runs for each web site downloaded using HTTP/1.1 with six connections. Using these means as a baseline, we then compute the difference in mean PLT between the baseline and the different web protocols we emulated. Figure 4.22 shows CDFs of this difference for the six regimes. Anything to the right of the vertical line at 0s has a larger completion time than HTTP/1.1.

Even before any close scrutiny, the figures immediately indicate that the PLT progressively improves as we change the web protocols in the following order: HTTP/1.0, HTTP/1.1, HTTP/1.1 Pipe Seq, HTTP/1.1 Pipe, HTTP/2 and QUIC-lite. For low bandwidths, the relative difference between protocols is minor because the overall PLT is quite large, as we can see in Figures 4.22(a) and 4.22(b). But as we increase the bandwidth, the discrepancy between protocols becomes more definitive, and the CDF curves peel away from each other.

Even before we embarked on our analysis, the fact that HTTP/1.0 is suboptimal had been self-evident. The graphs prove this point quite markedly; we only include HTTP/1.0 here for completion. HTTP/1.0 is especially bad at high bandwidths, where it increases the PLT over HTTP/1.1 by several seconds (Figures 4.22(d), 4.22(e), 4.23(d), and 4.23(e)). When there is enough bandwidth, HTTP/1.0 fails to take advantage of the available capacity because it does not amortise connection start costs and is always stuck at the beginning of slow-start, with a small congestion window. What is surprising though, it that for low bandwidths, for example 2.5Mbps in Figure 4.23(b), 30% of the web sites experience better PLTs under HTTP/1.0 than HTTP/1.1. HTTP/1.0 can do better than HTTP/1.1 in cases where there is not enough capacity and its flows are always in slow-start. In consequence, HTTP/1.0 doesn't experience the losses which occur in HTTP/1.1 as it ramps up its congestion window.

Next, we examine what happens to the PLT if we apply pipelining to HTTP/1.1 without making any modifications to the classic server architecture, such that pipelined requests are handled serially. In the figures, this option is labelled "HTTP/1.1 Pipe Seq". With a sequential server, we've seen in previous sections that we need a larger number of connections to achieve reasonably small PLTs. Even with the use of six connections per domain to increase parallelisation, this scheme suffers from HOL blocking at the server, and the graphs show that

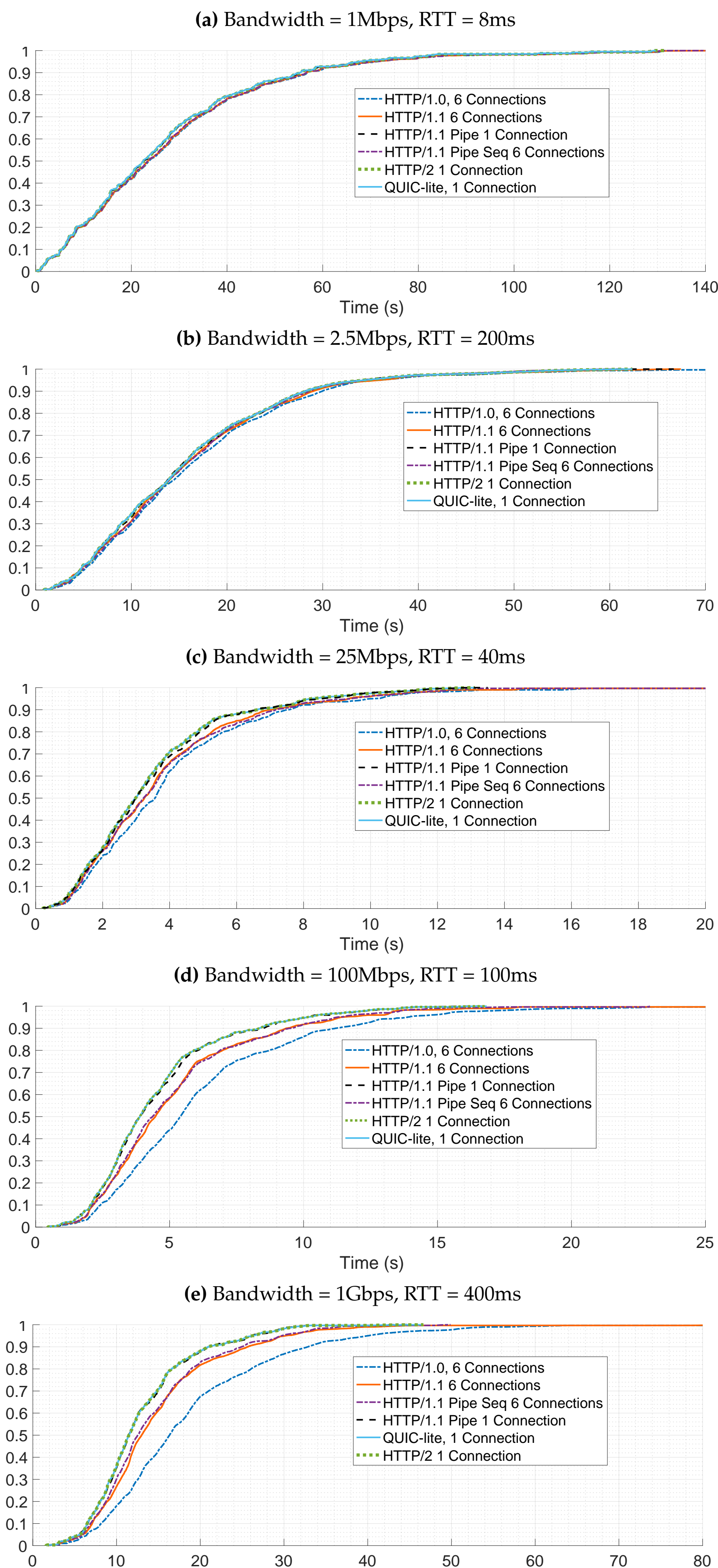


Figure 4.22: CDFs of the PLTs for 300 web sites across 50 runs under the different web protocols. The curves for QUIC-lite and HTTP/2 are overlaid.

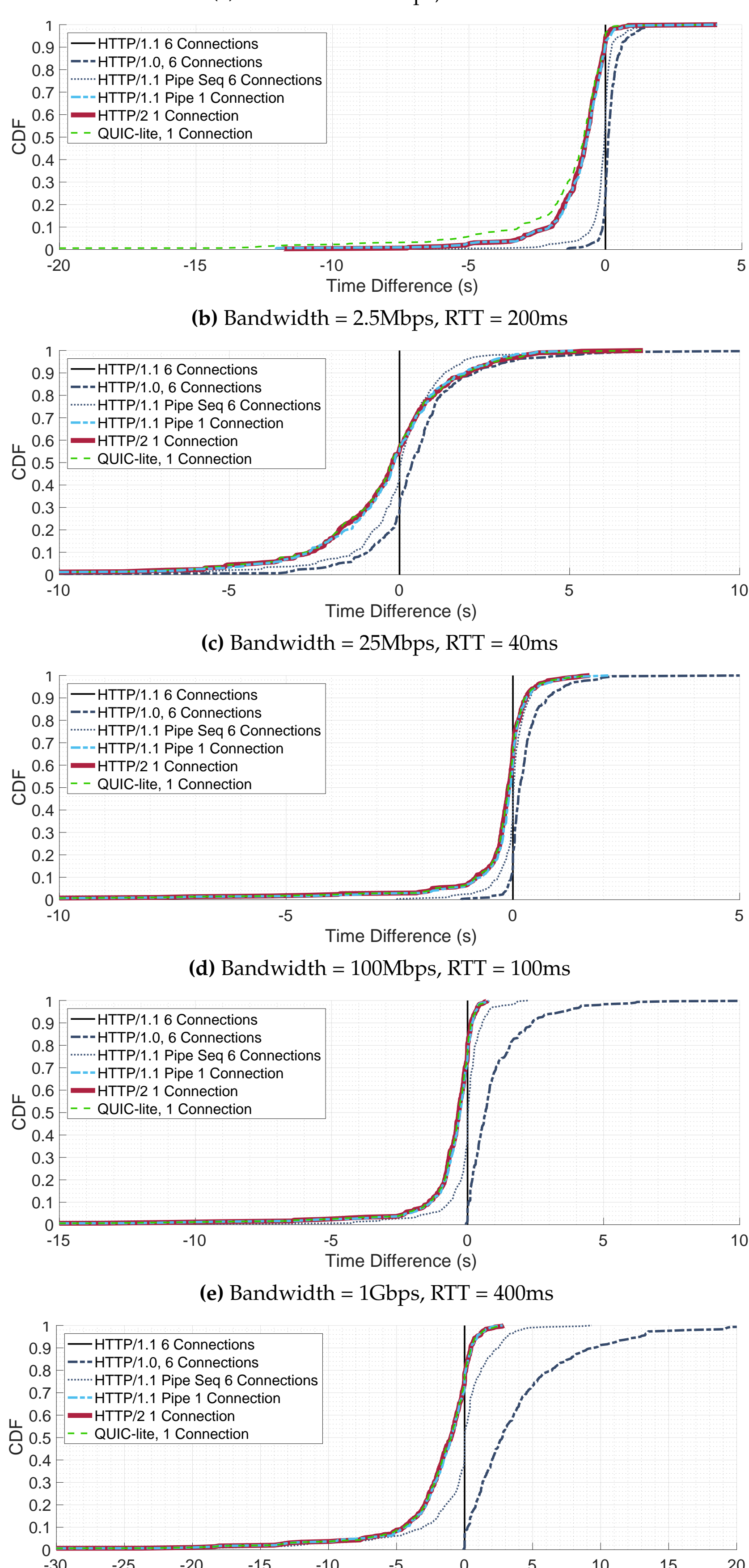


Figure 4.23: CDFs of the difference between the mean PLTs of 300 web sites under the different web protocols compared with HTTP/1.1 with six connections. The curves for HTTP/1.1 Pipe, HTTP/2 and QUIC-lite are overlaid.

for 40-50% of the web sites we emulated under different RTTs and bandwidths, HTTP/1.1 Pipe Seq produces higher PLTs than simply using unpipelined HTTP/1.1. We will investigate the serialisation delays more closely in Section 4.4.5, but suffice it to say that pipelining is disabled for HTTP/1.1 because it generally does not help, as is apparent from the graphs.

It is only when we use a parallel server with pipelined HTTP/1.1 that we finally begin to witness an improvement in PLT, regardless of how practical this option is. The "HTTP/1.1 Pipe" graphs in Figure 4.23 are shown by the dotted light blue lines. These lie quite close to the HTTP/2 curves, which indicates that we can attain most of the pipelining benefit without having to implement an out-of-order server, as we will see in Section 4.4.5. HTTP/1.1 Pipe and HTTP/2 behave in a similar manner, adhering to the trends we've explained earlier in this section. Pipelining and using one connection improves the PLT for cases where either the losses are higher in HTTP/1.1 due to multiple connections self-interfering (*i.e.* Figure 4.23(a)), or for the cases which do not experience losses at all (Figure 4.23(d) and 4.23(e)). In cases where the loss rate is comparable, and using multiple connections may be beneficial to increase utilisation, HTTP/2 and HTTP/1.1 Pipe perform less ideally, as is the case for Figures 4.23(b) and 4.23(c).

QUIC-lite, *i.e.* out-of-order packet delivery, only makes a difference when the BDP is low enough to incur a large packet loss rate, as is the case for Figure 4.23(a). For the rest of the graphs, packet losses are minimal or non-existent, which explains why the QUIC-lite curve lies directly on top of the HTTP/2 curve.

4.4.4 Prioritisation Schemes

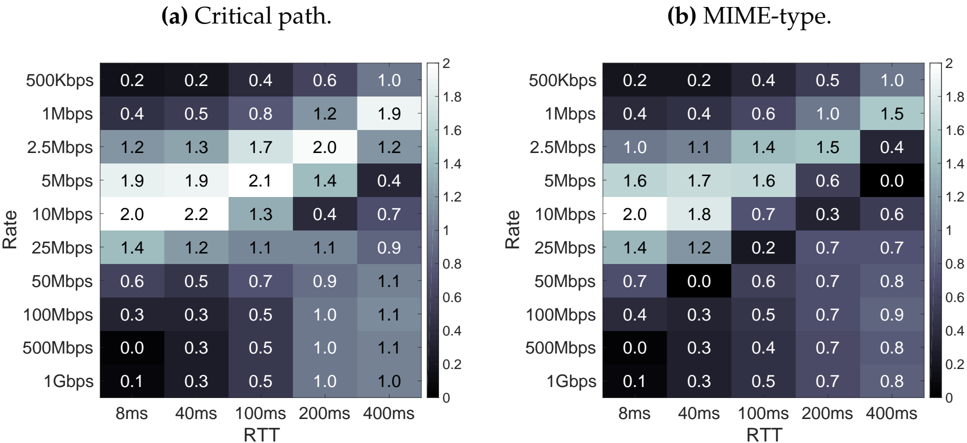


Figure 4.24: Percentage improvement in mean PLT for HTTP/2 with the specified prioritisation over HTTP/2 with no prioritisation.

Due to serialisation in the transport layer, when HTTP/2 multiplexes several resource streams onto one connection, critical resources may be delayed behind resources of lower priority, as we've seen in the previous section. To prevent cases where images or fonts defer the arrival of critical CSS files for example, HTTP/2 allows the browser to assign priorities to each

stream to indicate its importance. In this section, we will examine two different prioritisation strategies, the simple MIME-type strategy and the more precise critical path strategy, which we've described in Section 3.5.3.

Resource prioritisation, regardless of the strategy chosen, generally produces an improvement in PLT. Figures 4.24(a) and 4.24(b), which show the percentage improvement in mean PLT over unprioritized HTTP/2 if we apply either MIME-type or critical path prioritisation respectively, indicate a minor performance increase across the board. Additionally, the figures reveal several key properties. First, very little difference exists on average between using MIME-type prioritisation *vs.* critical path. Real browsers, as we've seen in Section 3.5.3, choose a strategy which combines features of MIME-type and critical path prioritisation. In essence, we are bounding the possible improvements by examining an ideal (but infeasible) omniscient strategy, and a much simpler but implementable MIME-type strategy. The figures show that the range is rather narrow, yet the behaviour of a real browser will fall between the two strategies. MIME-type prioritisation works reasonably well because it assigns higher priorities to resource types highly likely to be blocking resources when using critical path analysis, *i.e.* CSS and JavaScript files.

Second, the regions where prioritisation seems to help the most are those in which HTTP/2 performs relatively poorly, in other words the diagonal across the heatmap from Figure 4.16, and in high RTT regimes. When connections suffer from high RTT, the benefit of moving critical resources earlier becomes more evident. Similarly, for cases where packet drops cause HOL blocking while waiting for TCP retransmissions, it is usually beneficial to transmit the more important resources first.

After using the heatmaps to provide us with an average picture across 300 web sites, let us now focus on several distinct regions of operation to understand the variability in PLT across web sites as we change the prioritisation scheme. Figure 4.25 displays the difference in mean PLT between HTTP/2 with the chosen prioritisation scheme and a baseline HTTP/1.1 with six connections. The figures reinforce our earlier conclusions: both priority schemes perform better in general than no prioritisation at all, with critical path performing only slightly better than MIME-type. In areas where there are no losses and the bandwidth is fast enough, prioritising resources makes very little difference to overall PLT, as we can see for 100Mbps and 100ms in Figure 4.25(d). For lower bandwidths where losses are rampant, prioritisation provides a larger advantage. The effect is larger when the RTT is large, for example compare the cases of downloading web sites at 1Mbps but with an RTT of 8ms *vs.* 400ms, as shown in Figures 4.25(a) and 4.25(b). Because it takes longer to recover from the losses, critical resources may remain stalled for a longer duration.

With the previous overview of in mind, we now consider a more specific example to demonstrate how resource prioritisation affects the page load completion of a particular web site. Our chosen example is steamcommunity.com, whose dependency graph is shown in Fig-

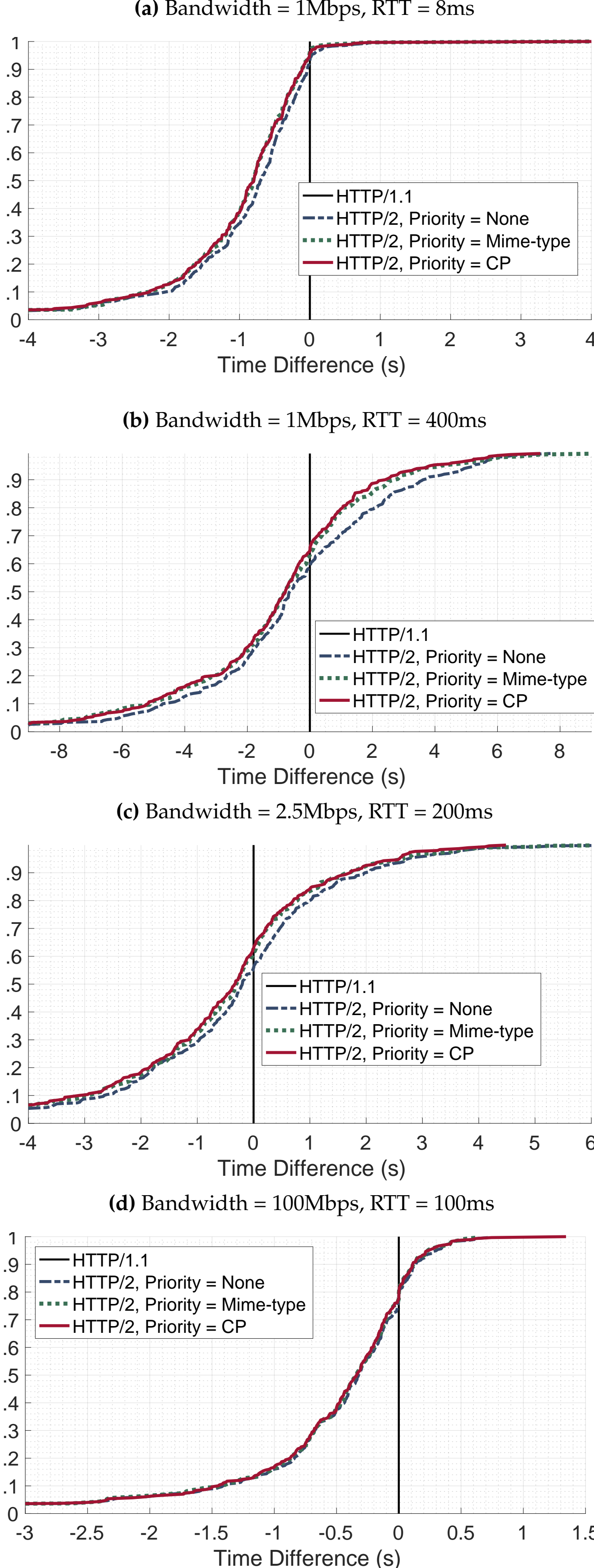


Figure 4.25: CDFs of the differences of HTTP/2 with 1 connection and different priorities, and HTTP/1.1 six connections. CP is a shorthand for critical path prioritisation.

ure 4.26. The graph reveals several synchronous JavaScript resources downloaded in the middle of the page, all of which must complete before the page can download a final dependent HTML resource. The JavaScript resources and the following lower priority image resources (not marked) reside on the same domain. Resource 25, marked in the graph, is the first blocking JavaScript of the page. Because we model the preloader, we expect it to begin pre-parsing the rest of the page as soon as the browser requests Resource 25.

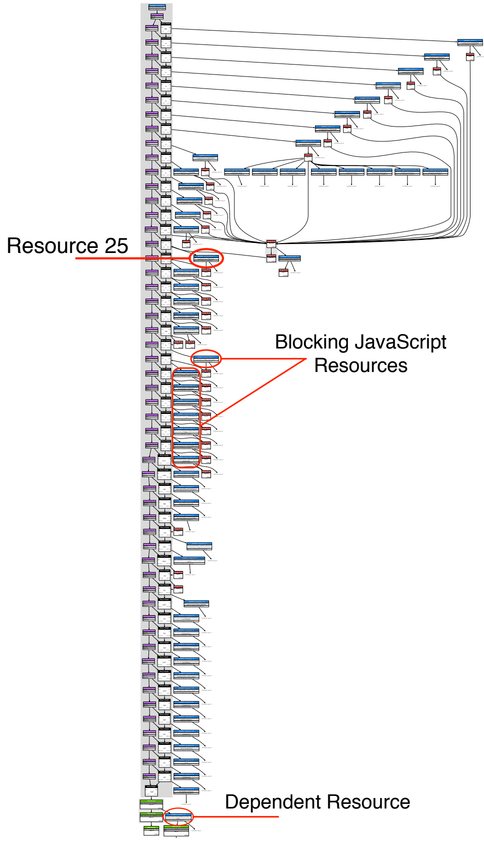
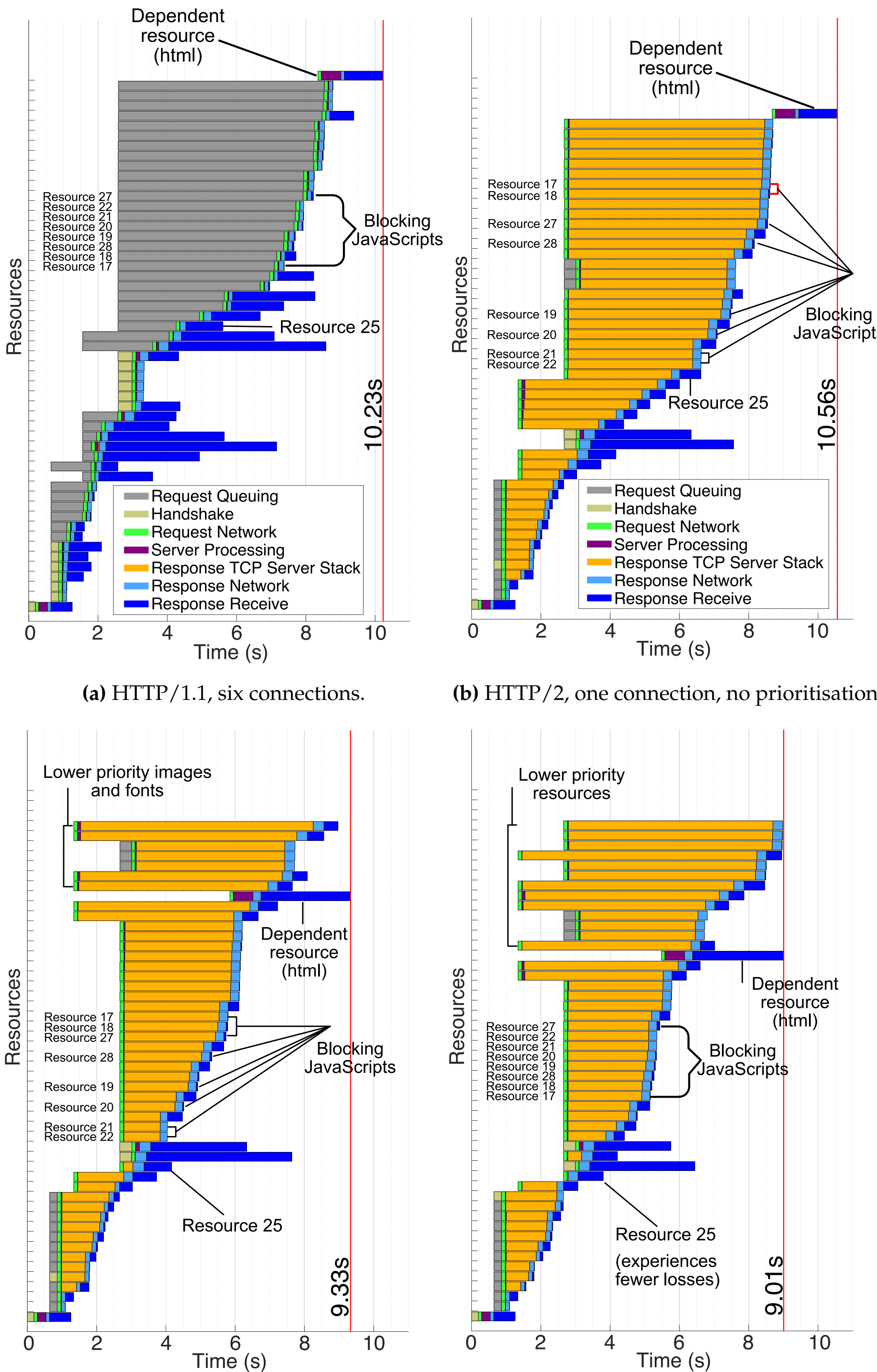


Figure 4.26: Dependency graph for steamcommunity.com.

To understand where the time loading steamcommunity.com is spent, in Figure 4.27 we plot the components which contribute to page load latency for each downloaded resource on a timeline. These components include TCP's 3-way handshake, request queuing at the client, half an RTT for transmitting the request to the server, the server processing delay, the response wait time in the server's TCP socket buffer, half an RTT for the client to receive the first byte of the response, and finally the time it takes to transmit the rest of the response over the network. We do not explicitly plot times spent parsing or evaluating JavaScript, but these delays appear as idle times in the graph. We mark the time the page's load event fires with a red line.

We begin with HTTP/1.1 using six connections. We emulated downloading the web page at a bandwidth 2.5Mbps and RTT of 200ms, a regime in which HTTP/2 performs particularly poorly. Due to the fact that each subsequent HTTP request must wait for the previous response to arrive, the timeline in Figure 4.27(a) shows that a large proportion of the time is spent in re-



(a) HTTP/1.1, six connections.

(b) HTTP/2, one connection, no prioritisation.

(c) HTTP/2, one connection, MIME-type prioritisation.

(d) HTTP/2, one connection, critical path prioritisation.

Figure 4.27: Timelines for downloading steamcommunity.com under different priority schemes, downloaded at 2.5Mbps and an RTT of 200ms.

quest queuing, as marked by the grey rectangles. Since the preloader kicks in when the browser requests Resource 25, all the subsequent resources are preloaded in order, which is why we don't see a step behaviour in the timeline graph as a result of blocking JavaScripts. Nevertheless, because HTTP/1.1 uses multiple connections per domain, the blocking JavaScript resources and other lower priority objects are downloaded in parallel, which allows the final dependent HTML resource to be downloaded even while previous resources complete. The page completes after 10.23s.

In contrast, when using HTTP/2 with one connection, the critical JavaScript resources are downloaded from the same domain and multiplexed with other less critical objects on the same connection (Figure 4.27(b)). Resource 17, the last of the blocking JavaScripts, is downloaded much later in the game, since it is delayed by lower priority images on the same connection. As such, the HTML resource, which requires Resource 17 to complete first, is downloaded late and pushes the completion time out to 10.56s.

Clearly, this scenario could benefit from applying some measure of prioritisation. Figure 4.27(c) shows the results of using the MIME-type strategy, which reduces the completion time to 9.33s. Applying the MIME-type rules, the browser assigns JavaScript objects a higher priority than images, prompting their timely download, which triggers the earlier download of the dependent HTML resource.

We include the critical path prioritisation case for completion. The completion time fairs slightly better under critical path prioritisation (Figure 4.27(d)), but the cause of this improvement is actually more subtle. MIME-type prioritisation considered all the JavaScript resources of equal priority, but critical path prioritisation recognises that earlier JavaScripts should have a higher priority since they block the subsequent ones from executing. But this advantage is overridden by the browser's preloader requesting all the blocking JavaScripts in close succession. The real reason for the improvement lies in the placement of random drops during the download of critical resources. Because under critical path prioritisation Resource 25 experiences less drops, the subsequent resources requested on the same connection experience a smaller delay, and the page completes faster. We later present an example where critical path prioritisation legitimately outperforms a MIME-type scheme based only on the merit of its prioritisation scheme in Section 5.3.4.

Is resource preemption when using prioritisation helpful at all? We answer this question using Figure 4.28, which shows the percentage improvement in mean PLT over unprioritized HTTP/2 if we use QUIC-lite with either MIME-type or critical path prioritisation respectively, with resource preemption enabled. The heatmaps show the same trends as those in Figure 4.24, with the larger improvements lying across the diagonal. At lower bandwidths, out-of-order packet semantics and resource preemption combine to produce an overall improvement in PLT. But for regions where there is little packet loss, out-of-order delivery has no effect on PLT. Instead, it is prioritisation with preemption that produces a larger win in PLT, especially when

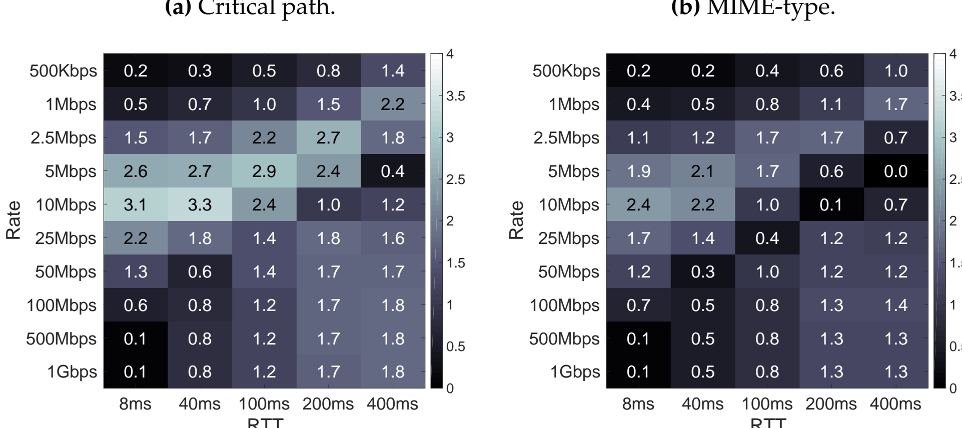


Figure 4.28: Percentage improvement of QUIC-lite with the specified prioritisation over HTTP/2 with no prioritisation.

compared to the scenarios with no preemption we've seen earlier.

We end this section with some closing remarks about the overall benefit of prioritisation. At first glance, it may appear peculiar that prioritisation does not truly generate much of a win in PLT. To perform prioritisation optimally, a single server must receive *all* the resource requests, and must correspondingly have all the responses ready at the same time after the processing delays. Only with all the responses available at once can the server produce a global ordering of resources and hence exact the largest improvement in PLT. Of course, these conditions do not represent the reality of web downloads. Interspersed requests arrive at the server which applies variable processing delays. At best, only a subset of resources are ready to be sent in the server's socket buffer. This implies that prioritisation works better if the processing delays are small, especially for critical resources. To make matters worse, web site content is not concentrated on one server, but spread out due to content sharding, ads and third party content.

Furthermore, upon close examination of our profiled web pages' dependency graphs, we conclude that most large web sites are optimised for "above-the-fold" speed and therefore apply resource prioritisation implicitly. They are fairly well structured and refer to critical resources early in the HTML, placing (for example) CSS files and synchronous JavaScripts at the top of the page. Finally, using the preloader greatly masks a lot of the benefit of using prioritisation. At heart, we cannot make large improvements with prioritisation outside of server push [51]. Nevertheless, we must point out that even a small 3% reduction in PLT at best has been large enough to justify a large amount of design work and a switch to a whole new protocol suite.

4.4.5 Server Design

Since we cannot really decouple the server design from the protocol used, we have already witnessed how the server design influences the completion time of pipelined HTTP/1.1 and

HTTP/2 in Section 4.4.3. In this section, we focus on a particular web site and use it to explain in more detail the impact of server design on PLT.

Figure 4.29 shows the timeline for downloading steamcommunity.com using the three different server architectures, serial, parallel, and out-of-order . Unlike the previous section, here we download the web site using only one connection per domain for all the variants, to allow the differences between the server designs to become more prominent. We apply critical path prioritisation to *all* designs so that we can remove prioritisation as the cause of difference between them. Recall that we apply prioritisation to packets within the transmit socket buffer, and can therefore easily prioritise resources for all HTTP variants. The bandwidth used for the download is 25Mbps and the RTT is 40ms.

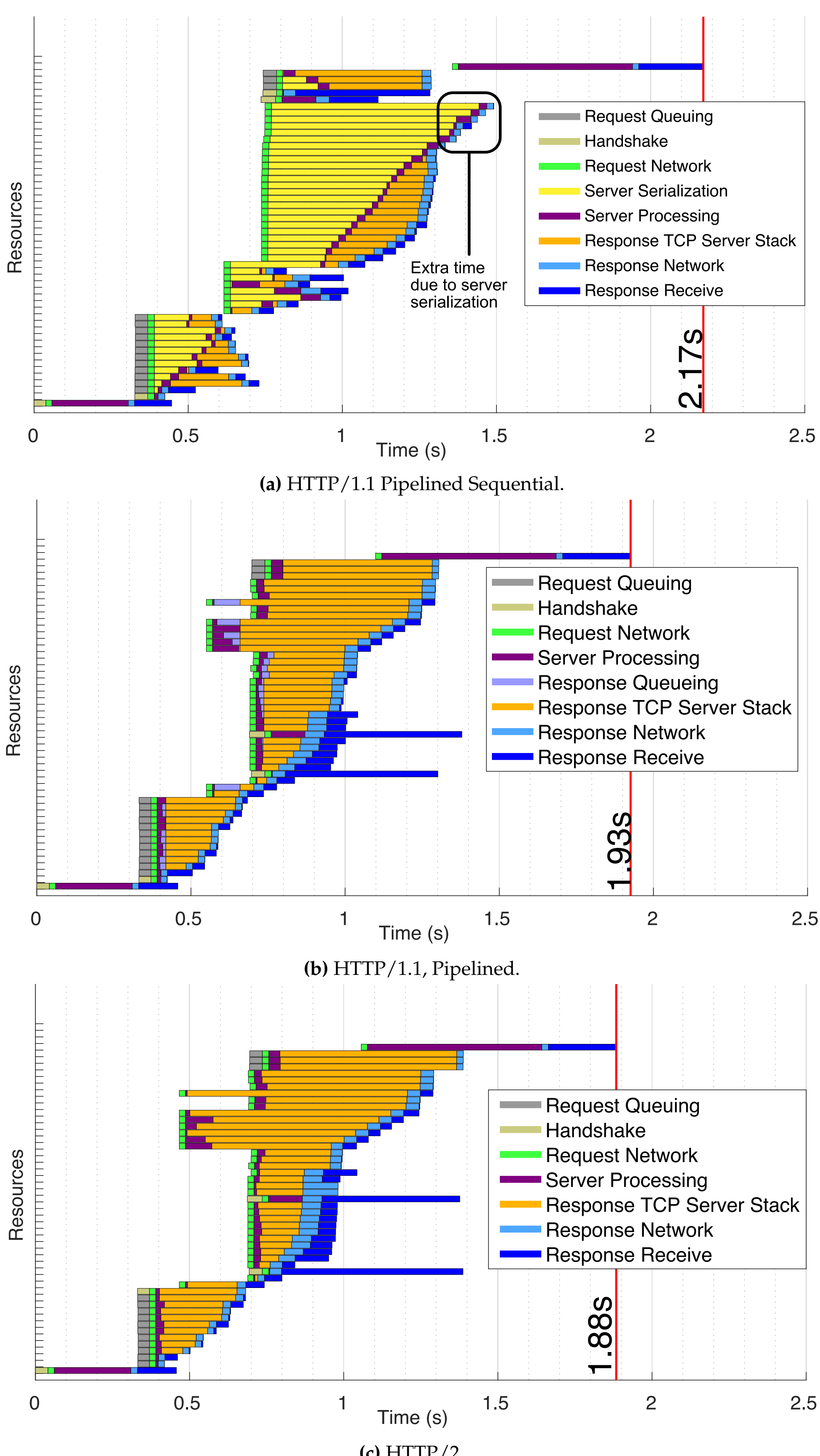
If browsers turn on pipelining in HTTP/1.1 by default, without a complete server redesign we would arrive at the timeline shown in Figure 4.29(a). The typical multi-threaded server must process each incoming request in FIFO order, therefore it will enqueue incoming requests until the time is right to process them. This serialisation delay appears clearly as the yellow lines in the figure. When the serialisation delay (in yellow) is in excess of the actual time spent enqueued in the TCP stack waiting for the previous resources to be transmitted over the network (in orange), the PLT experiences an unnecessary delay as clearly marked in the figure.

A tangible improvement in PLT can be achieved if the servers switch to a parallel design instead, processing multiple requests at once. We can see from Figure 4.29(b) that, although this design reduces the PLT, the server still must enqueue the responses after it processes them so that it can return them to the client in FIFO order, as shown by the light purple lines. HTTP/1.1 pipelining with parallel processing at the server seems like reasonable halfway-point solution for improving HTTP/1.1 without having to replace the protocol entirely. Unfortunately, this server design is impractical, because it requires provisioning extra memory at the server. Additionally, if a failed response terminates a TCP connection, the browser must reissue all unanswered requests, possibly duplicating the associated processing at the server, which is wasteful.

Finally, using HTTP/2 with an out-of-order server eliminates the need to return resources in FIFO order, and therefore any response queuing at the server. Figure 4.29(c) shows that by processing requests in parallel, and returning the responses out of order, the light purple disappears from the graph, and the PLT improves slightly further. In summation, the most efficient method for achieving request/response pipelining without incurring extra HOL delays is to use HTTP/2 with an out-of-order server design. Doing so eradicates any excess queuing at the server due to request or response queuing. The only remaining serialisation resides in the TCP stack.

4.4.6 Conclusions

Several network parameters, transport layer artefacts, and protocol design features interact together to influence a web page's PLT, as we've shown in this chapter. To reduce page load



(a) HTTP/1.1 Pipelined Sequential.

(b) HTTP/1.1, Pipelined.

(c) HTTP/2.

Figure 4.29: Timelines for downloading steamcommunity.com under different server architectures, downloaded at a bandwidth of 25Mbps, RTT of 40ms, and 1 connection per domain.

latencies, our analysis has demonstrated that we must adopt web protocols and server designs that increase parallelism, prevent HOL blocking in any layer of the web download, and maximise network utilisation. As a result, protocols like HTTP/2 and QUIC with out-of-order server processing and resource prioritisation emerge as the best candidates for fast web browsing.

But despite their success in reducing PLTs, we've shown that under certain network conditions and loss rates, HTTP/2 and QUIC's performance can be suboptimal. Even mechanisms like resource prioritisation, which were aimed to improve PLTs in light of serialisation in the TCP socket layer, fall short of generating a large gain in performance. Prioritisation can, at best, only occur among a small subset of resources due to the web page structure, server processing delays, and server sharding.

Unsharding web pages may help, as we will see in the upcoming chapter, especially since we've shown that at the time we collected our traces, most web site architectures were still geared towards supporting HTTP/1.1, and therefore sharded their content across multiple servers. Nevertheless, we've also shown that ads and third party content form a tangible proportion of a web site's content; for these domains, unsharding will have no effect.

To combat pathologies that arise due to packet loss, a possible solution is to use a different transport protocol which allows faster recovery from loss, for example TCP CUBIC, or transport protocols that reduce packet losses in the first place, like BBR. In the end, perhaps the most definitive way to produce a further reduction in PLT is to implement server push.

Chapter 5

Unsharding

5.1 Introduction

Although HTTP/2 was conceived with the specific intent of remedying HTTP/1.1's shortcomings, we've demonstrated in the previous chapter cases where it in fact *increases* PLT, for example when packet losses are high. But even for the scenarios where HTTP/2 was designed to perform better, particularly where the bandwidth and RTT are high and the connection has a high BDP, HTTP/2 still does not achieve the highest possible win with the web site traces we've collected, mainly due to a mechanism which still exists in the web today as a workaround for HTTP/1.1's suboptimal design.

To explain, let us recap some of HTTP/1.1's properties. HTTP/1.1 supports simple pipelining of requests, but web browsers disable this feature by default. Pipelining with HTTP/1.1's FIFO semantics may result in a slow response blocking all others and reducing performance. Instead, browsers open several concurrent connections to the same web domain, for example Chrome allows up to six parallel in-flight requests. Today's web pages, however, consist of on the order of a hundred objects, so six connections is often insufficient: requests for critical objects will remain queued in the browser, waiting for a free connection. As a workaround, web content publishers embraced *domain sharding*: instead of serving content from one origin, they spread the page's resources across multiple subdomains, bypassing the browser's six connection limit, and increasing parallelism. Sharding also permits parallel processing of simultaneous requests on the server, reducing server processing latency.

In contrast, an HTTP/2 browser opens one TCP connection to each server, and pipelines requests so they reach the server with minimal delay. An HTTP/2 server can process these requests in parallel and return responses in any order, or even multiplex them using intra-connection framing. Moreover, HTTP/2 allows the browser to choose a prioritisation of its requests to the server, which the server follows when ordering its responses. TCP's 3-way handshake and slow-start incur latency; HTTP/2 amortises this cost over the multiple objects transferred on the same connection. Fewer connections between the browser and a single server reduce contention for bandwidth at the last-mile bottleneck, lowering loss rates and, in turn, reducing latency for retransmitting lost packets.

HTTP/2's designers generally view domain sharding as an obstacle to HTTP/2 minimising PLT. If objects are spread over multiple sharded domains, HTTP/2 can neither maximally amortise connection startup costs across objects nor make fullest use of its prioritisation mechanisms (which communicate inter-object priorities only for those objects served by the same server). Standard advice is now for web designers to *unshard* their domains [16, 23]. In fact, web browsers already coalesce some sharded domains automatically: if two domain names resolve to the same IP address and are named as alternatives in a TLS certificate, the browser will use one TCP connection to communicate with both.

How good is this advice to unshard? We observe that even with HTTP/2 it is not difficult to deliberately create web pages that load faster when sharded than unsharded. For example, a series of blocking JavaScripts referenced at the end of the HTML file might be able to execute in parallel with other object downloads when sharded, masking the latency, but be serialised after the downloads when unsharded.

We wish to understand the benefits and costs of unsharding today's complex web sites. We demonstrate that the conventional wisdom is mostly correct: on average, unsharding definitely delivers a PLT improvement. However, the story is more complex than that. The benefits or otherwise of unsharding depend on subtle interactions between the web site's resource dependencies, the browser's use of HTTP/2 prioritisation, and both the bandwidth and latency of the network connection. Our study of 300 of the Alexa top-500 web sites reveals under which circumstances unsharding these websites delivers a performance improvement using HTTP/2 and under which it hurts performance. We observe that almost no web site benefits from unsharding under all network conditions, and a few almost always perform worse. We investigate the root causes, and identify both a deficiency in the current TCP congestion control standard [132] that web-like traffic happens to exercise, and web page dependency structures best avoided to minimise PLT when unsharding.

5.2 Methodology

We wish to compare sharded and unsharded versions of the same web page. Each page consists of resources downloaded from dozens of domains. To see how effective unsharding might be, we consolidate all URLs associated with a site's primary domain that were originally under the same second level domain. Thus media.gizmodo.com and gizmodo.com are both consolidated to gizmodo.com, but links to akamai.com are not consolidated. A few special cases also need considering, such as sites ending in .co.uk, where we consolidate the third-level domain. Although two URLs may belong to the same corporation, we do not coalesce them if they refer to two different primary domains, for example partner.googleadservices.com and www.googletagservices.com. Domains corresponding to advertisements or third-parties are not consolidated.

This form of unsharding is not intended to be predictive of the actual performance wins to be obtained, as web sites may have other reasons to distribute content across multiple servers.

For example, media.gizmodo.com and gizmodo.com may correspond to a different server infrastructures that may be difficult to combine without a major redesign. We cannot determine whether a page shards domains for performance reasons or not, but we assume it does so all the same, since it is impossible to make such a distinction without in depth knowledge of the server infrastructure. The results we present are only predictive of the *possible* performance gains, *if no other concerns applied*.

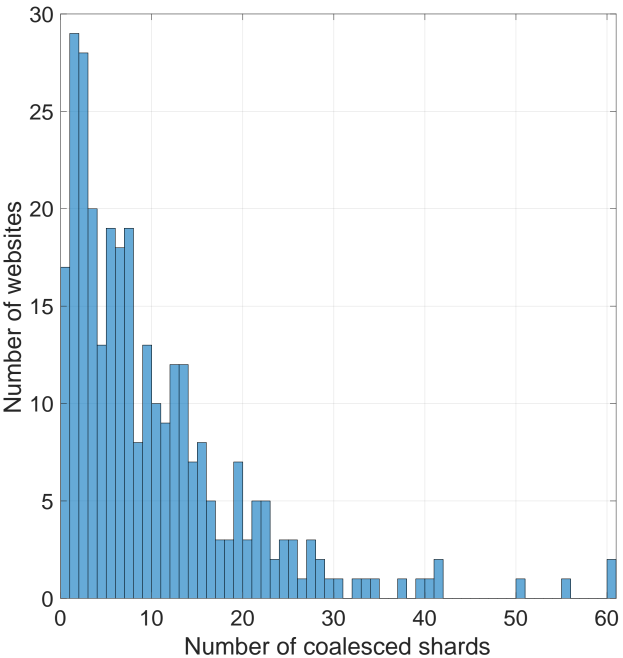


Figure 5.1: Histogram depicting the number of shards coalesced for the 300 web sites.

How much sharding do we actually see? Figure 5.1 is a histogram of the number of shards we coalesced for each of the 300 web sites in our corpus. These correspond to the shards of the web page's main domain plus any coalesced shards belonging to subdomains. The figure shows that 17 sites had no sharding whatsoever; we will exclude these sites from our unsharding experiments. A further 46 sites had two or three domain names associated with their servers. These are unlikely to be sharding for performance reasons, but more likely to be using multiple subdomains for operational reasons. We include these in our experiments, but the effect of coalescing for these is unlikely to be large. Around half the sites appear to shard for performance reasons, as they spread traffic across ten or more subdomains. Two sites have 60 shards, one has 55, one has 50, and 37 sites have between 20 and 42 shards.

Using PCP with the standard topology and setup, we emulated the download of the 283 Alexa web sites from our traces using HTTP/1.1 with six connections per domain and using HTTP/2 using a single connection per domain and MIME-type prioritisation. Although a dependency-based policy based on critical path analysis may have served us better, it outperforms any real browser policy because it knows the future. In contrast, MIME-type prioritisation is easy to implement, and comes very close in performance to the dependency-based

policy, as we have shown in Section 4.4.4.

After completing the experiments using the sharded domains, we then unsharded the domains, and repeated the HTTP/2 experiments. Once again, we used one connection per domain, and MIME-type prioritisation for the unsharded experiments. For each protocol we downloaded each web page under all the combinations of RTT and bandwidth values shown in Figure 5.2(a), and recorded the time it took before the `load` and the `DOMContentLoaded` events fired, indicating the page had loaded.

5.3 Results

We present an analysis of the possible performance gains of unsharding in this section. First, we discuss broad conclusions about unsharding and under which network configurations it produces the best performance wins. We then investigate what goes wrong when unsharding hurts HTTP/2 performance, identifying lessons for web page design through closer examination of particular case studies.

5.3.1 Overview

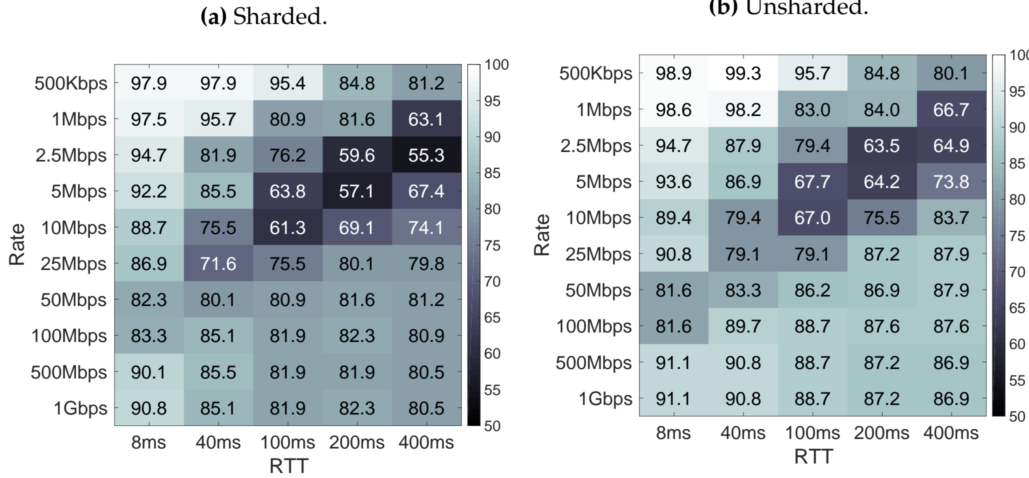


Figure 5.2: Heatmap showing the percentage of web sites with equal or faster mean PLT, as measured by the `load` event metric, using HTTP/2 (with MIME-type prioritisation) *vs.* HTTP/1.1 with 6 connections per domain, for web sites with sharded or unsharded domains.

Unsharding web pages boosts HTTP/2's performance with respect to baseline HTTP/1.1. We show this improvement for the 283 web sites we've examined in Figure 5.2, which exhibits two heatmaps that use HTTP/1.1 with 6 connections as the baseline. In Figure 5.2(a), we reproduce a figure similar to Figure 4.16 from Section 4.4.3.1, but this time applying MIME-type prioritisation to HTTP/2. Once again, for each *(Bandwidth, RTT)* we show the percentage of web pages which have less or equal mean PLT using HTTP/2 (with MIME-type prioritisation) when compared to HTTP/1.1. In contrast, Figure 5.2(b) shows the percentage of web pages which perform equal or better under *unsharded* HTTP/2 with MIME-type prioritisation when compared to HTTP/1.1.

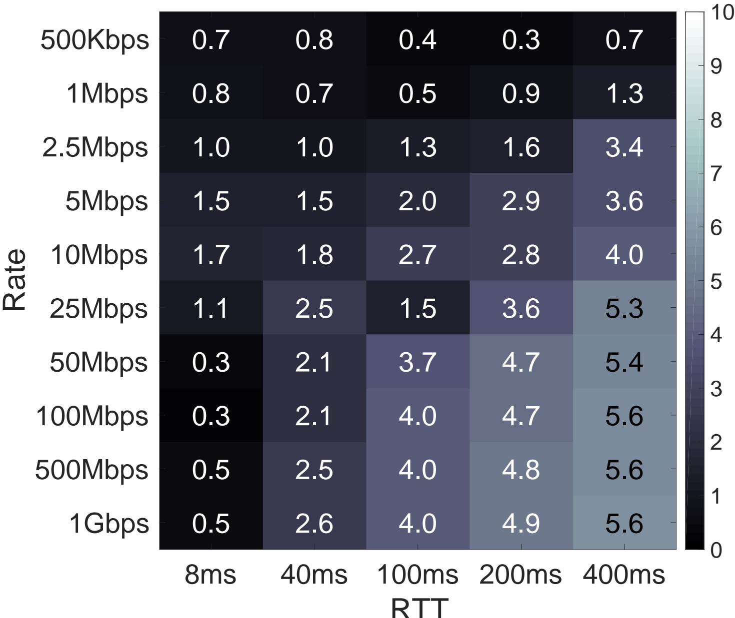


Figure 5.3: Percentage reduction in mean PLT as a result of unsharding domains under HTTP/2 with MIME-type prioritisation, when compared with the sharded domain case.

The root causes that give rise to the trends on both heatmaps, most notably the diagonal where HTTP/2 underperforms, have been described in Section 4.4.3.1; we do not repeat them here. One thing to note though is that MIME-type prioritisation generates a minor increase in the percentage of web pages which perform better than HTTP/1.1, as seen in Figure 5.2(a). Instead, we draw the reader's attention to the differences between the two heatmaps. By coalescing web page domains, the percentage of web pages which perform better under HTTP/2 *increases* for most combinations of bandwidth and RTT. Even the diagonal with the worst comparative performance improves with unsharding. For example, when the bottleneck bandwidth was 25Mbps and the RTT 400ms, 87.9% of unsharded web pages running HTTP/2 completed faster then HTTP/1.1 on average, in contrast to 79.8% of web pages under HTTP/2 with sharded domains.

The improvement generally gets larger as we move from left to right in the heatmap, thereby increasing the RTT. Similarly, the improvement is large in the lower right corner for high bandwidths and RTTs. To quantify the reduction in PLT, we compute $\frac{PLT_{shard} - PLT_{unshard}}{PLT_{shard}}$, the percentage reduction in mean PLT computed for each *(Bandwidth, RTT)* regime over 283 web sites, with 50 runs each, as shown in Figure 5.3. The figure confirms our earlier observations. Most notably, the lower right region shows the largest improvement in PLT as a result of unsharding web domains. Unsharding domains coalesces most of the downloaded resources onto a small number of connections. When the bandwidth and RTT are large, allowing for a large BDP and little packet loss, these connections can better grow their congestion windows in slow-start to fill the pipe, thereby utilising the available capacity more efficiently, and sending resources faster on the network.

Mode		Sharded *vs.* Unsharded					
Rate	RTT	$Plt_{shard}(s)$	$Plt_{unshard}(s)$	$\bar{x}$ (s)	p-value	σ	% Imp.
500Kbps	8ms	54.120	53.727	0.394	0.436	0.923	0.7
	40ms	54.477	54.067	0.411	0.419	1.249	0.8
	100ms	55.492	55.266	0.226	0.658	1.043	0.4
	200ms	57.136	56.945	0.191	0.712	1.743	0.3
	400ms	59.621	59.218	0.403	0.435	2.949	0.7
1Mbps	8ms	27.217	26.991	0.225	0.370	0.776	0.8
	40ms	27.725	27.542	0.182	0.473	0.742	0.7
	100ms	28.894	28.760	0.135	0.602	1.031	0.5
	200ms	30.228	29.960	0.268	0.301	1.653	0.9
	400ms	34.564	34.111	0.453	0.095	3.744	1.3
2500Kbps	8ms	11.272	11.165	0.107	0.286	0.454	0.9
	40ms	11.934	11.819	0.115	0.262	0.566	1.0
	100ms	12.963	12.803	0.161	0.126	0.973	1.2
	200ms	15.380	15.145	0.235	0.039	2.205	1.5
	400ms	20.623	19.926	0.696	0.000	4.568	3.4
5Mbps	8ms	6.261	6.167	0.095	0.069	0.457	1.5
	40ms	6.785	6.683	0.102	0.056	0.541	1.5
	100ms	8.279	8.115	0.164	0.006	1.137	2.0
	200ms	10.950	10.642	0.308	0.000	2.372	2.8
	400ms	16.052	15.490	0.562	0.000	4.078	3.5
10Mbps	8ms	3.950	3.884	0.066	0.035	0.341	1.7
	40ms	4.533	4.451	0.082	0.014	0.475	1.8
	100ms	6.144	5.980	0.164	0.000	1.162	2.7
	200ms	8.686	8.445	0.241	0.000	2.108	2.8
	400ms	13.726	13.191	0.536	0.000	2.751	3.9
25Mbps	8ms	2.746	2.716	0.030	0.200	0.190	1.1
	40ms	3.431	3.346	0.085	0.001	0.418	2.5
	100ms	4.886	4.814	0.072	0.027	0.904	1.5
	200ms	7.466	7.204	0.262	0.000	1.169	3.5
	400ms	12.978	12.313	0.665	0.000	1.787	5.1
50Mbps	8ms	2.476	2.468	0.008	0.720	0.122	0.3
	40ms	3.133	3.068	0.065	0.008	0.300	2.1
	100ms	4.576	4.411	0.165	0.000	0.421	3.6
	200ms	7.313	6.981	0.332	0.000	0.867	4.5
	400ms	12.844	12.169	0.675	0.000	1.525	5.3
100Mbps	8ms	2.382	2.376	0.006	0.776	0.098	0.3
	40ms	3.019	2.956	0.063	0.009	0.224	2.1
	100ms	4.514	4.338	0.176	0.000	0.358	3.9
	200ms	7.256	6.927	0.329	0.000	0.864	4.5
	400ms	12.792	12.100	0.692	0.000	1.412	5.4
500Mbps	8ms	2.332	2.320	0.012	0.589	0.060	0.5
	40ms	2.987	2.913	0.074	0.002	0.157	2.5
	100ms	4.497	4.320	0.176	0.000	0.357	3.9
	200ms	7.236	6.894	0.341	0.000	0.673	4.7
	400ms	12.784	12.087	0.698	0.000	1.244	5.5
1Gbps	8ms	2.329	2.316	0.012	0.581	0.052	0.5
	40ms	2.985	2.911	0.074	0.002	0.156	2.5
	100ms	4.495	4.320	0.175	0.000	0.353	3.9
	200ms	7.235	6.890	0.345	0.000	0.653	4.8
	400ms	12.783	12.083	0.700	0.000	1.237	5.5

Table 5.1: Mean PLT values for sharded and unsharded HTTP/2. $\bar{x}$ is the mean over all values of corresponding $(PLT_{shard} - PLT_{unshard})$ in seconds, σ is the sample standard deviation for $\bar{x}$. The % Imp corresponds to $\frac{\sum(PLT_{shard} - PLT_{unshard})}{\sum PLT_{shard}} \times 100\%$. The p-value represents the probability that the mean PLTs of sharded and unsharded HTTP/2 are not significantly different, and is the result of computing one-way ANOVA across the two groups.

Table 5.1 expands on the information displayed in Figure 5.3, and gives the absolute values of the mean PLTs for sharded and unsharded HTTP/2. Our metric of interest is actually $\bar{x}$, the difference between these two values. Using one way ANOVA to compare sharded and unsharded HTTP/2, we determined the p-values shown in the table. Each p-value indicates the likelihood of the null hypothesis, *i.e.* that the means of both groups are the same for the given bandwidth and RTT. As we can see, unsharding only produces a minor win on average over the sharded case. If we choose a significance level of 5%, unsharded HTTP/2 only achieves a statistically significant reduction in PLT when compared to the baseline if the BDP is large, for example for bandwidths larger than 5Mbps, or for RTTs larger than 8ms. For cases of low BDP, the connections do not need to slow-start to a large window, therefore there is no benefit in amortising connection startup costs by unsharding and coalescing traffic onto a single connection.

5.3.2 Unsharded *vs.* Sharded HTTP/2

Unfortunately, comparing unsharded HTTP/2 against HTTP/1.1 as a baseline fails to convey the entire picture regarding the effects of unsharding. Instead, a useful measure would be to compare unsharded HTTP/2 against the sharded case. But, observing only the mean improvement in PLT between sharded and unsharded HTTP/2 as we have done earlier masks crucial variations across web sites.

Figure 5.4 gives a similar overview to that in Figure 5.2, but instead of comparing HTTP/2 with HTTP/1.1, it compares the original sharded domains to unsharded versions of the same domains using HTTP/2 for both. Figure 5.4(a) shows results measured using the `load` event to determine completion, whereas Figure 5.4(b) shows results measured using the `DOMContentLoaded` event. Broadly, the results are similar for both metrics, and indicate that unsharded domains generally perform better. But, for some combinations of bandwidth and RTT, unsharding hurts performance almost as often as it helps, especially when measured by `load` event. For example, only 59.2% of runs measured by `load` event improve or stay the same when using 2.5Mbps and 200ms RTT, which are not atypical figures for an ADSL link in an area where the telephone exchange is a long way away. Therefore, although unsharding generally increases the benefits of HTTP/2 relative to HTTP/1.1, for some web sites unsharding may in fact reverse or reduce the improvement achieved by sharded HTTP/2.

The dark blue diagonal band in Figure 5.4(a) is interesting: both above and below this band unsharded domains perform better more frequently. Of particular concern is the fact that some of the network conditions within this band correspond to common settings for modern home networks. Several effects combine to hurt unsharded domains in this band. Figure 5.5 shows a heatmap comparing packet drop rates by network parameters. Each box indicates the percentage of web sites with those parameters where the mean packet loss rate was higher for HTTP/2 with unsharded domains. Three distinct regions are visible:

- In the bottom right, the zero values indicate that no loss was observed, as the delay-

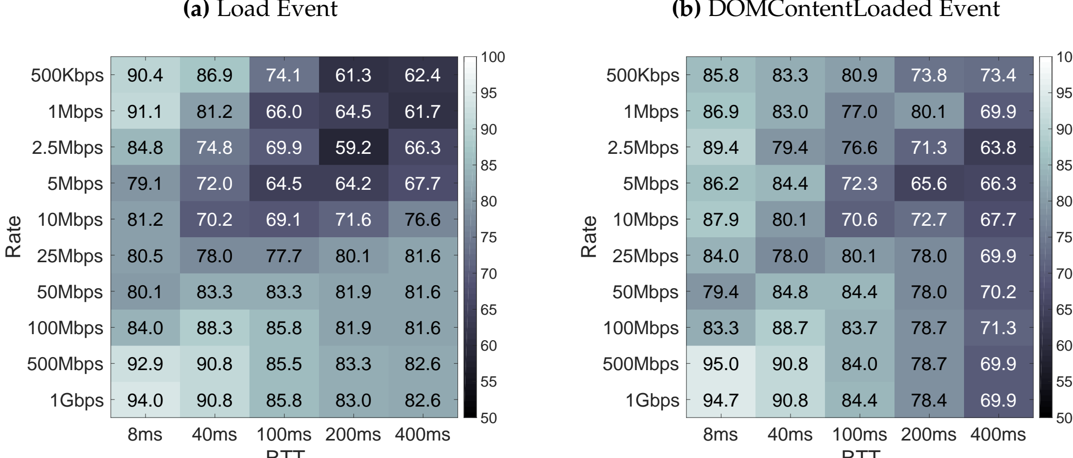


Figure 5.4: Heatmap showing the percentage of web sites with equal or faster mean PLT, as measured by the `load` or `DOMContentLoaded` events, using HTTP/2 with unsharded domains *vs.* sharded HTTP/2.

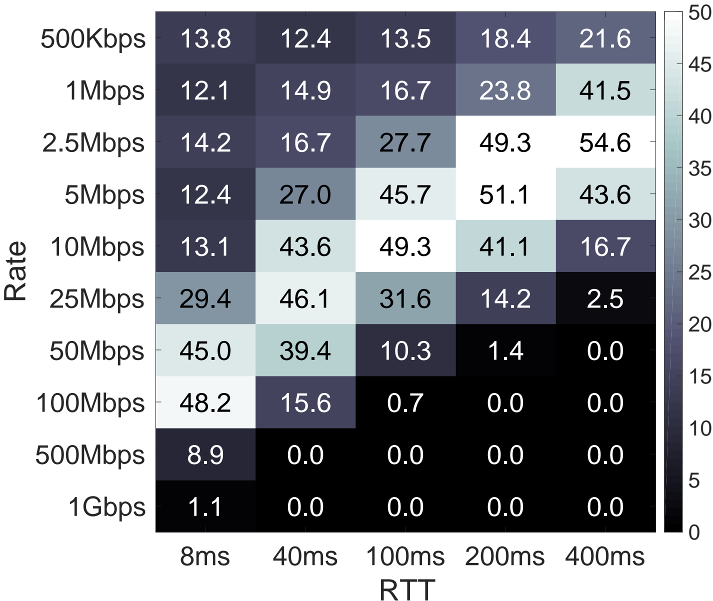


Figure 5.5: Heatmap showing the percentage of web sites where the packet drop rate was higher with unsharded HTTP/2 than the sharded case.

bandwidth product of the network was large enough that connections finished in slow-start. In this region unsharded domains do well, as they have more data per connection, so slow-start faster.

- In the middle band, over 40% of runs exhibit a higher loss rate on unsharded domains. This may seem counter-intuitive, as unsharded domains have fewer connections competing with each other, but in fact this is not the dominant factor. In this region, unsharded connections commonly slow-start to a large enough window to exhibit packet loss, whereas sharded domains more often finish a connection while still in slow-start without congesting the link. We will investigate how this interaction between HTTP/2

and TCP plays out in more detail in Section 5.3.4.1.

- In the top left corner, the delay-bandwidth product of the network is much smaller, and in these runs even the connections with sharded domains finish slow-start and cause congestion. These connections then compete with each other, whereas unsharded domains use fewer connections, and so exhibit lower loss rates.

When measuring `DOMContentLoaded`, as in Figure 5.4(b), the effect of packet loss is less strong, as fewer connections have finished slow-start so early in the page download process. The diagonal exists, but it is less pronounced. Instead the dominant effect is that the performance win achieved with unsharding decreases as the network RTT grows.

It is no coincidence that our analysis seems to hark back to a similar analysis in Section 4.4.3.1, where we compared the performance of HTTP/2 with HTTP/1.1. In a sense, sharding tricks the browser into creating more connections to increase parallelism, and unsharding condenses these connections back down to a smaller number. As a result, all our previous conclusions with respect to reducing the number of connections when moving from HTTP/1.1 to HTTP/2 apply. Under network conditions where the loss rate is comparably high for unsharded HTTP/2, it makes sense to shard and distribute the loss over multiple connections, in order to make better use of the available capacity instead of waiting for a single connection to recover and slowly grow its window again. Moreover, increasing the RTT on the sole connection to an unsharded domain will delay critical resources serialised on this slow connection, a trend clearly visible for the `DOMContentLoaded` case.

5.3.3 PLT Differences

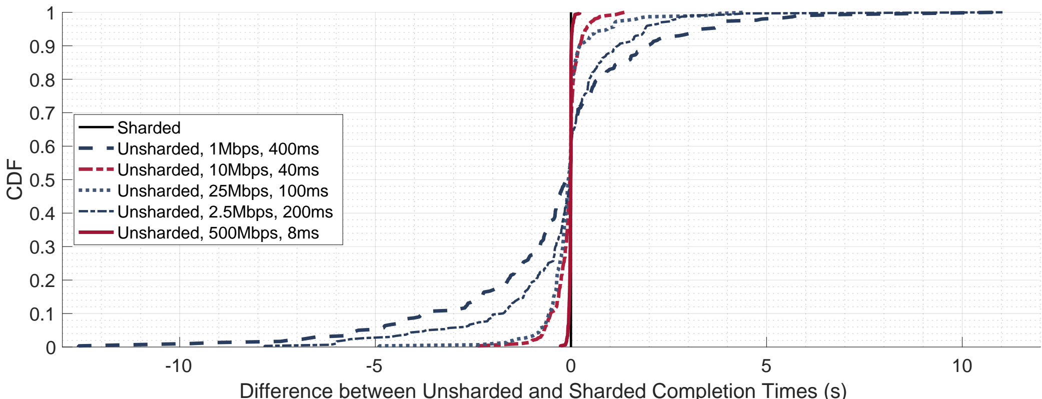


Figure 5.6: CDF of the difference between the mean unsharded completion times and sharded completion times for some typical bandwidth and RTT values.

The previous heatmaps tell us whether unsharding reduces PLT, but not the magnitude of this reduction. Figure 5.6 shows several curves, each corresponding to one of the boxes from Figure 5.4(a). Each curve shows a CDF across web sites. We take the mean load time of the 50 runs for each web site with these network parameters, and calculate the difference between the mean for the sharded and the unsharded domains. For example, a difference of -2.0 seconds

indicates the unsharded web site, on average, finished two seconds faster than the sharded domain.

With high bandwidth and low RTT, unsharded HTTP/2 is nearly equivalent to the sharded case. The difference is so small as to be negligible, as is the case for a bandwidth of 1Gbps and an RTT of 8ms. The 10Mbps, 40ms curve is similar to a fairly typical modern home network, but is right in the region where unsharded HTTP/2 often has a higher packet loss rate. About 20% of sites perform worse when unsharded, but the impact is usually relatively small. The difference is much larger with lower bandwidth and higher RTT, and this is unsurprising as in most other regions the network is often not the bottleneck. These correspond to bandwidths and RTT regimes along the diagonal of the comparison heatmap, where the loss rate is also higher for the unsharded case, *e.g.* for bandwidth of 1Mbps and RTT of 400ms. Naturally, the tails on either end of the CDFs increase as we increase the RTT. In the end, most web sites benefit from unsharding, but some suffer from more than several seconds of additional delay.

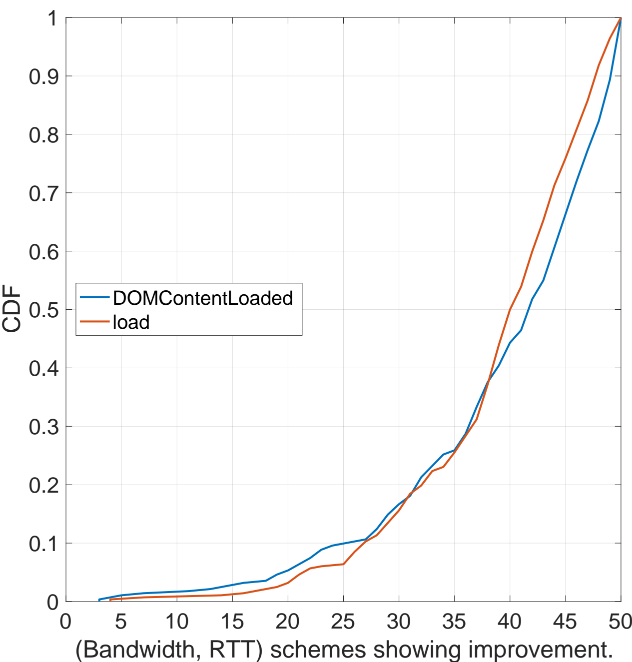


Figure 5.7: CDF across the 283 web sites showing, when we unshard each site, how many out of the 50 *(Bandwidth, RTT)* combinations give better (or equal) performance.

Do some web pages perform consistently poorly when unsharded, irrespective of network conditions? Figure 5.7 shows a CDF across web sites of the number of network scenarios (Bandwidth, RTT) where the web site performs better (or at least no worse) when we unshard it. 50% of web sites perform equally or better when unsharded in at least 40 of the 50 network scenarios. Clearly unsharding is of benefit to most web sites most of the time. This applies, whether we measure by `DOMContentLoaded` or by `load` event. However, almost no web site performs better unsharded in all network conditions, and 10% of web sites perform worse when unsharded in more than half of the network scenarios. These 10% of web sites merit further in-

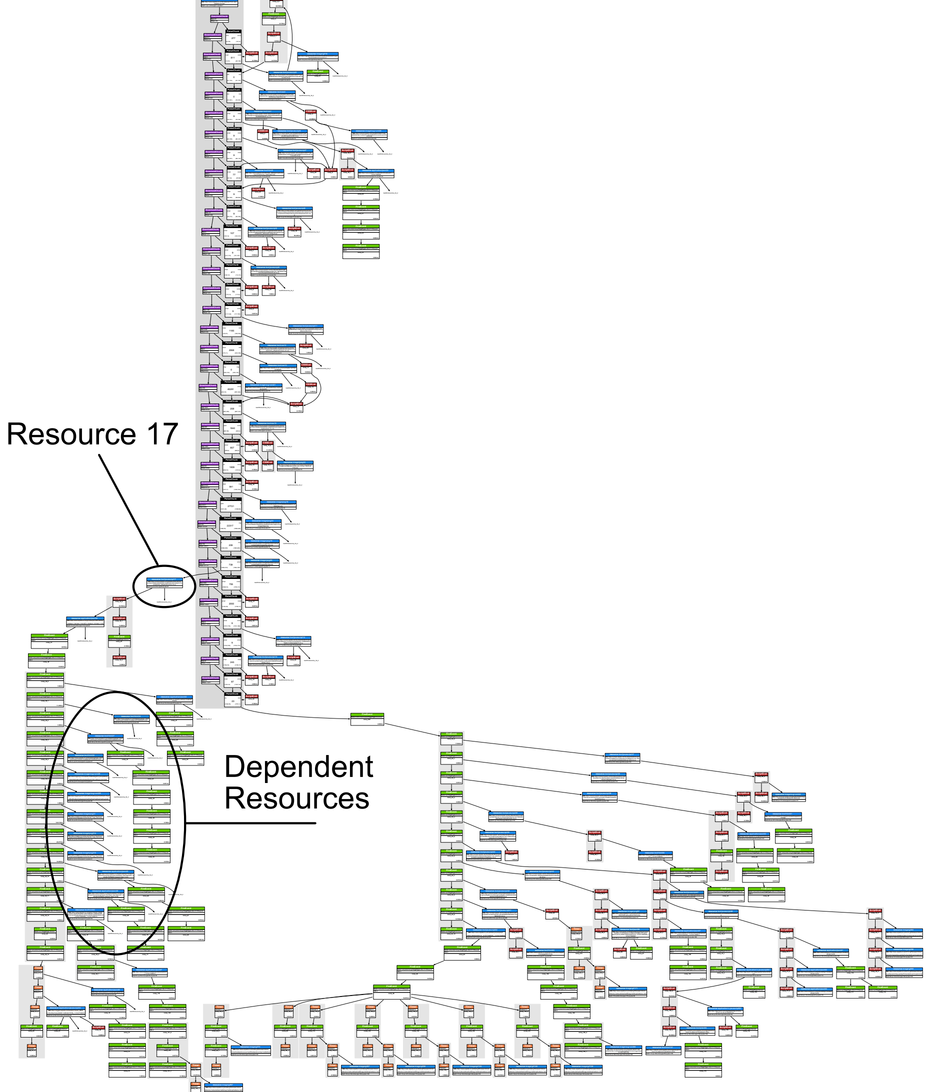


Figure 5.8: Dependency graph for expedia.co.uk.

vestigation to see what they are doing wrong and how their page structure could be improved. In the next section we examine some of these in depth.

5.3.4 Case Studies

A detailed view of how each web site behaves when its domains are unsharded is more nuanced than the high level picture seems to indicate. Whether or not a web site will benefit from unsharding depends greatly on the its dependency graph structure. It is the interplay between a web site's dependency relationships and the transport protocol itself that primarily contributes to the web site's completion time, and we will show examples how this relationship changes when a web site coalesces all its traffic onto one connection.

First, consider a web site from the lower left corner of Figure 5.7. When unsharded, expedia.co.uk performs worse across 46 of the 50 network parameter combinations we tested: it takes longer to complete when compared to the sharded case almost independently of network conditions. To understand this counter-intuitive result, we will show results from a run with a bandwidth of 1Mbps and RTT of 200ms; the effects are more obvious here, though they persist

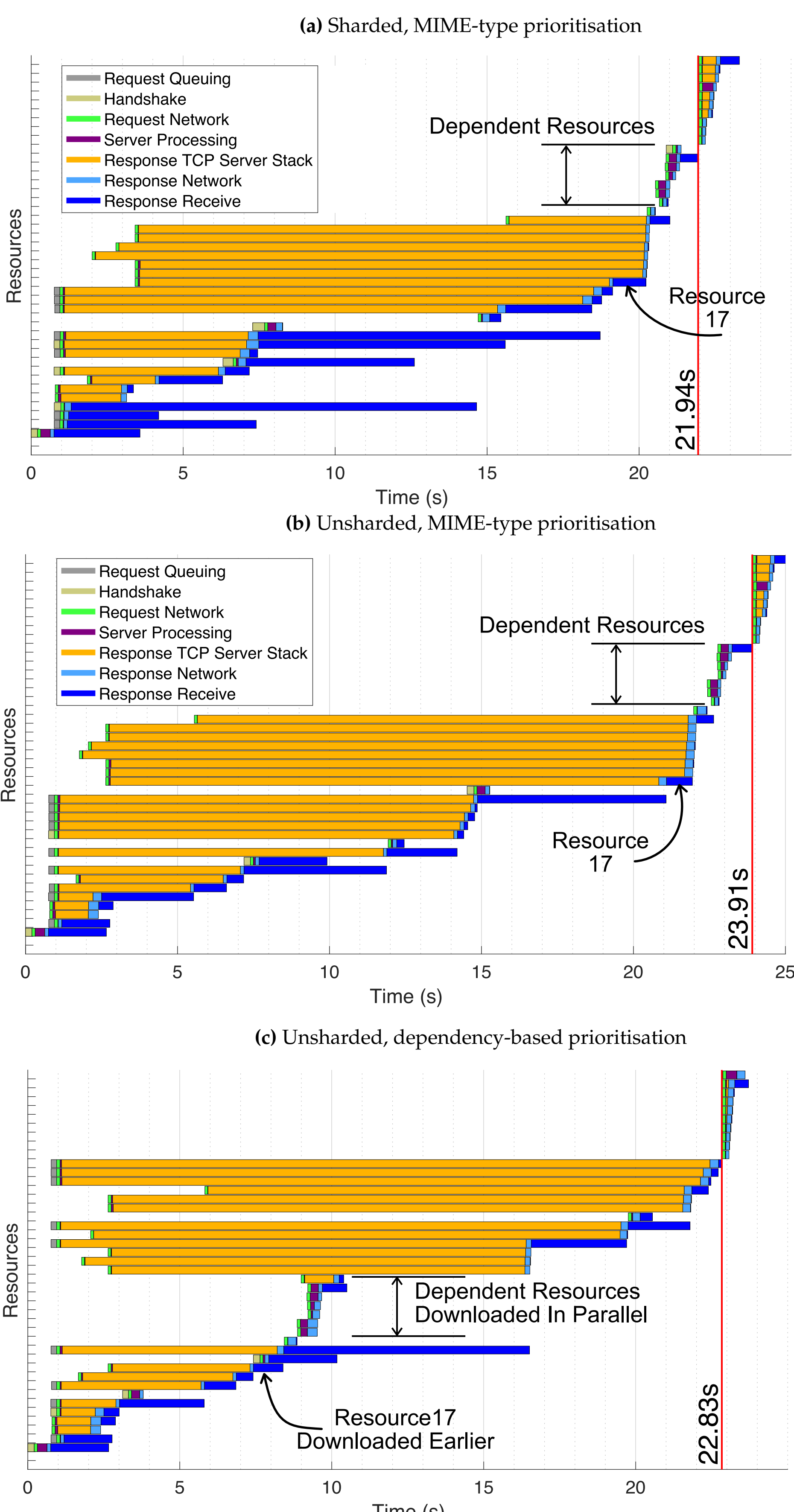


Figure 5.9: Timeline plot of expedia.co.uk downloaded at 1Mbps and 200ms RTT using HTTP/2

under other conditions.

In Figure 5.9(a), we explore where the time loading expedia.co.uk is spent using a timeline plot depicting the components that contribute to page load latency for each downloaded resource. We can now compare the sharded and unsharded timelines for expedia.co.uk as shown in Figures 5.9(a) and 5.9(b). On close examination, we discover that the unsharded version of Expedia suffers because it downloads a critical JavaScript file, Resource 17, later than its sharded counterpart. A quick glance at Expedia's dependency graph, shown in Figure 5.8 shows that Resource 17 is referenced later in the page, and when evaluated triggers the download of several other resources which are necessary for its completion.

Although critical path analysis of the Expedia dependency graph places Resource 17 higher in the priority chain than several of the preceding JavaScript resources, MIME-type prioritisation gives all the JavaScript resources equal weights. Resource 17 is therefore serialised onto the single unsharded connection way too late in the game, which causes a period of idleness while the browser evaluates it. Its children, in turn, are not parallelised with other network downloads. The sharded Expedia, on the other hand, masks this prioritisation error because it moves some of the preceding resources that delay the critical resource onto other parallel connections.

A major conclusion to glean from this example is that the correct prioritisation is crucial for the minimisation of completion time in HTTP/2 in the wake of unsharding. It may be wise to restructure web sites to guard against scenarios like this, where one possible solution is to pull the critical resources earlier in the HTML page. But sometimes it may be hard for web developers to determine which JavaScript files are more important that others without running some form of critical path analysis.

Now that we've identified incorrect prioritisation as the culprit for Expedia's poor performance when unsharded, what happens if we use a more accurate prioritisation scheme? Figure 5.9(c) shows the result of rerunning the same experiment with a critical-path dependency-based prioritisation scheme, as determined by an omniscient observer.

Dependency-based prioritisation pulls Resource 17 earlier, allowing its dependent resources to be downloaded in parallel with another connection, improving load time by nearly two seconds. However the unsharded Expedia is still about a second slower than the sharded version. The reason is not obvious from the Expedia timeline plots, and we must delve into TCP time/sequence plots to understand what is happening. Rather that do this for Expedia, where the effect is subtle, we will instead look at a web site where the symptoms are more obvious. The root cause turns out to be due to a pathological interaction between TCP and web traffic.

5.3.4.1 HTTP and TCP interaction

Consider the case of quora.com, which we downloaded at a bandwidth of 50Mbps and RTT of 40ms, a combination where unsharding outperformed sharding in 80% of the web sites ex-

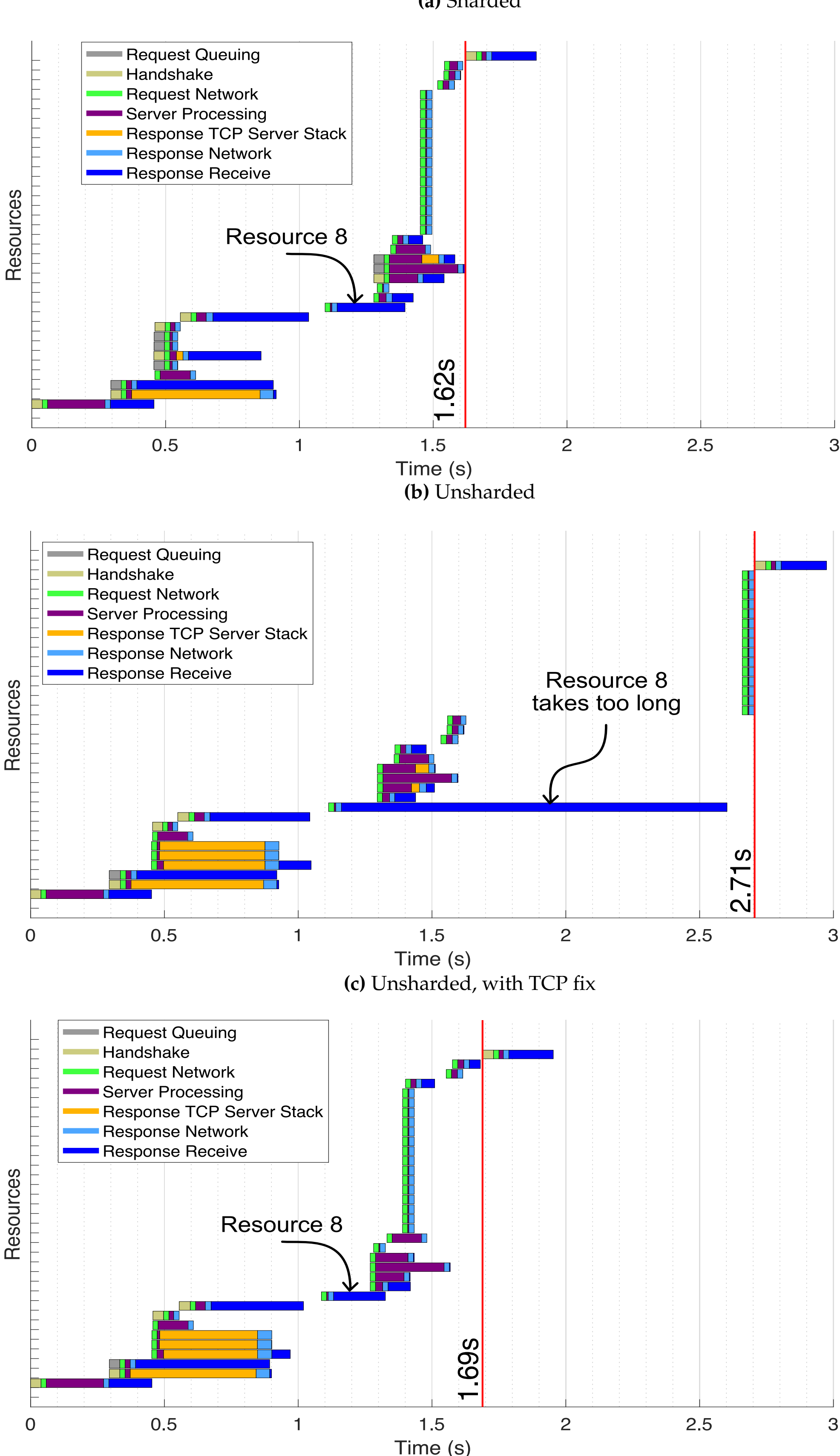


Figure 5.10: Timeline plot of quora.com downloaded over a 50Mbps link with 40ms RTT using HTTP/2.

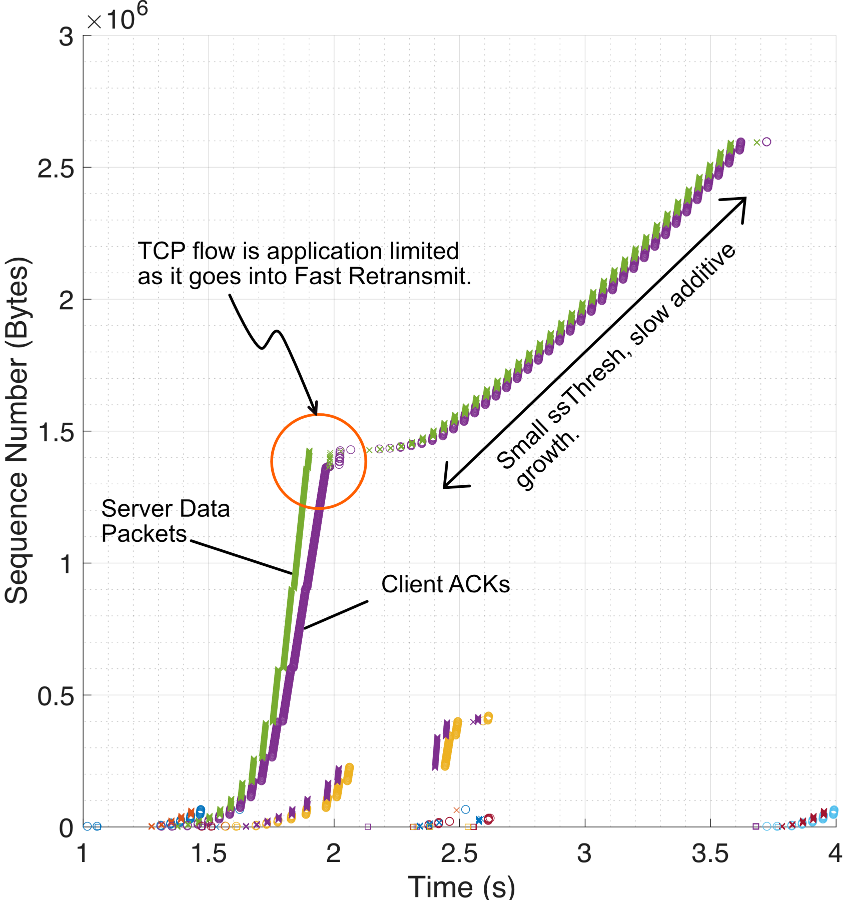


Figure 5.11: Time/sequence plot for unsharded quora.com downloaded on a 50Mbps link with 40ms RTT.

amined. Figures 5.10(a) and 5.10(b) show the timeline plots of the sharded and unsharded downloads of quora.com respectively. From the plots, it is clear that when unsharded, Resource 8 takes a disproportionately long time to complete downloading over the network. The root of this discrepancy can be best explained by looking at the time/sequence plot of the download, shown in Figure 5.11. We observe that after experiencing some packet loss, Resource 8 is sent on a connection that takes a very long time to recover in TCP's additive-increase regime. Although an alternative TCP congestion control scheme like *CUBIC* [62] could reduce this effect somewhat, we would like to understand why TCP is in this situation in the first place.

We discovered from Quora's dependency graph (not shown) that Resource 8 is a CSS file downloaded programmatically by executing some JavaScript. As such, there is a period of time where the connection is idle as it waits on more data from the application which is evaluating the script. At that precise moment, the TCP connection experiences some loss and goes into Fast Retransmit while it is application limited.

This unfortunate series of events would not be too worrisome by itself, but it triggers detrimental side effects as a consequence of how TCP sets `ssthresh` at the point of entry into Fast Retransmit, and how it sets `cwnd` when it exits.

TCP NewReno [132] and, by extension, modern SACK TCPs are conservative and try to

avoid setting cwnd and ssthresh to large values when they are application limited. In this regime a TCP flow does not have a fresh measure of the number of bytes it can safely send without causing loss. Therefore on entering Fast Retransmit, rather than setting ssthresh to cwnd/2, it sets it to be FlightSize/2, where FlightSize is the number of bytes which have been sent and not yet acknowledged at the point the loss was detected. For bulk transfers this is sensible, but data on HTTP connections is often very bursty, as objects are frequently small, so TCP is frequently application-limited, especially as it finishes sending a resource. In the Quora example, only the last six packets of the previous resource are in flight when the loss is detected. TCP enters Fast Recovery, but FlightSize is six packets, and therefore ssthresh is set to three packets. On exiting Fast Recovery, cwnd is set to be min(ssthresh, FlightSize + SegmentSize), as specified in the RFC [132]. As a result, our time sequence plot shows that TCP sets cwnd to one packet on exiting Fast Recovery, since the FlightSize is effectively zero at this point. TCP enters slow-start (cwnd is less than ssthresh), and almost immediately exits due to the small value of ssthresh. Subsequently, the TCP flow is doomed to suffer through a very long period of additive increase starting from a small window; this fails to reach link capacity in a timely manner.

To some extent, the preloader can reduce the likelihood that an HTTP/2 TCP flows is application limited just as it exits slow-start; the preloader masks the severity of this problem in many of our runs. For the particular scenario we've chosen, Resource 8 cannot be preloaded since it is requested as a result of evaluating a JavaScript file.

Ns-3 faithfully implements the NewReno RFC, as does FreeBSD, which is often used for web servers. Linux is less conservative when setting ssthresh and cwnd in Fast Recovery. We re-ran our unsharded Quora experiment, this time setting ssthresh to cwnd/2 on entering Fast Recovery, and setting cwnd to ssthresh on exit. Figure 5.10(c) shows the result, and demonstrates that this change eliminates the pathology that had caused Resource 8 to download too slowly. This approach may, however, have other downsides, and may increase the likelihood of packet loss when exiting Fast Recovery. Sharded domains may also see this pathology, but often their connections see no loss because they don't slow-start to a large enough window. When they do suffer, other connections can often use the bandwidth while the affected connection slowly recovers, reducing the severity.

5.3.4.2 Interactions between Preloader, Prioritisation, and Unsharding

Finally, we would like to decouple prioritisation and preloading from the HTTP/2 download process, and examine how these two components affect the relative performance of sharded *vs.* unsharded versions of a particular web site. We examine gizmodo.co.uk, whose dependency graph is shown in Figure 5.12 because of its interesting structure, where it attempts to download several blocking JavaScript objects (denoted by (1) in the graph) at a relatively late stage during the parsing of its HTML. These JavaScript files block the firing of the page's DOMContentLoaded event, which consumes several milliseconds of processing time and requests

several other resources (denoted by (2) in the dependency graph) before the page completes.

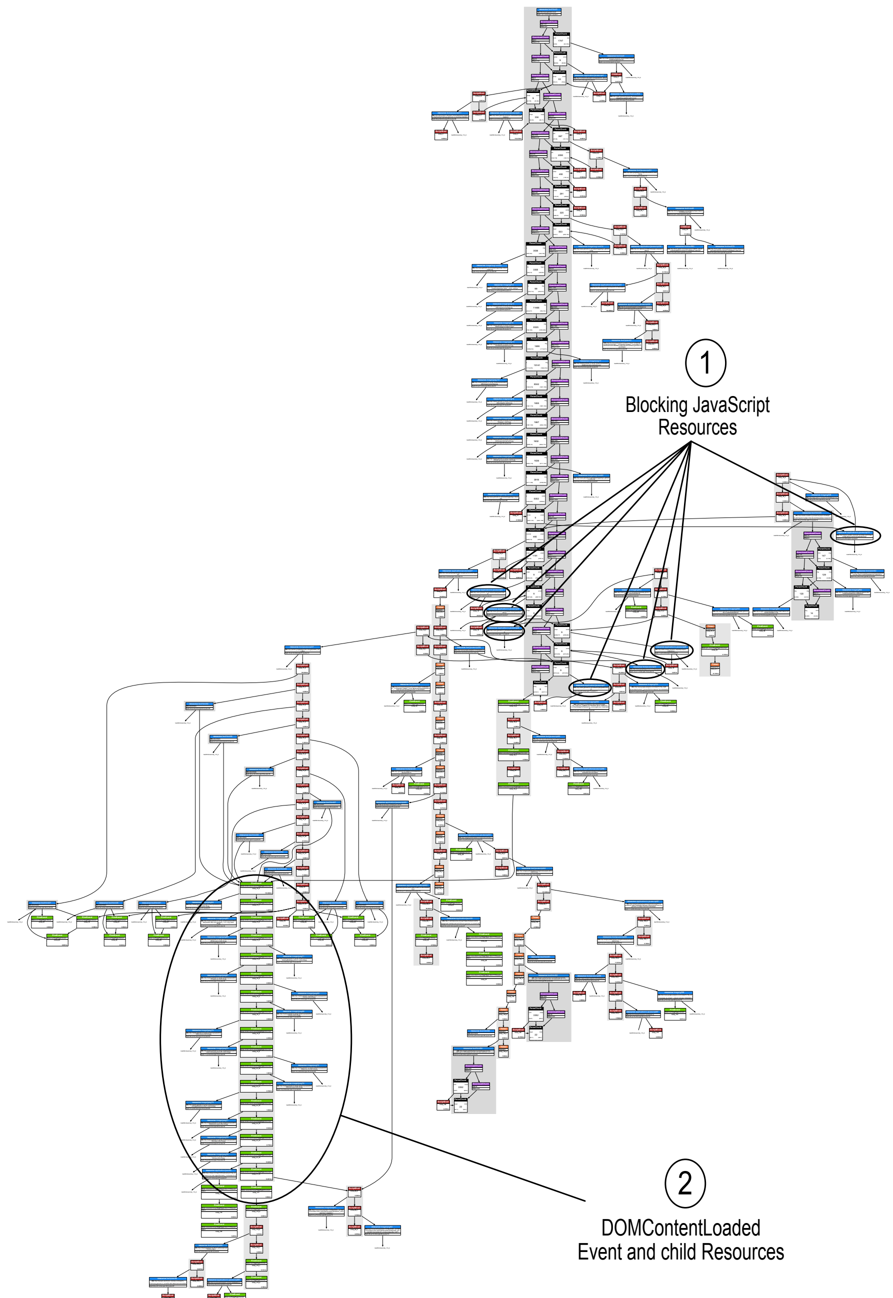
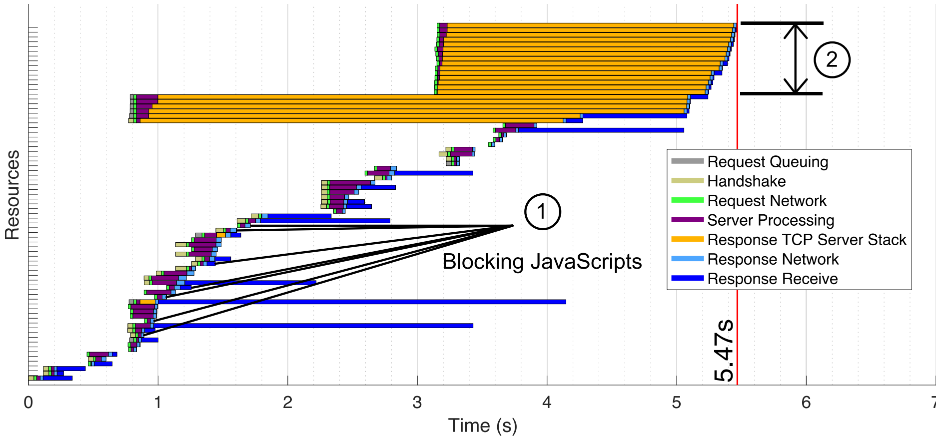


Figure 5.12: Dependency graph generated for gizmodo.co.uk.

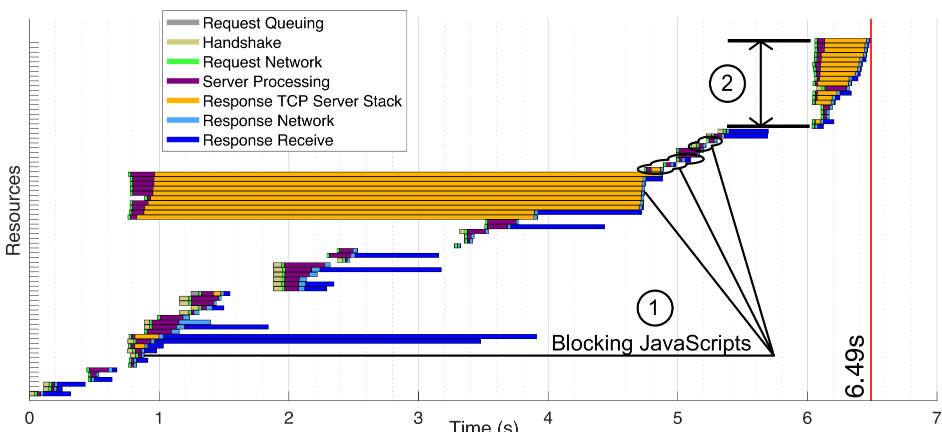
Figures 5.13(a) shows the timeline plot for downloading Gizmodo at 10Mbps, 40ms RTT for a sharded web site running without the preloader. We can clearly see the step behaviour of the blocking JavaScript resources that need to be downloaded in order. Despite the absence of the preloader, the sharded Gizmodo manages to parallelise blocking resource downloads with other downloads and processing delays. It therefore allows the `DOMContentLoaded` event to fire early, requesting the resources marked by (2) without having to idle the network for any period of time.

Figure 5.13(b) shows the same experiment for an unsharded Gizmodo, with no priori-

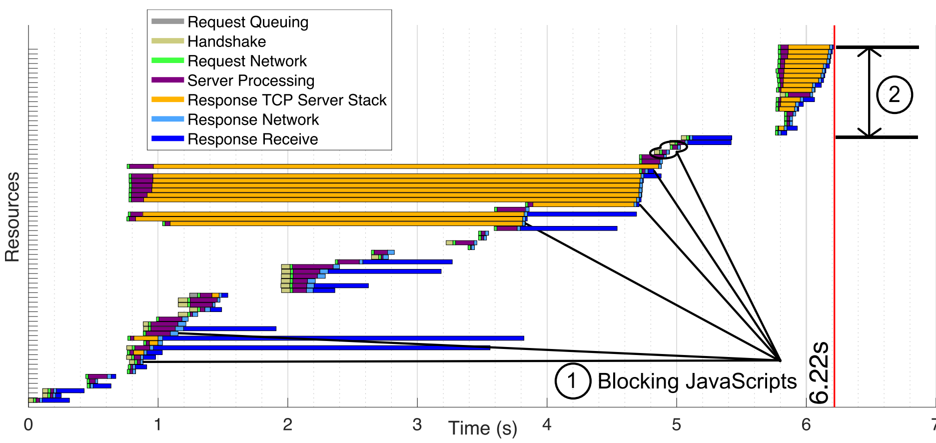
(a) Sharded, No Preload



(b) Unsharded, No Priority, No Preload



(c) Unsharded, Priority, No Preload



(d) Unsharded, Priority and Preload

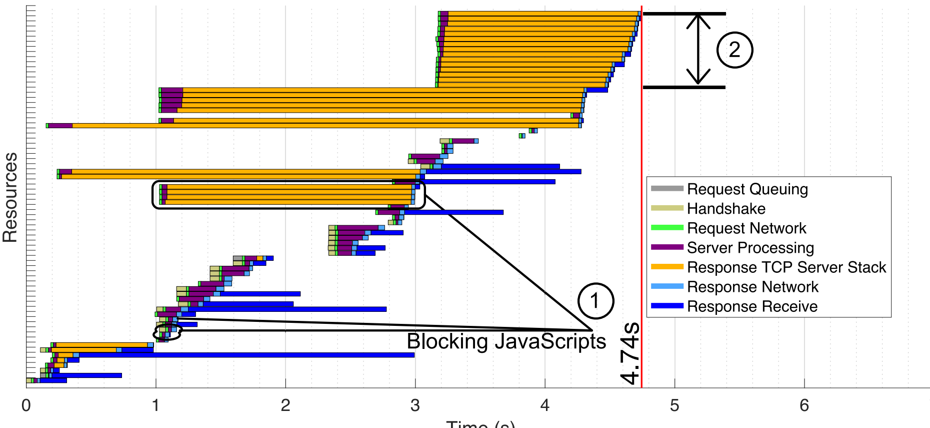


Figure 5.13: Timeline plots for gizmodo.co.uk downloaded on a 10Mbps link with 40ms RTT

tisation and no preloader. In this case, the blocking JavaScript resources share a connection with lower priority objects, and we see the step behaviour once again. Because the downloads happen sequentially, the `DOMContentLoaded` event is pushed to a later time, and the network experiences a period of idleness whilst the event is evaluated and before its dependent resources are requested, which increases the page load time even further.

Prioritisation slightly improves page completion time, as seen in Figure 5.13(c), but only up to a point. Recall that prioritisation happens at the server with respect to resources it has already received requests for. Since the server cannot know about the next blocking JavaScript resources before receiving requests for them, there is little it can do. JavaScript resources still exhibit step behaviour, and there is still an idle period before the resources triggered by the `DOMContentLoaded` event are requested; this hurts page completion time.

Finally, when we add the preloader in Figure 5.13(d), the step behaviour disappears. Requests for all the blocking JavaScripts are sent at once by the preloader, allowing the server to make more informed priority decisions. As a result, `DOMContentLoaded` fires early, preventing the period of idleness we experienced without the preloader. Comparing this final graph with Figure 5.13(a), we see that the unsharded version beats the sharded version after enabling the preloader. Unfortunately, if we also enable the preloader with the sharded site (graph not shown), any benefit disappears. In summary, when we unshard domains for web sites with similar structures to Gizmodo, we need both prioritisation and preloading to be active in order to get the performance gains of HTTP/2 or, in some cases, just to not perform worse. Really the web page structure itself is to blame here, and simple restructuring of the page would likely give better performance.

5.4 Conclusions and Discussion

For most websites, we found that unsharding web domains, a method recommended for improving web sites under HTTP/2, does indeed improve performance most of the time. Nevertheless, even after aggressive unsharding, the PLT reduction on average is not very large. Unsharding seems to help most where the BDP is high, and the network can support a large number of bytes in flight before experiencing any loss, conditions which make placing more bytes onto a single connection largely beneficial. Even after unsharding, third party web sites and ads still prevent HTTP/2 from achieving an even larger win by coalescing connections together.

Although unsharding delivers reasonable performance gains, it does not consistently do so. The network conditions where unsharding works least often also happen to be common network conditions for modern home networks. Under these conditions, unsharding predominantly suffers from the same pitfalls as those which arise from reducing the number of connections per domain. The slow recovery of a lone connection experiencing loss greatly delays the delivery of the web page's critical resources and increases the PLT. If web page designers plan to unshard their domains, they must keep several key factors in mind in order to maximise the

benefit they can achieve.

First, a strong requirement for fast unsharded HTTP/2 is a transport protocol that probes more aggressively for available bandwidth, and recovers swiftly from packet loss. TCP CUBIC, which is already employed as the default transport in most operating systems, seems like a good contender. But even CUBIC can encounter conditions where the congestion window growth is deterred. BBR, with its lower packet loss rate and probing mechanism is an attractive alternative. Out-of-order packet delivery, in the style of QUIC, would help here too.

Second, a good object prioritisation scheme is vital to improving HTTP/2 with unsharded domains. We have shown how web page structures can interact with HTTP/2 to give worse performance when unsharded. To some extent, HTTP/2 priorities can mitigate these problems. Although MIME-type prioritisation seems to get us most of the way, a browser is better served using a form of dependency-based prioritisation scheme. Even then, origin servers are not omniscient, and cannot always see what resources will be required next. Web site designers would do well to profile their sites using the techniques shown in this section, to understand where performance can be lost.

Lastly, to our surprise we found a pathological interaction between part of the RFC covering TCP Fast Recovery and HTTP/2 traffic. Linux, by virtue of not fully implementing the RFC, is not susceptible, but FreeBSD is. How best to update this RFC to avoid this and other unwanted side effects should be the topic of future work.

To close, we identify some caveats to our analysis. Without an in depth knowledge of a web site's server infrastructure, there is no sound way for deducing which sharded domains may be coalesced in practice. We have been aggressive in combining all domains with the same top-level domain name, thereby possibly overstating the gains of unsharding. However, we are less interested in the absolute magnitudes of the possible improvements, but would rather uncover the general trends and cases where unsharding may underperform. With a more conservative unsharding scheme, the limitations we have highlighted will still exist, albeit to a lesser extent.

Chapter 6

Multipath for Web

6.1 Introduction

From previous chapters, we've seen that PLT is expected to decrease as the bandwidth available to the browser increases—higher bandwidth allows a web page's objects to be transmitted to the client faster. This relationship is especially pronounced for traditionally bandwidth-constrained mobile devices such as smartphones. For example in 2011 the introduction of 4G/LTE brought an order of magnitude increase in bandwidth, and a dramatic improvement in PLT compared to the previous slower 3G.

On mobile devices, users often have two network interfaces available, such as WiFi and LTE, but only one of them gets used at a time when accessing the web. Therefore, we could potentially further increase the bandwidth available to a browser by downloading a web page via both interfaces simultaneously, opening up a new avenue for reducing PLT. Multipath TCP (MPTCP) is a state of the art transport protocol that allows the browser to do just that by pooling bandwidth across multiple paths [32, 33, 73].

In theory we should be able to simply enable MPTCP and reap the benefits of added bandwidth and therefore reduced PLT. Surprisingly, though, this is not always the case. Using two paths sums their capacities, but risks incurring the greater of their latencies. Therefore, in many cases, using multiple paths actually *increases* PLT, despite the increase in total available bandwidth. For example, in Figure 6.1 we set up a typical mobile scenario: there are two paths with equal bandwidth, but one of them experiences five times the RTT of the other. We simulated the loading of 300 from the top 500 Alexa web pages with both regular single-path TCP and MPTCP. We observe that in the case of MPTCP 70% of the web pages loaded slower than in the single-path case.

In this particular scenario, the culprit is the mismatch between the delays of the two paths. MPTCP sends web resources on both paths, but some resources are more critical than their counterparts. In the cases where MPTCP underperforms, it picks the higher-delay path for some of the packets that carry these critical resources.

This example is just one of the many ways the interaction between bandwidth and delay can unexpectedly influence PLT in a multipath environment. To make things even more com-

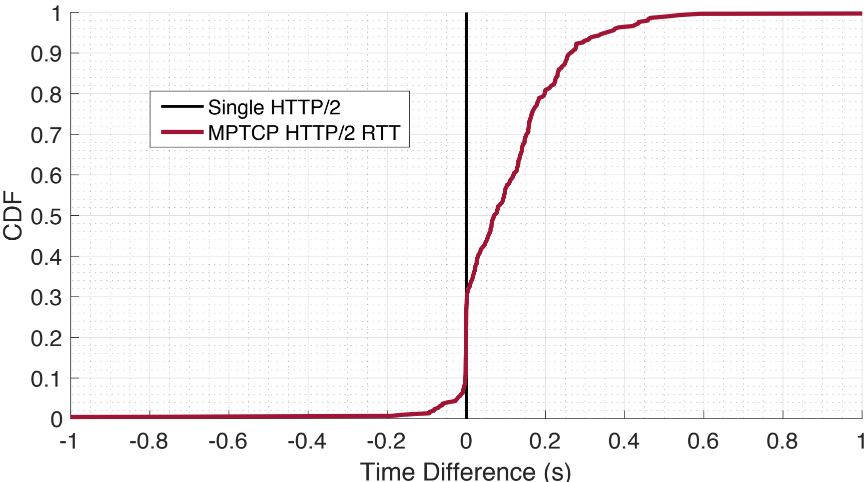


Figure 6.1: CDF of the difference between PLTs of 300 web pages relative to single-path TCP, 50Mbps bandwidth, 100ms RTT on 1st subflow, 500ms on 2nd one.

plicated, web sites are increasingly adopting HTTP/2 as their de facto protocol [24, 25, 140], and more recently, Google has enabled QUIC in its Chrome browser [21,27,28].

How do those new web-oriented transport protocols affect PLT in multipath settings? Are problems like the one shown in Figure 6.1 fundamental, or can we adapt MPTCP to work better with web transfers? These are the questions we answer in this chapter.

First, we present an evaluation of modern web protocols over multipath transport. In particular, we investigate the most common case, where a mobile device has two interfaces and downloads web pages over two distinct paths. We provide a thorough analysis of how protocols like HTTP/2 and QUIC-lite (HTTP/2 with out-of-order delivery, see Section 4.3) behave over multipath for 300 top Alexa web sites and a range of bandwidth and RTT values. We evaluate these scenarios for several two-path topologies where we vary the bandwidth and RTT on the first or second path.

We find that using MPTCP for web traffic can increase PLT due to three basic reasons, which we explore in greater detail in Section 6.3:

- Scheduling packets that carry critical resources on the high latency path. The scenario depicted by Figure 6.1 falls into this category.

- Striping packets from one web resource across multiple paths. MPTCP's in-order semantics can impose extra delay when a resource is striped across two subflows and one experiences high RTT or loss.

- Growing the congestion window too slowly due to MPTCP's linked increases algorithm (LIA). LIA limits subflow congestion window growth in pursuit of fairness. However, with small competing flows, sometimes this leaves a path underutilised.

After we've isolated the cases where MPTCP fails to produce an improvement in PLT, we present several mechanisms to address MPTCP's shortcomings when used for web traffic on

two paths. Most importantly, we design and evaluate a new web-aware MPTCP scheduler. We show that it strictly improves PLT compared to the default MPTCP RTT scheduler, and does no worse than single path TCP operating over the best path. Rather than adaptively switching on MPTCP depending on the scenario, we show that with certain mechanisms in place, switching MPTCP on by default can become an attractive option for reducing web PLT most of the time, whilst doing no harm in the worst case.

Such a web-aware MPTCP scheduler either needs changes to the socket layer, or requires a userspace MPTCP implementation. QUIC is already a userspace stack, and we show that QUIC's properties render it particularly advantageous if modified for use as a multipath transport and combined with our scheduler. In particular, QUIC's out-of-order packet delivery feature was designed to reduce PLT for single path web downloads; we show that it is even more powerful when applied to a multipath setup.

6.2 Methodology

To understand the impact of multipath transport on web performance, we used PCP to measure the page load times of varied styles of web sites under a wide range of network conditions. For these measurements, we chose the web protocol, number of connections per domain, object prioritisation scheme, and server design that had generally produced the best PLTs for the single-path case, as seen in Chapter 4. These choices also reflect the current state of the art, and the direction we expect the web to move in.

We accordingly evaluated single-path and multipath web downloads using HTTP/2 and QUIC-lite running with the typical one connection per domain. Armed with our dependency graphs for the 300 Alexa web sites, we emulated the download of each site using HTTP/2 with critical-path prioritisation and communicating with a server that processes incoming requests out-of-order (Section 3.7.6).

We used an omniscient dependency-based policy that uses the critical path in the dependency graph to assign priorities. This dependency-based policy knows the future, so should outperform any real browser dependency policy. We have shown in Section 4.4.4 that a simpler prioritisation scheme like MIME-type will still perform reasonably well. We choose to run our experiments with the best prioritisation scheme to demonstrate the possible gains, keeping in mind that a less optimal scheme will perform very similarly. In the following sections, we describe the details of the experimental setup PCP used for multipath web downloads.

6.2.1 Topology

We modelled a multipath web download using the topology shown in Figure 6.2, which is an extension of the basic topology we have described in Section 3.7.7.1. The figure depicts a client with two network interfaces, typically WiFi and 3G/LTE, over which it can download a web page using two independent paths. Since the paths are independent, each path has its own individual bottleneck, which will typically occur at the last hop before the client.

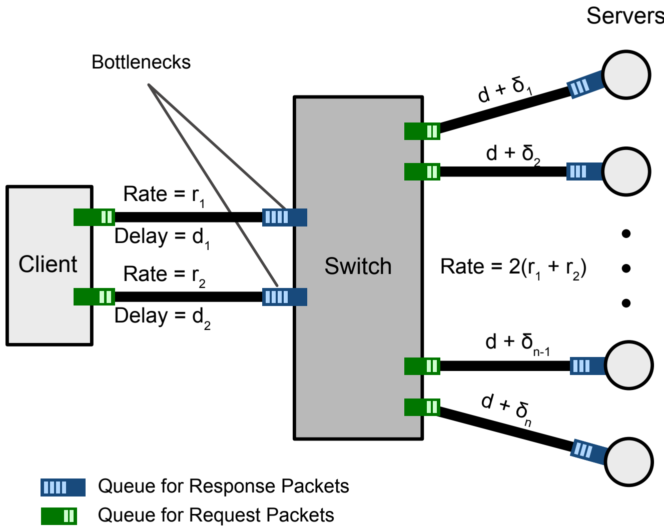


Figure 6.2: Multipath web topology.

The bottleneck bandwidths are r_1 and r_2 respectively, and the one way delays are d_1 and d_2 on the two paths between the client and the switch. Note that the model sets server link capacity to twice the sum of the bottleneck capacities of both paths in order to keep them busy, and prevent a bottleneck from forming at the servers. As before, we sized each bottleneck queue to have slightly more than one bandwidth delay product of buffering for that particular path, to allow TCP to fully utilise the link without building excessive queues.

To account for the variance in results due to transport protocol artefacts and an overly deterministic simulator environment, we jittered the RTT for each run (by adding a random δ on the server link delays sampled from the range $\pm 0.1d$), and used RED queues to add randomness to the packets dropped at the bottlenecks, as described in Section 3.7.7.3. Using this topology, we explored different network conditions by varying the available bandwidth for each of the two paths (r_1 and r_2) and their RTTs (by varying d_1 and d_2).

Once again, we used basic point-to-point connections for the link-layer, instead of modelling precise WiFi and 3G/LTE stacks. As discussed in Section 3.7.7.1, this simplification still manages to capture important aspects of mobile web. The model is reasonable for our target use cases, which assume a fixed bottleneck bandwidth that does not vary in time, and predictable RTT. Long queuing delays in the case of 3G are captured by increasing the RTT on one of the paths. We also assume no outages will occur on either path.

6.2.2 MPTCP in ns-3

At the transport layer, we used MPTCP over the two independent paths, or single-path TCP over one of the paths for comparison. We extended *ns-3* to incorporate our implementation of MPTCP based on work from [141, 142]. We implemented an MPTCP socket such that it maintains the typical socket semantics in accordance with the specifications in RFCs [33, 73, 143],

and is transparent to both the application and IP layers, appearing as a regular TCP socket. As such, an MPTCP connection level "meta socket" forms a new layer underneath the application, which is responsible for creating and managing TCP subflows, as shown in Figure 6.3. We implemented the following MPTCP features in *ns-3*:

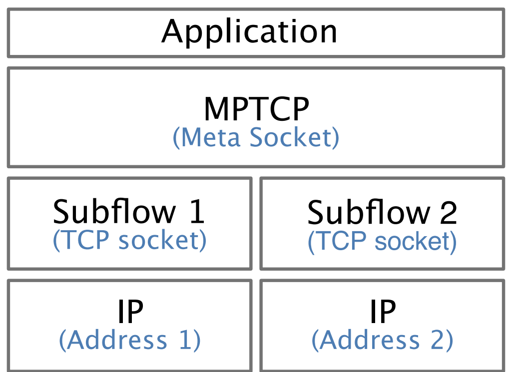


Figure 6.3: MPTCP protocol stack.

- **Connection Initiation and Path Management** An MPTCP connection begins the same way as regular TCP, with a 3-way handshake. To negotiate MPTCP and establish the first subflow, the client and server append an `MP_CAPABLE` option to handshake packets, which also encapsulates authentication parameters. In terms of path management, the client is aware of its multiple interfaces and will establish a new MPTCP subflow on the second interface address using a new 3-way handshake carrying the `MP_JOIN` option. Section 6.3.4 discusses connection initiation in more detail.

- **Data Transmission** MPTCP adds connection-level 64-bit sequence numbers to allow the correct reassembly of segments arriving on multiple subflows when their RTTs are different. These are dubbed Data Sequence Numbers, or DSNs, which elicit corresponding Data ACKs. Data ACKs are part of the flow control mechanism, and indicate the advancement of the receive window at the connection level. MPTCP maintains a mapping between DSNs and the corresponding Subflow Sequence Numbers (SSNs). Whenever data packets are transmitted, MPTCP appends the "Data Sequence Signal" (DSS) option to them. The DSS option contains the packet's DSN in order to correctly map the SSN into the data sequence space. DSS options also carry Data ACKs. MPTCP therefore decouples congestion control, which happens at the subflow level, from flow control, which occurs at the connection level.

- **Buffer Management** Given the dichotomy of sequence spaces between the meta socket and its TCP subflows, we simplified the design by maintaining separate receive and transmit socket buffers at both the connection and subflow levels. Obviously, this design does not match real MPTCP kernel implementations which are memory constrained, but

it works well in the simulator without affecting behaviour. As per the MPTCP specification, we implemented a connection level receive buffer in the meta socket which is shared among the subflows, and is the receive window advertised in the TCP headers. Subflows always pass incoming packets up to meta socket, even if they are out of order, and deal with lost packets relative to their subflow level receive buffer. As for transmit buffers, the meta socket buffers any incoming packets from the application layer. Whenever a subflow notifies the meta socket of available space in its congestion window, the meta socket decides, using the packet scheduler, whether to enqueue a packet to the subflow's transmit buffer so that it can be sent immediately. By enqueueing packets only when a subflow has the option to send them, no proactive buffering occurs at the subflow level transmit buffers. These buffers are used only to deal with lost packets and retransmissions.

- **Loss Recovery** We implemented negative acknowledgements (NACKs) for each subflow independently in accordance with Section 3.7.7.5. A subflow is responsible for recovering its own lost packets, and retransmission must always happen on this subflow to maintain its integrity. But MPTCP additionally enables mechanisms which allow for connection level retransmissions, where segments from one subflow can also be retransmitted on the other in the event of a timeout, for example, or to perform opportunistic retransmission and penalisation (RP, as described in Section 2.3.2). By default, we did not implement this behaviour in our baseline MPTCP. First, there is no standardised algorithm for performing connection level retransmissions in the event of packet loss. Second, our experiments are not receive buffer limited, which removes the need to use RP. We later describe a mechanism for performing connection level retransmissions as an optimisation on top of baseline MPTCP and examine its benefits.

- **Congestion Control** We implemented the Linked Increase congestion control on each subflow, as described in Section 6.2.3.

- **Packet Scheduling** Whenever a new packet arrives at the meta socket from the application layer, or when a subflow receives an ACK that frees up some space in its congestion window for a new packet, the meta socket must make a choice: which of its available subflows will it use to send the next packet? We would like to examine several scheduling algorithms to accomplish this packet assignment. Therefore the meta socket uses plugable packet schedulers to determine which subflow should send first. By default, MPTCP uses an RTT scheduler which picks the subflow with the lowest RTT to transmit on first.

- **Congestion Window Validation** We implemented congestion window validation for each MPTCP subflow (see Section 3.7.7.6). Since MPTCP flows are more application limited than their single-path counterparts due to the splitting of resources across two paths, there is a higher chance of loss caused by stale un-updated congestion windows

when a connection resumes after some idle time. Therefore, congestion window validation plays a more crucial role for MPTCP. Congestion window validation has two components: checking for idle time and determining whether a flow is application limited (Algorithms 11 and 12 in Appendix C). For MPTCP, special care must be taken when performing the second action. For regular TCP, we perform the application limited check after transmitting each data packet. But for MPTCP, although a subflow may appear to have untransmitted segments immediately after transmitting a packet, it may become application limited later, when a second subflow transmits all the remaining data in the connection level socket buffer. The subflow cannot tell whether the remaining bytes in the transmit buffer will be scheduled on the other subflow in the future. Therefore, the meta socket itself should check if either of its subflows are application limited *every* time it schedules a packet on a particular subflow. The check for idle time, on the other hand, should still occur for each subflow separately.

6.2.3 MPTCP Congestion Control

To motivate the design choices behind multipath TCP's congestion control, recall that TCP uses window-based congestion control composed of additive increase and multiplicative decrease [144]. To probe a connection for its capacity, TCP uses additive increase to grow its congestion window at a rate of one segment per RTT. On the other hand, when TCP experiences a loss, it halves its congestion window using multiplicative decrease. In other words:

- For each ACK received, TCP increases its window by $1/w$.
- For each loss, TCP decreases its window by $w/2$.

At start of a connection, or after experiencing a timeout, TCP grows its congestion window exponentially using slow-start. More recent variants of TCP such as CUBIC try to recover faster from losses, and ramp up quicker when network is underloaded [62].

With TCP's basic operation in mind, MPTCP's congestion control algorithm was designed to accomplish following goals [32, 73]:

Be fair to TCP flows on bottleneck link. MPTCP should not consume more capacity than a single TCP flow on the bottleneck link. Therefore, during additive increase, the growth factor for each subflow should be weighted by a tunable factor a so as to take some fixed fraction of the bandwidth that regular TCP would take. MPTCP adjusts the aggressiveness of a to account for the RTT differences between paths.

Choose efficient paths. MPTCP moves traffic from congested paths to less congested paths. To accomplish this feat, it couples the subflows' congestion windows. In other words, during additive increase each ACK grows a subflow's window by $\frac{a}{w_{total}}$, a proportion of the total window across all subflows.

Adapt when congestion changes. For MPTCP to adapt to change, it must continue to probe all available paths for bandwidth. Therefore, MPTCP keeps a moderate amount of traffic on

each path, but places more traffic on the less congested path. To achieve this goal, instead of setting a subflow's congestion window to be half of the total window during multiplicative decrease, MPTCP simply halves the subflow's congestion window instead.

Don't oscillate. MPTCP shouldn't shift traffic from one path to another if it encounters small, transient congestion changes.

Combining all these goals together, we arrive at MPTCP's Linked Increase Algorithm (LIA), which performs the following:

- For each ACK received on subflow r, increase the window w_r by min$(a/w_{total}, 1/w_r)$.
- For each loss on subflow r, decrease w_r by $w_r/2$.

Where a is given by the following equation:

$$a = w_{total} \frac{\max_r w_r/\mathsf{RTT}_r^2}{(\sum_r w_r/\mathsf{RTT}_r)^2} \tag{6.1}$$

MPTCP performs slow-start on each subflow independently, growing the congestion window of each subflow by one segment for each ACK received.

6.3 Performance of MPTCP for Web

We show in this section the results of emulating 300 web site downloads with MPTCP and single-path TCP as a baseline. We picked seven different bandwidth levels between 500Kbps and 1Gbps and four different delay values between 8ms and 200ms that are representative of the range of network conditions experienced by web traffic today. For each of our 300 websites we performed 50 runs for each *(Website, Bandwidth, RTT)* combination to ensure the results are statistically significant, requiring $300 \times 4 \times 7 \times 50 = 420,000$ emulated page downloads for each of single-path TCP and MPTCP. For each download we recorded PLT as the time it took before the `load` event fired, indicating the page had loaded.

6.3.1 Equivalent Paths

The first scenario that we examine is when both subflows have the same RTT and the same bottleneck bandwidth. Figure 6.4 displays a heatmap of the *(Bandwidth, RTT)* values we used. In each cell we record the quantity $\frac{\sum(PLT_{SP} - PLT_{MPTCP})}{\sum PLT_{SP}}$ — *i.e.* the percentage reduction in mean PLT across all downloads when using multipath instead of single path. For both cases, the pages were downloaded using HTTP/2 with one connection, critical path prioritisation, and running over MPTCP. The values in all cells are positive, indicating that when the two subflows experience similar network conditions, using multiple paths is beneficial in all cases.

For low bandwidths under 10Mbps, the heatmap confirms our expectations: adding an extra path halves the PLT. The web pages for these relatively low rates are bandwidth limited, therefore adding extra capacity will make each resource download faster. As the rates increase, the improvement decreases, as web pages switch from being bandwidth limited to

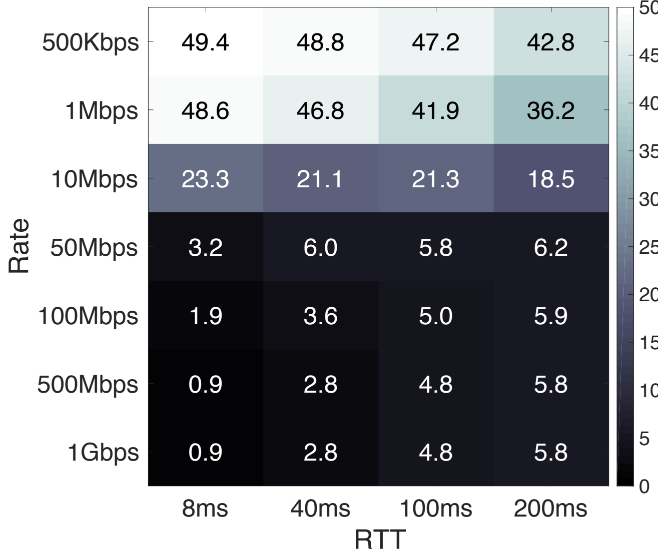


Figure 6.4: Equivalent paths: heatmap of the mean percentage reduction in PLT for HTTP/2 running over MPTCP compared to HTTP/2 running over single path TCP.

being latency bound. When the bandwidth is no longer the bottleneck, we can only get minor improvements as we increase the rate. Therefore, pooling bandwidth over multiple paths may not derive as large a win as one might hope for higher rates.

For bandwidths where web downloads are latency bound (*e.g.* 500Mbps), it is surprising that multipath still achieves a win at all. There is no big difference in PLT for a single path doubling its rate from 500Mbps to 1Gbps, since in both cases TCP flows spend most of their time in slow-start due to a very low loss rate. But for multipath, we have two 500Mbps subflows operating in parallel. Having two paths sometimes allows the same resource to be transmitted in one fewer RTT in slow-start, depending on how the object size aliases against the congestion window size. Even though, in the worst case, the the second subflow may have begun slow-start two RTTs behind the first (see Section 6.3.4), a large object can be transmitted in fewer RTTs if is it striped across both paths. For each connection to a unique domain, we may save one additional RTT by using multiple paths, a fact which explains why we still get an improvement. As we move from left to right in the lower half of the figure, the improvement increases as path RTTs increase.

If multipath is enabled for web downloads, adding an extra path of equivalent bandwidth and RTT represents the best case scenario in terms of possible PLT reductions. From the heatmap, we can conclude that enabling multipath is in fact a worthy venture, despite some added complexity required to support it. First, for bandwidths between 1Mbps and 50Mbps, which fall in the range of typical DSL and 3G/LTE deployments, multipath achieves a sizeable reduction in PLT over single path. If we consider the global average speeds from Akamai's State of the Internet [138], the average speed of 7.2Mbps for non-mobile traffic is close to our 10Mbps data point. As we can see, using MPTCP with equivalent paths may lead to an appri-oximately 20% reduction in PLT at the global average speeds. But even for higher bandwidths,

where MPTCP can only achieve up to 6% average improvement over the single path case, the reduction in PLT is still large enough to justify enabling multipath, since similar improvements have prompted a switch in protocol suites in the past.

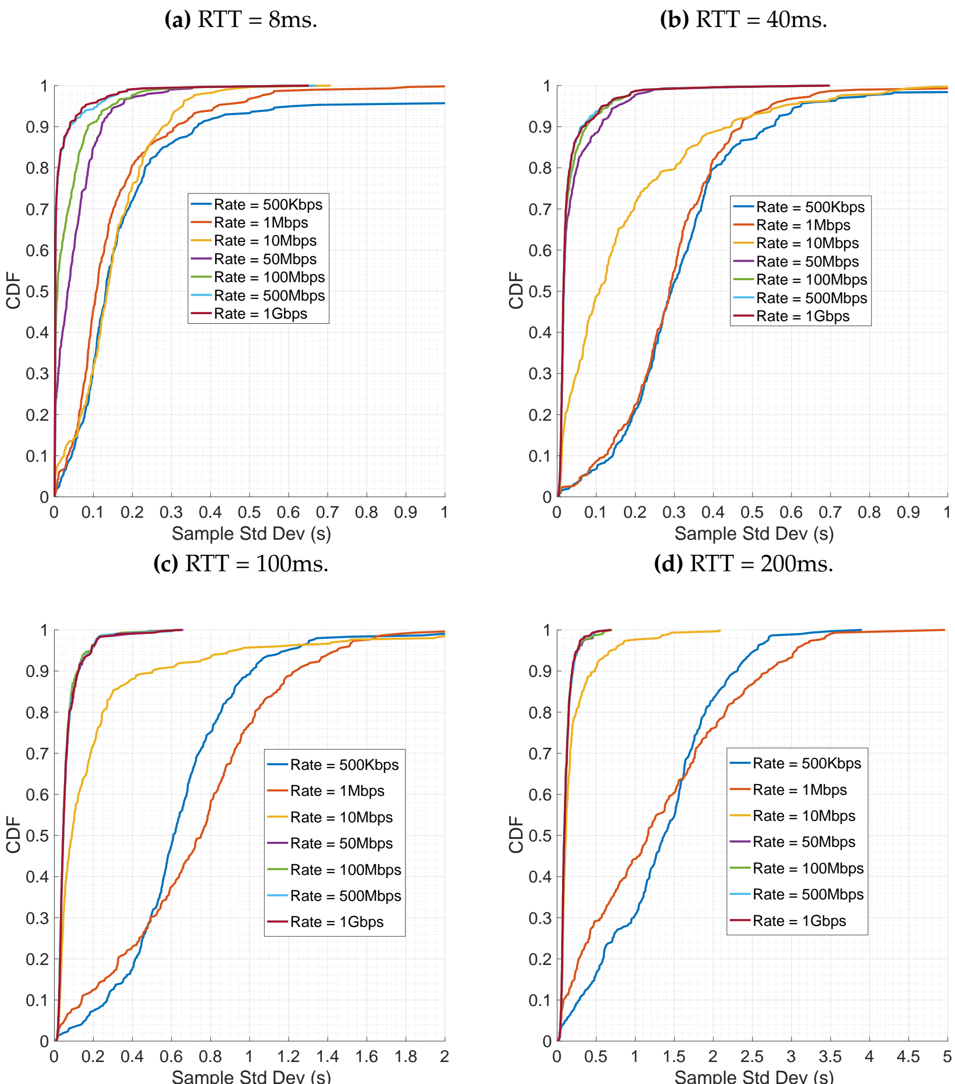


Figure 6.5: For equivalent path MPTCP, CDFs of the sample standard deviation of the PLTs of each web site over 50 runs for the different bandwidths, and RTTs.

To get a sense of the variance within the 50 runs for each combination of *(Website, Bandwidth, RTT)*, Figure 6.5 plots the CDFs of the sample standard deviation for 300 web sites across 50 runs. Higher bandwidths show little variance of PLT across runs, since flows spend most of their time in slow-start. Lower bandwidths, like 500Kbps and 1Mbps, experience random packet loss, and, depending on where the loss occurs, some runs will have shorter PLTs than others due to faster recovery time, eliciting results with higher standard deviations. The higher the RTT, the longer it takes to recover, thus increasing the standard deviation.

6.3.2 RTT Mismatch

While we generally expect HTTP/2 to perform well over MPTCP if the RTTs and the bandwidths of the two paths are comparable, we've seen from the beginning of this chapter that a large mismatch in RTTs may in fact increase PLTs with respect to single path TCP on the lower-RTT path. To further examine this effect, we repeated the last experiment, but this time we set the second path's delay to be $5\times$ that of the first path. Whereas Figure 6.4 showed mean improvement, perhaps a more useful summary metric for the RTT mismatch experiment is the *fraction of sites* that download faster. Figure 6.6 shows the percentage of websites (out of the Alexa 300) which either reduce their PLT in the multipath case with respect to TCP on the "best" path, or maintain it for each *(Bandwidth, RTT)* combination. We allow for a margin of 20ms when comparing whether single and multipath PLTs are equal to account for variation from random packet drops across runs.[1]

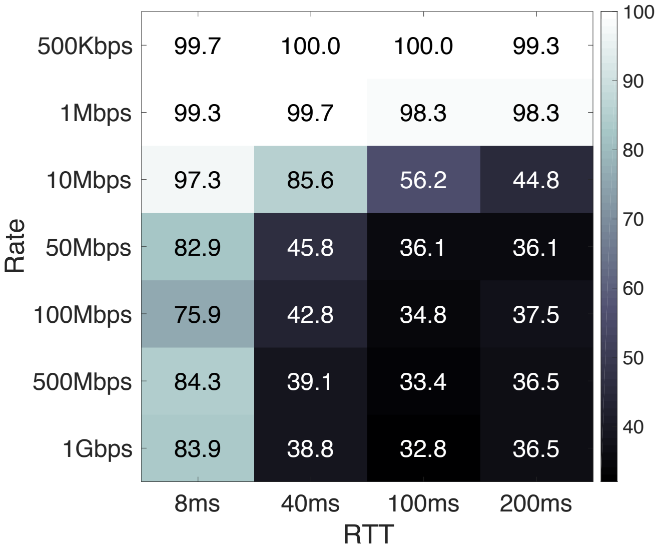
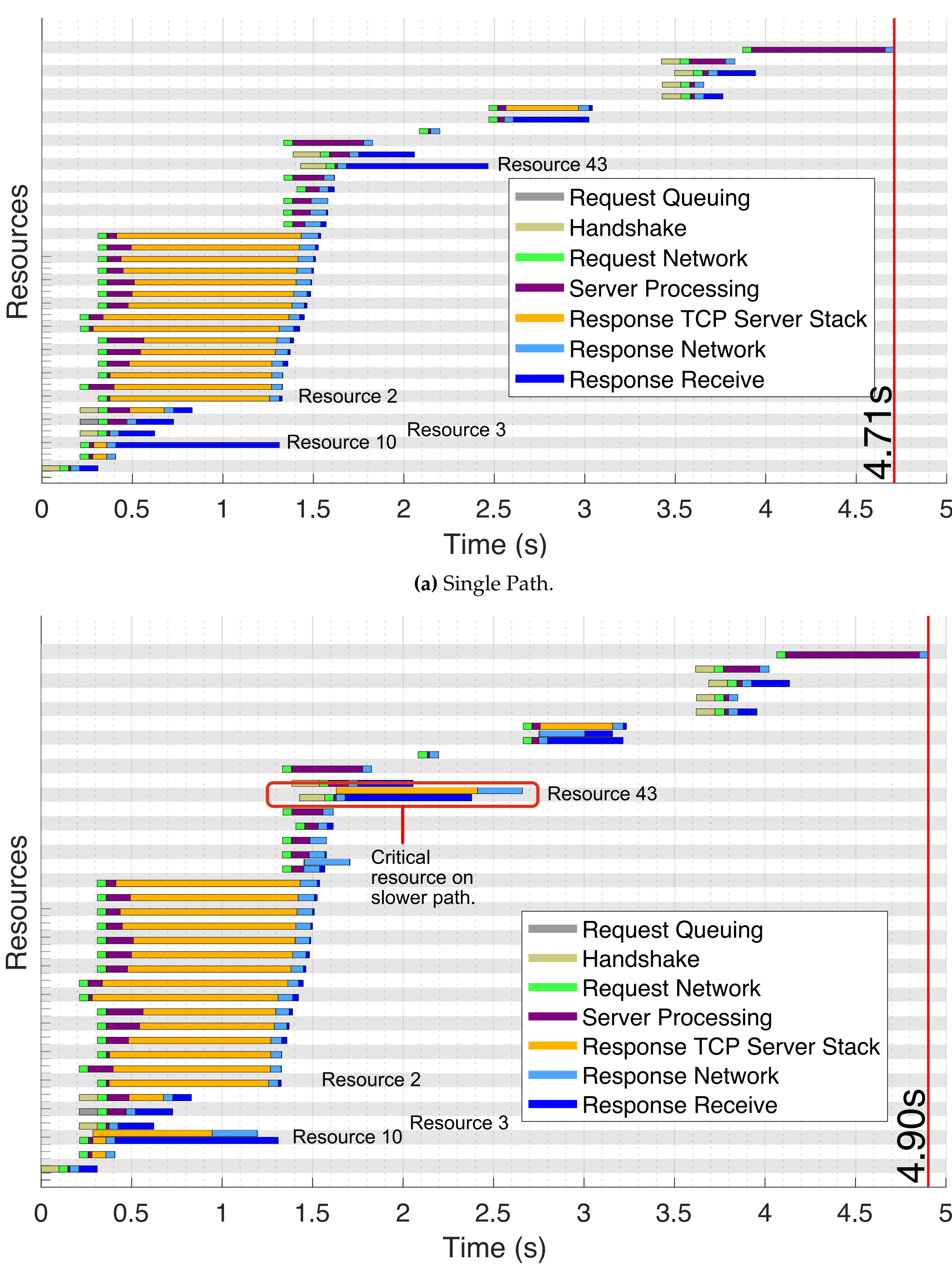


Figure 6.6: $5\times$ RTT on second path: heatmap indicating the percentage of web sites with equal or better PLT using MPTCP vs single-path TCP.

Once again, adding a second path helps decrease PLT in low-bandwidth cases, where web sites are bandwidth limited (*e.g.* less than 10Mbps). When we move away from the bandwidth limited region, we find that adding a second high delay path is actually detrimental. We can see that for more that half of web sites in this regime, we are actually better off disabling MPTCP and picking the single path with the lower RTT. Further, there is a trend of decreasing benefit as we move from left to right and the RTT increases.

To understand why using a higher delay path hurts PLT we focus on a single web site (pages.github.com) and in Figure 6.7(b) we plot the components which contribute to page load latency for each downloaded resource on a timeline plot, with multipath subflows for each resource grouped together in a shaded band. If we compare the timelines of pages.github.com when it is transmitted using single-path TCP (Figure 6.7(a)) *vs.* MPTCP, we find that the web

[1] 20ms is well within the PLT standard deviations across the 50 runs, and is smaller than human reaction time.



(a) Single Path.

(b) Multipath with 5× RTT on second path.

Figure 6.7: Timeline plot of pages.github.com downloaded over a 50Mbps link with 100ms RTT using HTTP/2.

page's PLT increases under MPTCP. Unsurprisingly, MPTCP's default RTT scheduler places very few resources on the second, slower path, especially since the RTT of second path is half a second. In addition, the 3-way handshake to set up the second subflow on the slower path often takes longer than the time it takes to send the full resource on the first subflow. Nevertheless, some resources are large enough to make it onto the second path. In fact, packets from the critical Resource 43 are scheduled on the high delay path, increasing the completion time.

We observe the same problem in all cases where we have large RTT mismatch and single-path TCP outperforms MPTCP—page load times are likely to increase if the MPTCP scheduler places packets from a critical resource on the slower path. To a lesser extent, web traffic transmitted over subflows with a mismatch in RTTs will suffer from extra delays due to reordering. Segments from the faster path will have to wait in the client's receive buffer until segments from the slower path show up. As such, MPTCP will delay the delivery of completed resources to the application layer because of TCP's in-order semantics.

6.3.3 Bandwidth Mismatch

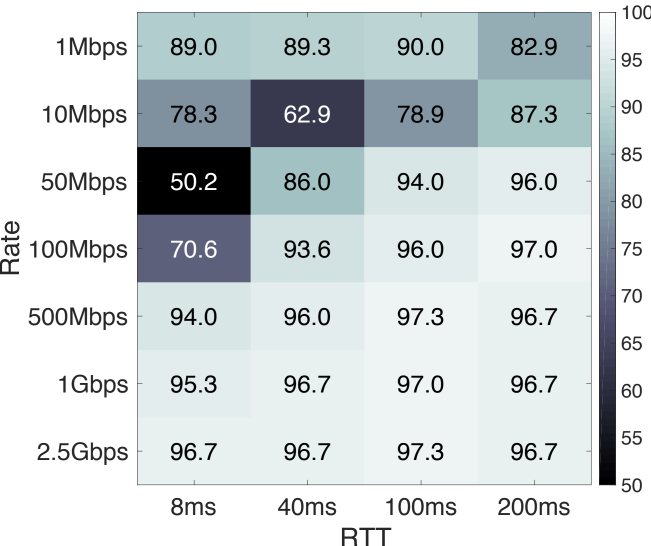


Figure 6.8: $\frac{1}{5}$ bandwidth on second path: heatmap indicating the percentage of web sites with equal or better PLT using MPTCP vs single-path TCP.

After demonstrating how delay mismatch hurts multipath PLTs, we now turn to a scenario where the two paths have the same delays, but different bandwidths. We repeat our experiment, but this time we reduce the bandwidth of the second path to be $\frac{1}{5}$ that of the first path. The results are shown on Figure 6.8 where the vertical axis shows the bandwidth of the higher bandwidth path. We compare against single-path TCP with the higher bandwidth value.

The lower right region of the heatmap shows cases where both subflows have enough capacity for the web page downloads to be latency bound. These cases are similar to the high-bandwidth ones from Figure 6.4—most TCP transfers complete in slow start and adding a second path does not change that, as the second path has high enough bandwidth.

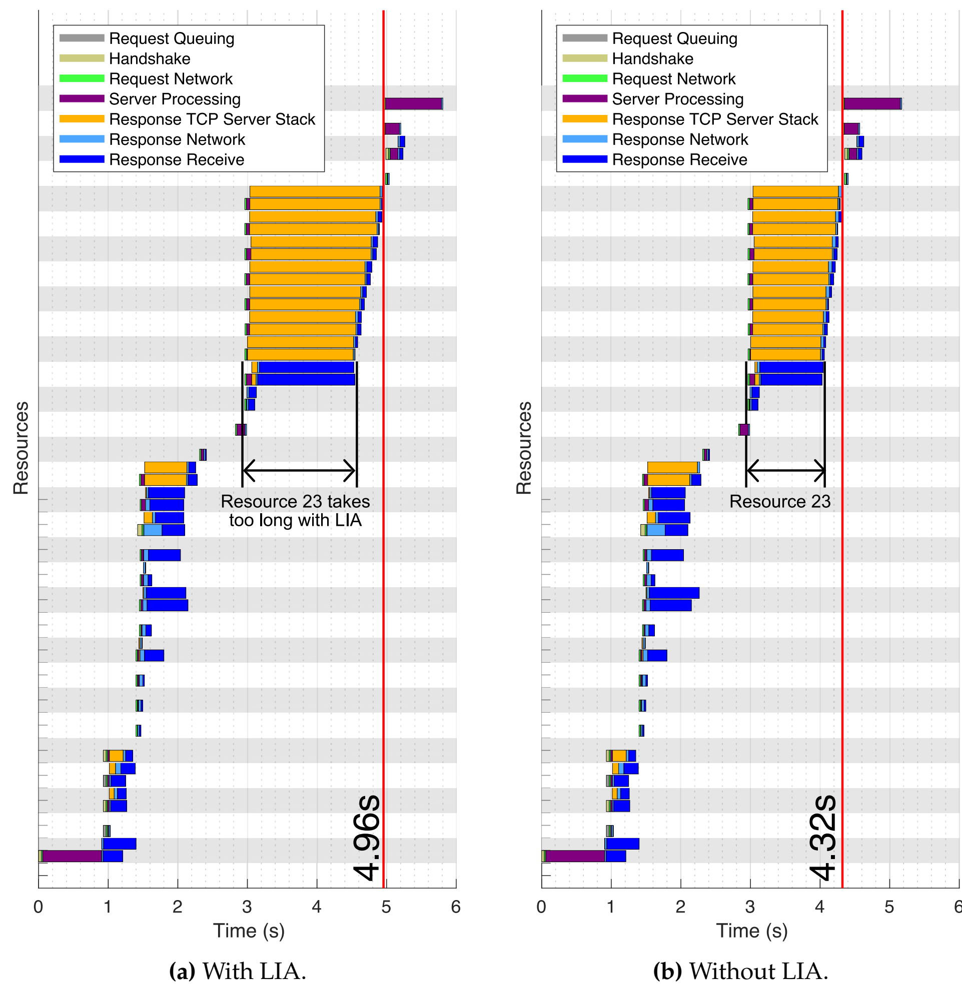
6.3.3.1 Artefacts of Linked Increase

In the upper region the bandwidth is low enough to experience losses. With MPTCP those losses are a lot more likely to occur on the second path, which has a fifth of the first path's bandwidth. The losses trigger MPTCP's LIA congestion control, described in Section 6.2.3. Both subflows slow-start independently but, after a packet loss, LIA considers the total congestion window on both subflows in the congestion avoidance phase. For each subflow, LIA reduces the AIMD additive increase constant proportionate to that subflow's fraction of the total congestion window. There are two potential problems here. As web pages are downloaded from multiple servers, self-competition often occurs. A new connection may start after a previous one has already filled the link, so the new connection suffers a loss and exits slow-start early. When this happens on the low-bandwidth path, which is more common, LIA will often already have achieved a large window on the high-bandwidth path; LIA then increases cwnd very slowly on the low bandwidth path, effectively adding almost no additional bandwidth—MPTCP doesn't hurt here, but it fails to add capacity. When the loss happens on the high-bandwidth path, the low-bandwidth path still has sufficient window to reduce the AIMD additive increase constant significantly, slowing the window increase. In this case MPTCP's LIA can actively hurt.

Let us consider a concrete example: Figures 6.9(a) and 6.9(b) show timeline plots for groupon.com downloaded with and without LIA enabled at a *(Bandwidth, RTT)* where we note a drop in MPTCP performance with respect to single path TCP. We observe that with LIA enabled, the critical Resource 23 takes longer to download than in the case where LIA is disabled and regular AIMD is used instead. We examine the time sequence plots for groupon.com downloaded using LIA in Figure 6.10 to investigate further. Resource 23 is split across the two subflows in both Figures 6.10(a) and 6.10(b). If we compare the gradient of Resource 23 on the first path with earlier subflows on this path, we can see that the window growth is quite slow and doesn't achieve the capacity of the link fast enough. To understand why, we zoom into the beginning of the subflow which transmits Resource 23 in Figure 6.10(c). The subflow loses all of the segments in its initial window, and immediately times out. The congestion window of this higher bandwidth subflow is therefore set to one segment, and it never gets a chance to ramp up with slow-start before entering the additive increase phase.

Suppose the subflow congestion windows are w_1 and w_2. Without any other subflows sharing the link, at equilibrium we expect $w_1 = 5w_2$ if the second path has $\frac{1}{5}$ the bandwidth. Therefore the subflows on paths 1 and 2 should respectively increase their windows by $\frac{5}{6}a$ and $\frac{1}{6}a$ each RTT. But, since the subflow on path 1 has reduced its congestion window to one segment, it will have to increase its window by $a/(1 + w_2)$ each RTT. Note that the larger the value of subflow 2's congestion window w_2, the smaller the increase in subflow 1's congestion window.

Due to early self-congestion on the first path, LIA has effectively shifted traffic from the



(a) With LIA. **(b)** Without LIA.

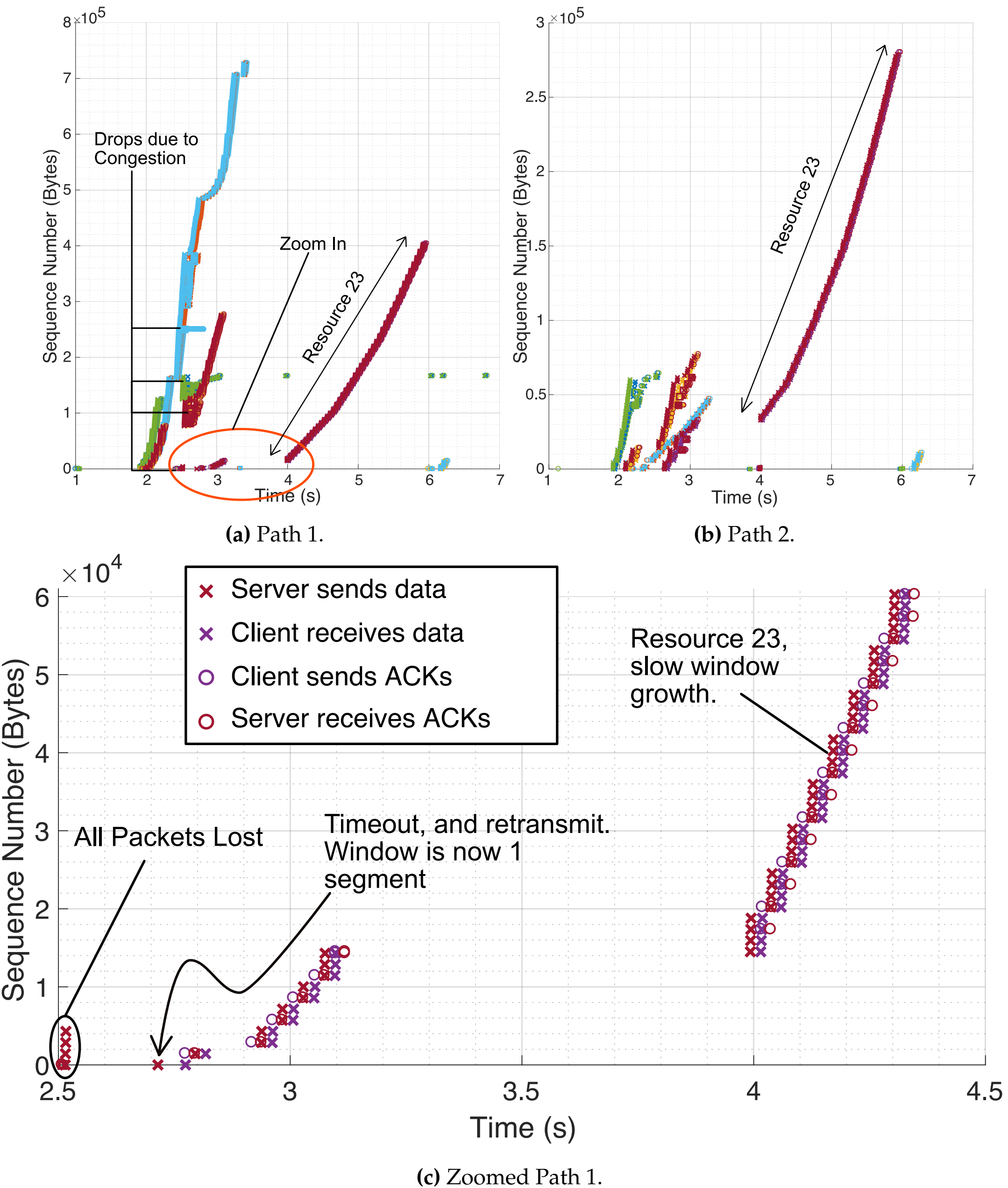
Figure 6.9: Timeline plots for groupon.com downloaded using MPTCP with a bandwidth of 10Mbps and RTT of 40ms, with $\frac{1}{5}$ bandwidth on the second path.

high bandwidth path to the low bandwidth path, which will experience more loss in the long run. Even though this congestion disappears from the high bandwidth path as resources complete, we've shown that LIA fails to grow its window fast enough to recover the available capacity.

6.3.3.2 Tail Loss

LIA alone does not explain why MPTCP performs worse than single path TCP for the top of the leftmost column in Figure 6.8, where the RTT is 8ms. As we've seen, web transfers are often application-limited, consisting of multiple resources with only a handful of packets each. In the case of a packet loss at the end of a resource, when there are not enough packets in flight to trigger fast retransmit, TCP will suffer a retransmit time out. This is a well known problem with TCP, known as tail loss [145].

What is tail loss? An example of tail loss can bee seen in Figure 6.11. The figure shows a time sequence plot for amazon.co.uk downloaded using MPTCP with $B_1 = 10$Mbps, $B_2 = 2$Mbps

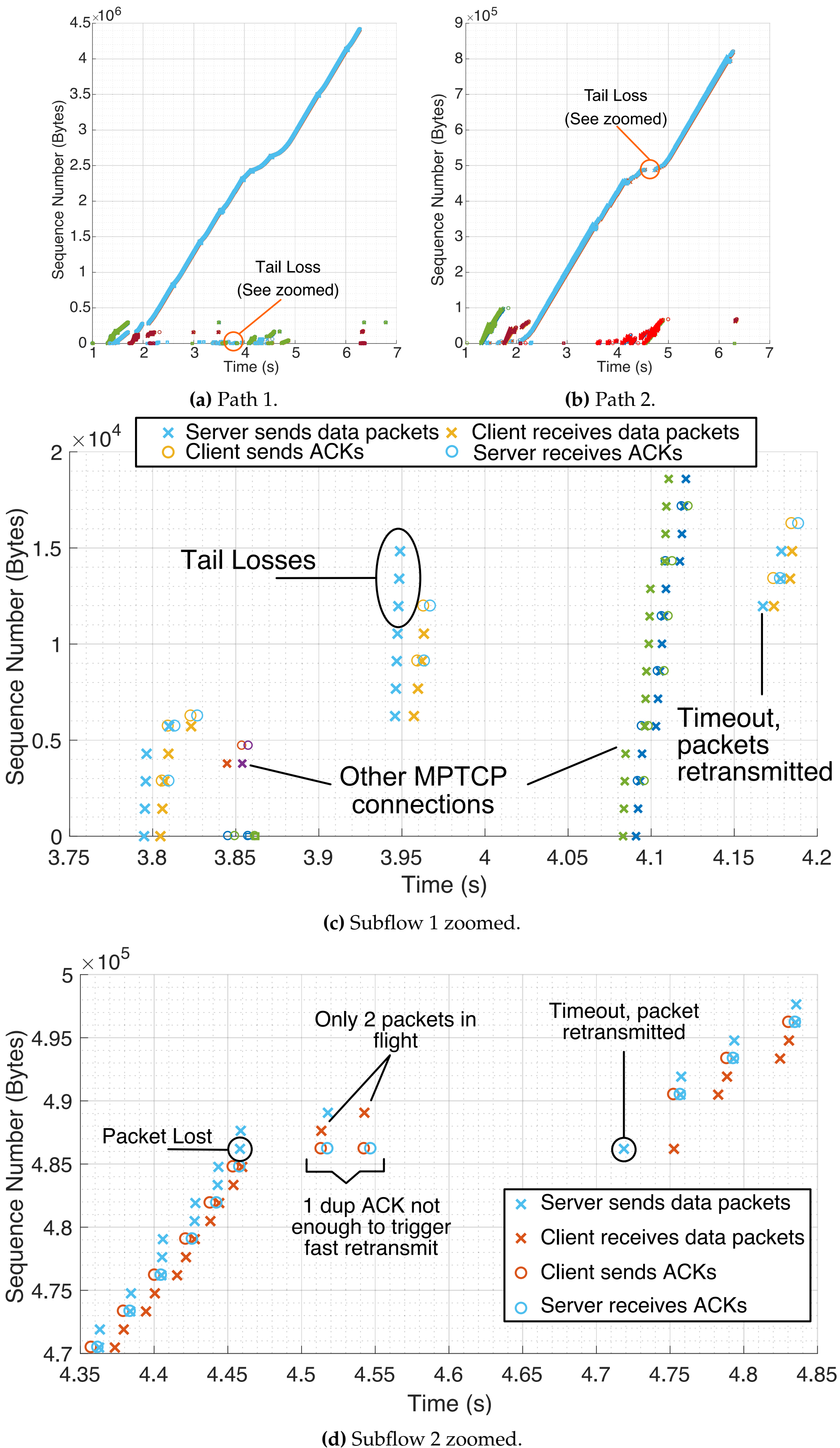


(a) Path 1.

(b) Path 2.

(c) Zoomed Path 1.

Figure 6.10: Timeline plots for groupon.com downloaded using MPTCP with a bandwidth of 10Mbps and RTT of 40ms, with $\frac{1}{5}$ the bandwidth on the second path.



(a) Path 1.

(b) Path 2.

(c) Subflow 1 zoomed.

(d) Subflow 2 zoomed.

Figure 6.11: Time sequence plot of amazon.co.uk downloaded using MPTCP with a bandwidth of 10Mbps on the first path, 2Mbps on the second, and 8ms RTT on both paths.

and $RTT_1 = RTT_2 = 8$ms. During the download, subflows on both paths (Figures 6.11(a) and 6.11(b)) experience tail loss. If we zoom in, we can see two separate examples of this phenomenon in action. Figure 6.11(c) shows tail loss on a subflow on path 1. This subflow is application limited, since it transmits a first resource within its initial congestion window, and a second resource after some period of idleness. As we can see from the figure, the last three packets of the second resource (consisting of seven packets in total) are lost. Without more packets in flight after the gap in the sequence space, the receiver cannot send any duplicate ACKs, and has no way of triggering Fast Retransmit. The subflow is therefore forced to time out before attempting to retransmit the missing packets. Unfortunately, since the RTT of the flows is 8ms, the time out duration is clamped at the minimum RTO, which is 200ms in our experiments. Although a smaller minimum RTO would be a better choice for the case where the RTT is 8ms, modern operating systems do not typically adapt the minimum RTO according to the observed RTT. Instead, they pick a fixed value, which we model using our experiments.

Figure 6.11(d) shows another instance of tail loss occurring on the second path. In this scenario, the subflow is not application limited, but rather congestion window limited. Because the bandwidth of the path is relatively low, the congestion window in this case is only three packets. Therefore, when a loss occurs as shown in the figure, there aren't enough packets in flight to trigger Fast Retransmit. Once again, the subflow is forced to wait 200ms before timing out in order to retransmit the missing packet.

Why does tail loss hurt MPTCP? To answer this question, we must first understand the relationship between tail loss and a path's bandwidth and RTT. To that end, for web sites downloaded using single path TCP, we plot the median tail loss rate *vs.* bandwidth at different RTT values in Figure 6.13(a). We include the bandwidth values for the high and low bandwidth paths used in the MPTCP experiments. The tail loss rate of a web site is the number of times we determine a tail loss has occurred using the mechanisms described Section 6.4.5, divided by the total number of downloaded resources for that site.

In general, the figure shows that the tail loss rate is higher for lower values of RTT, since these paths have a higher number of instances where flows are application limited. Conversely, as the bandwidth increases, the packet loss rate decreases, and therefore tail loss goes down to zero for high bandwidths.

We observe an interesting trend for the case where the RTT is 8ms. As the bandwidth increases from 200Kbps to 2Mbps, we transition from a region where the web downloads are bandwidth limited to a region where there is enough bandwidth that web transfers are somewhat application limited: web objects aren't packed back-to-back on a connection, meaning there are a large number of small discrete objects, each susceptible to tail loss. At the same time, the bandwidth is still low enough that packet losses are common (see Figure 6.12 for median loss rates). Note that the low RTT exacerbates loss because we give each link a buffer size

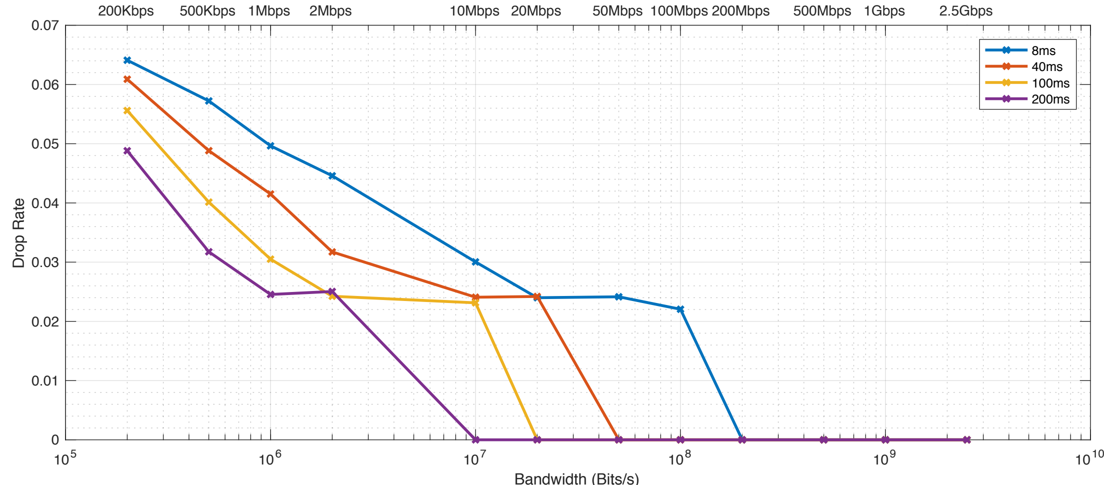


Figure 6.12: Median packet loss rate for single path TCP *vs.* bandwidth for different RTTs.

proportional to the delay-bandwidth product in an effort to avoid buffer bloat.[2] With these two forces at play, the tail loss rate increases as we increase bandwidth and transition into an application limited region, but as we increase the bandwidth past 2Mbps, losses decrease as well, and therefore the tail loss rate goes down once again.

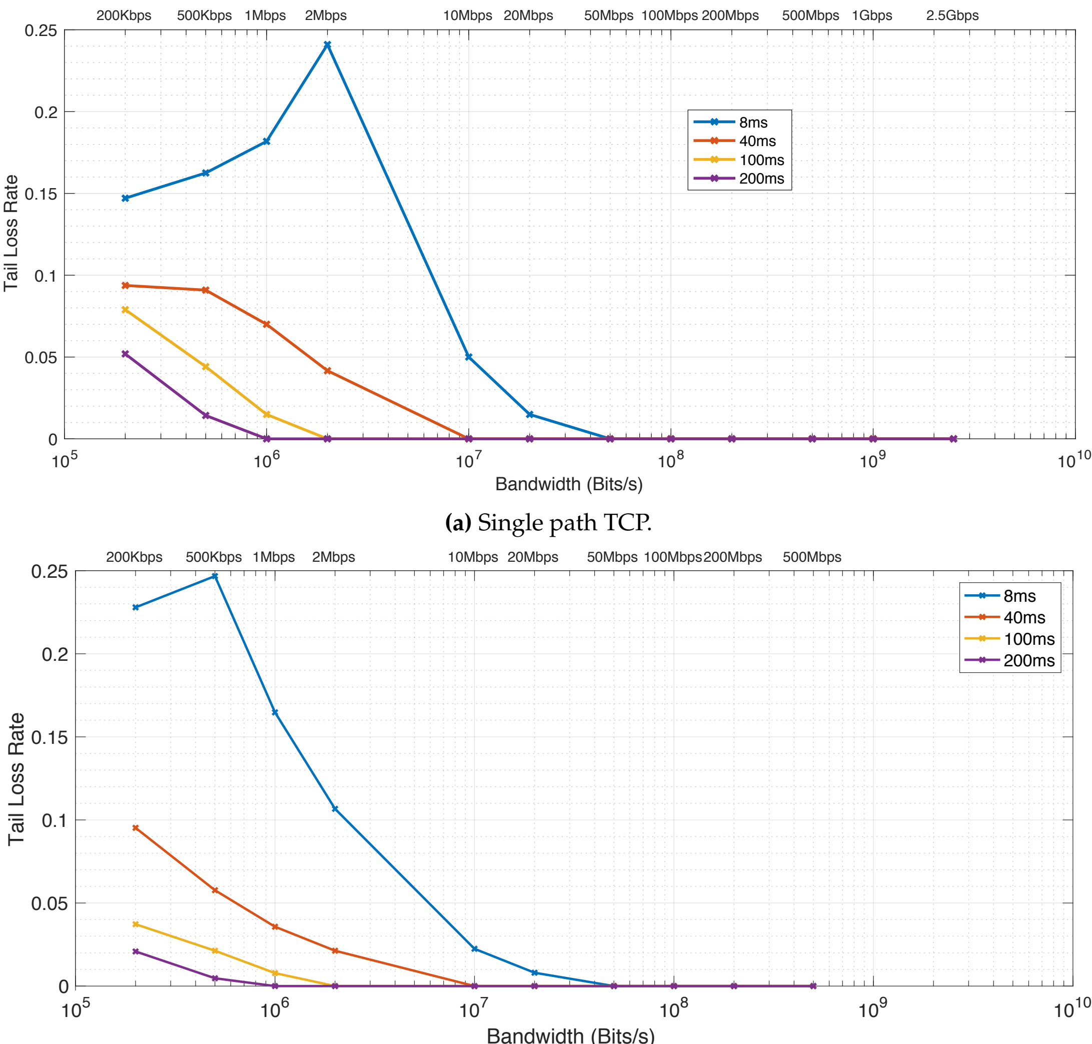
The key to explaining the poor performance of MPTCP in Figure 6.8's top left region is recognising the disparity in tail loss rates between the two MPTCP paths involved. Consider, for example, an MPTCP web download with path bandwidths 10Mbps and 2Mbps respectively, and an RTT of 8ms. If MPTCP decides to send some packets on the slower path, the packets will experience *five times* the tail loss rate of the faster path according Figure 6.13(a). Similarly, the 50Mbps path has zero tail loss, so moving any traffic onto the other path will subject it to a significant amount of tail loss. Even though most of the packets will arrive on the high bandwidth path, TCP cannot release them to the browser until the timeout has expired for the lost packets on the other path, and they have been received successfully after a retransmission. In the case where the RTT is 8ms, because our experiments use minimum RTO of 200ms, the sender must wait a comparatively long time before recovering from loss, thus increasing the PLT.

Tail loss is worse for MPTCP: striping a web page's small objects across two paths means there are twice as many tails. Figure 6.13(b) shows the tail loss rate across both paths of an MPTCP web download. Comparing equivalent bandwidths and RTTs to the corresponding single path graph shows that the tail loss rate increases when we have two paths.

6.3.4 "Worst" Path First

So far, we've examined cases where MPTCP initiates the first subflow on the best path. In this section, we will examine the effect of initiating an MPTCP connection on the "worst" path from among those available. Figure 6.14 outlines what happens at the beginning of a new

[2]Overbuffering the paths would help here, but goes against current best practice.



(a) Single path TCP.

(b) MPTCP with $\frac{1}{5}$ bandwidth on second path. The x-axis indicates the lower bandwidth.

Figure 6.13: Median tail loss rate *vs.* bandwidth for different RTTs.

MPTCP connection with two available paths. We can see that MPTCP performs a separate 3-way handshake to set up the two subflows on each of the paths. The first handshake, in addition to performing its traditional TCP role, transmits the `MP_CAPABLE` option necessary to check whether MPTCP is supported by both sides, and convey the authentication primitives required for identifying the MPTCP connection [33].

To begin using the second subflow on an additional interface, the client and server must exchange another 3-way handshake with the `MP_JOIN` option appended (used to identify the connection to be joined by the new subflow). MPTCP RFC 6824 [33] states that new subflows must not be established until DSS option has been successfully received across the path. The ACK sent by the client to the server in the initial handshake carries the client and server's keys which are required for authenticating the connection, and setting up MPTCP. Since that ACK itself is not acknowledged in return, the client has no way of knowing whether the server has in fact accepted the MPTCP connection, and not fallen back to regular TCP. The DSS option,

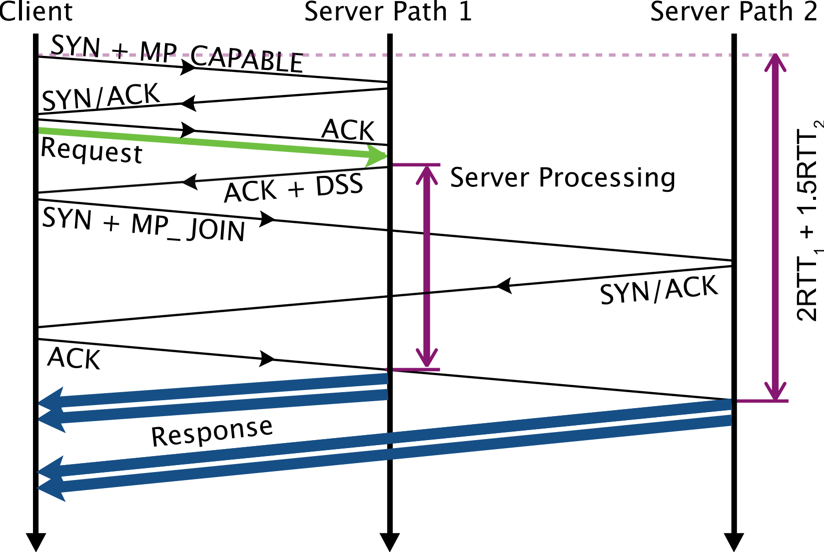


Figure 6.14: Initiating an MPTCP connection and establishing two subflows.

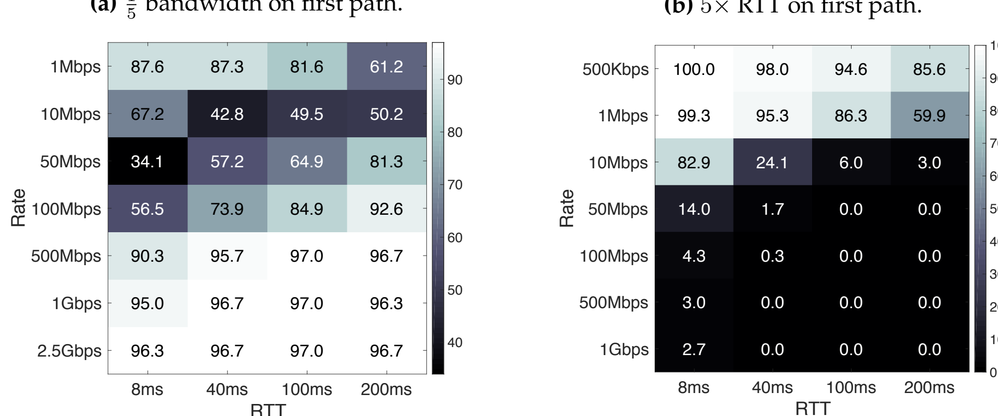
which carries the connection level sequence number and ACK number for a transmitted packet, acts as an implicit signal that the server is ready for MPTCP. Therefore the DSS option must be conveyed back to the client before it can establish a new subflow. With this restriction in mind, we can see from the figure that the client must transmit the HTTP request first and receive a corresponding ACK with the DSS option before it can establish the second subflow.

Essentially, it takes a total of $2RTT_1 + 1.5RTT_2$ from the beginning of the connection to the point where the second subflow can start transmitting data. If the requested resource is small enough to fit in one window, and if its server processing time is shorter than the time it takes to set up the second subflow (equivalent to $0.5RTT_1 + 1.5RTT_2$), then the server will transmit the entire resource on the first subflow — the second subflow will not be used at all.

With the overheads of starting a new MPTCP connection in mind, we examine the importance of picking the "better" path for establishing the first subflow by repeating the delay and bandwidth mismatch experiments from 6.3.2 and 6.3.3, this time initiating all new MPTCP connections on the slower path.

Figure 6.15(a) looks at the case where one path has $\frac{1}{5}$ of the bandwidth of the other path, and MPTCP sets up the first subflow on the lower-bandwidth path. Comparing this with Figure 6.8, where MPTCP picks the higher-bandwidth path first, we observe a similar trend in PLTs. The patterns in both heatmaps are the same—MPTCP does a good enough job of shifting traffic onto the second, better, subflow after starting on the lower-bandwidth path. Starting on the lower-bandwidth path does slightly worse in the upper right portion, because MPTCP must wait for two 3-way handshakes to complete before setting up the second subflow, which consequently begins slow-start much later in the game.

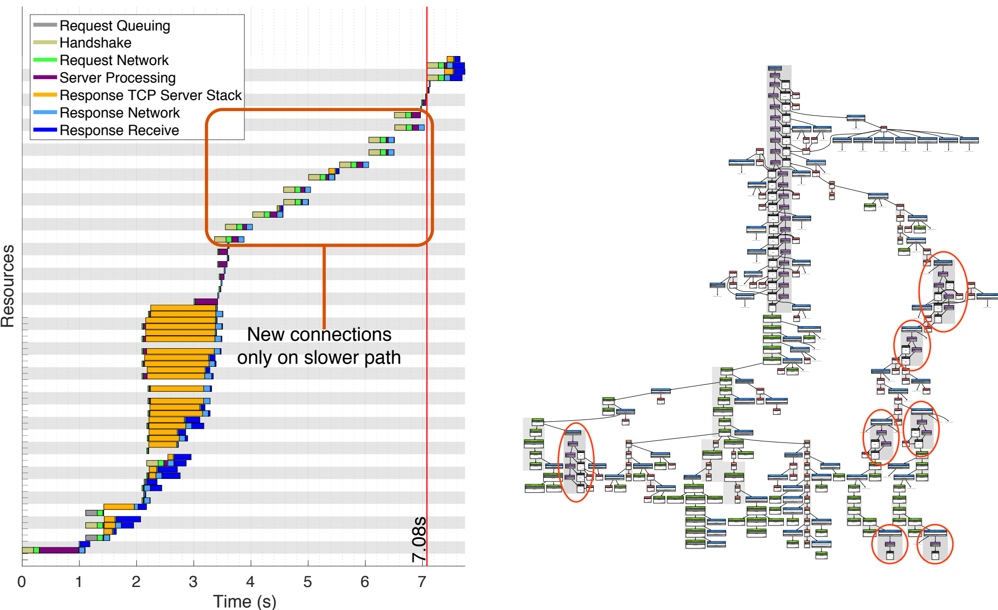
Figure 6.15(b) examines the case where one path has $5\times$ the delay of the other path, and MPTCP sets up the first subflow on the higher-RTT path. Compare these results with the ones



(a) $\frac{1}{5}$ bandwidth on first path. **(b)** $5\times$ RTT on first path.

Figure 6.15: Percentage of web sites with equal or better PLT using MPTCP vs single-path TCP when MPTCP uses the "worst" path first.

in Figure 6.6, where MPTCP picks the lower-RTT path first. Since a lot of connections will only transmit a few packets, choosing the best interface on which to perform the first handshake is essential. Getting this wrong causes MPTCP to experience a severe increase in PLTs across the board.



(a) Timeline diagram for intuit.com, where the second path has 10Mbps and 40ms RTT, and the first path has $5\times$ the RTT.

(b) Dependency graph generated for intuit.com, iframes are indicated in red.

Figure 6.16: Downloading intuit.com using MPTCP which initiates connections on the "worse" path first.

Why is this choice so important? Choosing the interface for the higher RTT path first will result in a significant overhead due to the initial 3-way handshake consuming a disproportionately large chunk of the total download time. A good example of this behaviour is intuit.com, which contains multiple iframes (see Figure 6.16(b)). For each iframe, MPTCP needs to open a new connection, each of which only transmits a handful of small web objects. Figure 6.16(a)

shows how MPTCP loads intuit.com when MPTCP uses the high delay path first. Small objects, which fit in one window, never get to use the better path. Additionally, the sever processing times are smaller than $0.5RTT_1 + 1.5RTT_2$, indicating that the second subflow still hadn't been established at the point when the server was ready to transmit the object. As a result the total PLT of MPTCP is 7.08s seconds, significantly larger than the 4.15s single-path TCP PLT (not shown here).

6.3.5 Combination Paths

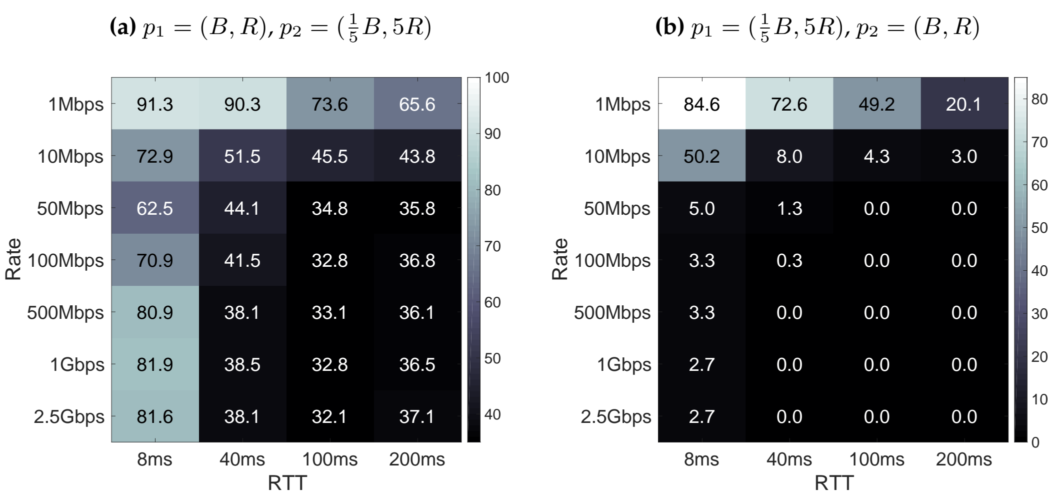


Figure 6.17: Percentage of web sites with equal or better PLT using MPTCP *vs.* single-path TCP on the "best" path ($p = (B, R)$).

So far, we have only examined the behaviour of MPTCP in scenarios where we have fixed one metric (RTT or bandwidth), and varied the other across the paths. We extend our examination to paths which differ in both RTT and bandwidth in this section. Let us denote the base bandwidth as B and base RTT as R, so that we can concisely specify the *(Bandwidth, RTT)* tuple for each path p_r. The path index denotes its order with respect to MPTCP connection initiation: MPTCP therefore initiates new connections on p_1 and establishes subsequent subflows on p_2.

We begin with the case where one of the paths has both $5\times$ the RTT *and* $\frac{1}{5}$ the bandwidth of the second path, *i.e.* that path has $(\frac{1}{5}B, 5R)$. When MPTCP initiates connections on the second, better path, we obtain the heatmap shown in Figure 6.17(a). Immediately, we can see that this heatmap is in a sense a merge of the two heatmaps in Figures 6.6 and 6.8 in terms of the observed trends. The lower right corner once again shows that for bandwidths where web downloads are latency bound, scheduling critical resources on the slower path increases PLTs for multipath with respect to single path TCP. In the regions where the web downloads are bandwidth limited, we see the effects of tail losses and LIA as described in Section 6.3.3.

On the other hand, if MPTCP initiates connections on the worse path, we obtain the heatmap shown in Figure 6.17(b). As we've seen before, the cost of initiating MPTCP on a bad path is prohibitive, and the performance of MPTCP is especially poor since we are now

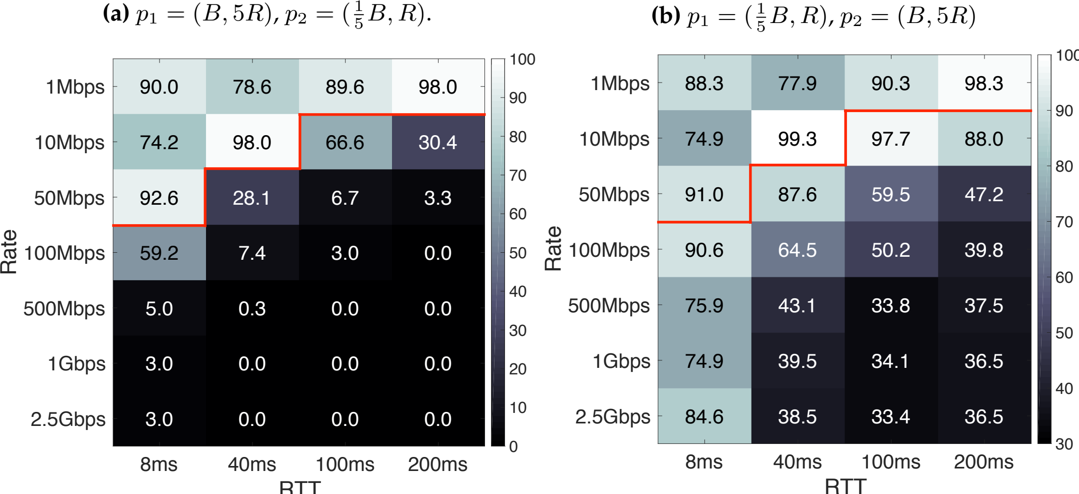


Figure 6.18: Percentage of web sites with equal or better PLT using MPTCP *vs.* single-path TCP on the "best" path. Above the red line, $(B, 5R)$ is the best path. Below the read line, $(\frac{1}{5}B, R)$ is best.

comparing it to single path TCP on the path which has both higher throughput *and* lower latency.

Next, we examine the case where one path has $5\times$ the RTT and the other has $\frac{1}{5}$ the bandwidth, *i.e.* the path tuples are $(\frac{1}{5}B, R)$ and $(B, 5R)$. Recall that our goals is to compare MPTCP's performance with TCP on the best path. For this combination of paths, it is unclear which of them is in fact "better" and therefore to be used for baseline TCP in our heatmaps. Upon investigation, we discovered that for bandwidth and RTT combinations where a TCP flow is bandwidth limited, single path $(B, 5R)$ has the lowest mean PLT across web sites, and is therefore the best path. Conversely, for regions where a TCP flow is typically application limited, path $(\frac{1}{5}B, R)$ is the best path in terms of PLT. Therefore, if we compare the multipath case for a particular *(Bandwidth, RTT)* combo with the single path case on the path with the lowest mean PLT, our heatmaps will have two regions. We use a red line on the heatmap to split distinguish between them. Above the red line, the mean PLT across all web sites (with 50 runs per web site) was lower for the path $(B, 5R)$. As a result, each square above the red line compares MPTCP with single path TCP running on a path with $(B, 5R)$ as the baseline. Below the red line, the path $(\frac{1}{5}B, R)$ has lower mean PLT and is used as the TCP baseline instead.

Figure 6.18(a) shows a heatmap for the case where MPTCP initiates connections on the path with $(B, 5R)$. Although MPTCP performs well above the red line (where $(B, 5R)$ is the best path), the region below the line is another example where choosing a high RTT path for initiating the MPTCP connections greatly penalises PLT.

On the other hand, if MPTCP initiates connections on path $(\frac{1}{5}B, R)$, Figure 6.18(b) shows that we get a better outcome for MPTCP. Above the red line, we get the effects that match the upper left diagonal of Figure 6.15(a). Although the first path has $\frac{1}{5}$ the bandwidth, MPTCP still manages to shift traffic onto the second better path. Conversely, below the red line we

observe a trend that matches the lower right diagonal of Figure 6.6. Although the second path has a higher bandwidth, the fact that it also suffers from high RTT will negatively impact PLT. Latency bound flows will experience higher PLTs if their resources are scheduled on a high delay path; the extra bandwidth has little to no effect since the flows are already application limited.

6.4 Improving MPTCP

The previous section highlighted regimes where MPTCP increases web page load times compared to single path TCP, and presented limitations in the existing protocol responsible for the increase. In this section, we introduce mechanisms which mitigate MPTCP's shortcomings by addressing each limitation in order.

We present solutions which are unconstrained by the isolation between the transport and application layers. Our goal is to test the boundaries of possible solutions and to design a better multipath web protocol regardless of the socket API. In principle, both the server and the client retain all the information necessary to achieve this goal, irrespective of which layer this information resides in. In Section 6.6.4, we explain how we might implement the cross-layer interaction between TCP and the application in order to convey this required information.

6.4.1 Stream Scheduler

We first address the most serious MPTCP shortcoming with respect to web page completion times. As we've seen in Section 6.3.2, page completion times suffer if MPTCP's default RTT scheduler places packets from critical objects on high delay paths. We have developed an HTTP-aware Stream Scheduler that uses knowledge of web resources and their priorities to make smarter decisions to choose between available subflows. We focus our discussion on the case of two subflows, corresponding to the common scenario of having a WiFi and 3G/LTE interface. We believe our approach should generalise to more than two paths. However, web browsing over more than two paths is unlikely in real world deployments. As a result, we do not explore these cases in our work.

The intuition behind the Stream Scheduler is quite simple: schedule each packet from a web resource (or "stream" in the HTTP/2 sense) with the goal of minimising the resource's overall completion time. By minimising the completion time of each critical resource, we expect overall page load time to decrease. The question now becomes: how do we minimise a resource's completion time given the choice of two subflows?

When transmitting segments towards the beginning of a resource, adding extra bandwidth by sending segments on a second path is advantageous and will minimise the resource's completion time. At this point, sending a segment on the higher delay path does not generally increase completion time because we are still at the beginning of the resource. On the other hand, when transmitting towards the end of a resource, sending the last segments on the higher delay path would increase the resource's completion time. The key is to ensure that we never

transmit a resource's segment on a particular subflow if we expect it to arrive *after* the time required to transmit all of the resource's remaining bytes on the other subflow. Doing so would only increase PLT, and we are better off sending all the resource's outstanding segments on the other subflow instead.

Algorithm 6 Simple Stream Scheduler

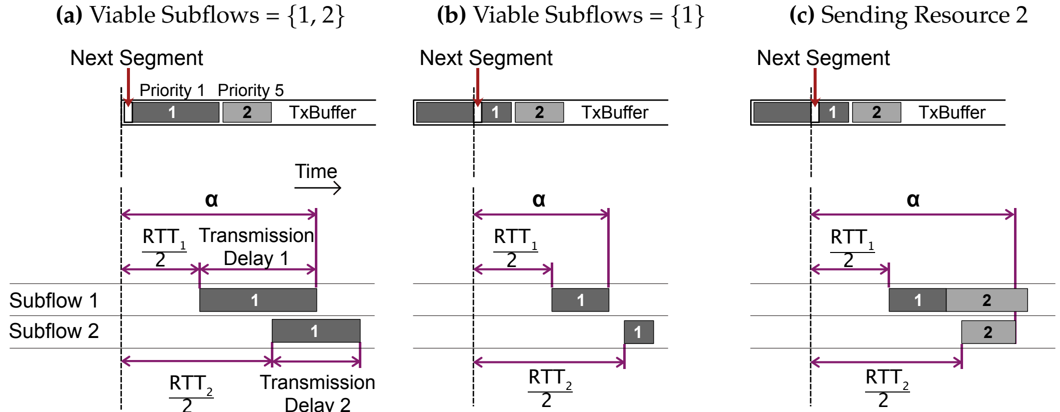
```
1: function GETSUBFLOW(subflows)
2:     availableSubflows ← Subflows with space in cwnd
3:     bytes ←SIZE(resource in TxBuffer)
4:     α ← ESTIMATECOMPLETION(bytes, subflows)
5:     viableSubflows ← [subflow s in availableSubflows if RTT_s/2 ≤ α]

6:     if not ISEMPTY(viableSubflows) then
7:         subflowToSend ← viable subflow with highest RTT
8:         return subflowToSend

9:     return NULL.
```

Applying the principles above, we arrive at Algorithm 6, which describes the Stream Scheduler's basic operation. An MPTCP scheduler calls GETSUBFLOW whenever space opens in any subflow's congestion window, and whenever new data is queued for transmission. It returns the subflow on which to send the next packet in the transmit buffer, or NULL if it is better to not send and wait for the other subflow's cwnd to open.

The scheduler first determines which subflows have available space in their congestion window to transmit a new segment (line 2). It then determines the remaining unsent bytes of the resource at the front of the TxBuffer on line 3. Given the number of bytes to send, the scheduler estimates their completion time if sent using each subflow (irrespective of whether the subflow has space available at this instant), and sets α to the shortest of these times (line 4). α is the minimum completion time, *i.e.* the resource arrival time if we send the remaining bytes of the resource only on the best subflow. We discuss how to estimate completion times in Section 6.4.1.2. A subflow is viable only if it can deliver a single packet before α. It takes roughly $\frac{RTT_s}{2}$ for a packet to reach the client using subflow s (assuming the serialisation delay of a single segment is negligible), so $\frac{RTT_s}{2} \leq \alpha$ tests whether a subflow is viable (line 5).

Sometimes more than one subflow is viable. A typical example occurs when the flows have been idle for some time because the connection is application limited. In this case, both subflows have several packets worth of free space in their respective congestion windows, and both will end up transmitting several packets in succession when a new resource arrives from the application layer. Intuitively, it matters little how we stripe packets across two viable subflows. However, TCP usually cannot tell precisely which packets are queued for transmission below TCP, so there is always some uncertainty as to whether a transmission has happened or not. When TCP releases a packet, α will reduce, as the remaining bytes to send from the resource have reduced. If it first packs packets onto the lower RTT path, the higher RTT path may become un-viable prematurely, because TCP just "sent" a burst of packets, but they haven't ac-

tually been transmitted yet. To avoid this, we first send packets on the higher RTT path, as it will become un-viable first as α reduces. This mechanism better spreads traffic across both paths when it is advantageous to do so.



(a) Viable Subflows = $\{1, 2\}$ **(b)** Viable Subflows = $\{1\}$ **(c)** Sending Resource 2

Figure 6.19: Choosing viable subflows

With the basics out of the way, let us consider a more concrete example that will demonstrate the Stream Scheduler in action. Suppose a server would like to transmit two resources which it has enqueued in its socket buffer, as shown in Figure 6.19(a). The top part shows the send buffer which contains two resources. The next segment to transmit is at the beginning of Resource 1. The lower part of the figure shows how long it would take to transmit all of Resource 1 on either of the available subflows. The server's scheduler must choose between the two subflows, each with different RTTs and bandwidths, to send the first segment of Resource 1 on.

The Stream Scheduler estimates how long it will take to transmit Resource 1 in its entirety on both of the subflows, and computes α, the minimum of these two values. From the figure, we see that α is equivalent to the time it takes to send the entire resource on Subflow 1. Since $\frac{RTT_1}{2} \leq \alpha$ and $\frac{RTT_2}{2} \leq \alpha$, both subflows are viable, and we transmit the segment on Subflow 2 as it has the higher RTT.

Later on in the transmission, the scheduler attempts to transmit a segment of higher sequence number from the same resource. Since the resource has less data to send, we obtain the scenario shown in Figure 6.19(b). Now, if we attempted to send the segment on Subflow 2, it would take longer to arrive than if we just sent *all* of Resource 1's remaining bytes on Subflow 1. Therefore, Subflow 1 is the only viable subflow.

6.4.1.1 Lookahead

Suppose in Figure 6.19(b) that Subflow 1 cannot send the current segment due to lack of space in its congestion window, and Subflow 2 is the *only* available subflow. Seemingly, the scheduler must choose between two unattractive options: defer sending anything and risk wasting capacity, or send the segment on Subflow 2 anyway and suffer extra delays in completion time.

A third and better option exists: since Subflow 2 has the capacity to send right away, it can

send a resource found later in the transmit socket buffer. Typically this will be a lower priority resource. Even though it may not be viable to send the first resource on Subflow 2, it may be viable to send the second resource, as its α includes the time to send both the first and second resources on Subflow 1.

To illustrate, consider the example in Figure 6.19(c). Subflow 2 cannot send Resource 1 segments since they will arrive too late. But, if we apply the same logic for determining viable subflows described earlier, we find that Subflow 2 can deliver a Resource 2 segment before Subflow 1 can deliver *both* the remaining Resource 1 bytes and Resource 2 in its entirety.

Note that α in this scenario is the minimum of the time it would take to transmit Resources 1 and 2 on Subflow 1, and to transmit only Resource 2 on Subflow 2. But, if we keep in mind that $\frac{RTT_2}{2}$ will always be smaller than the time it takes to fully transmit Resource 2 on Subflow 2, we only need to check whether the time it takes to send Resource 1 and 2 on Subflow 1 is less than $\frac{RTT_2}{2}$. As a result, we can simplify the algorithm and compute α as the minimum time it takes to transmit the remaining bytes of both Resources 1 and 2.

The loop in Algorithm 7 accomplishes the lookahead—the scheduler continues to look later in the transmit queue till it finds a resource to send on the slower path without incurring delays in the completion time. This algorithm returns a pair containing both the resource to send and the subflow on which to send it.

Algorithm 7 Stream Scheduler with Lookahead

```
1:  function GETSUBFLOW(subflows)
2:      availableSubflows ← Subflows with space in cwnd
3:      bytes ← 0

4:      for resource in TxBuffer do
5:          bytes ← bytes+ SIZE(resource)
6:          α ← ESTIMATECOMPLETION(bytes, subflows)
7:          viableSubflows ← [subflow s in availableSubflows if RTT_s/2 ≤ α]

8:          if not ISEMPTY(viableSubflows) then
9:              subflowToSend ← viable subflow with highest RTT
10:             return (resource, subflowToSend)
11:     return NULL.
```

In order to send a segment from Resource 2, which originally has a higher sequence number, we rearrange the transmit buffer to bring the segment forward such that its sequence number is now in order. Without rearranging the transmit buffer, extra complexity is required to keep track of which segments were already sent, and may incorrectly signal to the receiver that a loss has occurred. Rearranging the buffer simplifies the design of the system, but necessitates using a QUIC-like out-of-order delivery of segments to the application layer, in order to fully benefit from lookahead.

6.4.1.2 Completion Estimator

The Stream Scheduler needs to estimate the completion time of a resource (or a group of resources) on each of the available subflows in order to compute α. As shown in Figure 6.19(a),

this completion time is the sum of the propagation delay of the path and the transmission time of the bytes we wish to transmit. We can estimate these values from a subflow's RTT and congestion window metrics as measured by TCP, taking into account whether TCP is in slow-start or congestion avoidance.

A resource's one way propagation delay is typically $\frac{RTT}{2}$ for that subflow. As for the remaining bytes' transmission time, if a subflow is in congestion avoidance, then we can compute it using the estimated bandwidth $B = \frac{W}{RTT}$. Here, the window $W = \min(cwnd, rwnd)$, and is equivalent to the amount of bytes a sender can transmit. Since most web objects are small, and our experiment sets rwnd to a large value, we are not receive window limited in our experiments anyway. One point to keep in mind is that cwnd is artificially inflated during New Reno's Fast Retransmit phase, therefore we set W to $ssThresh$ during Fast Retransmit, since $ssThresh$ stores the cwnd value at the point a loss is detected, and window inflation commences.

On the other hand, if the subflow is in slow-start, then it will grow its window by one segment for each ACK received. Since we use delayed ACKs in our experiments, after an RTT then next $cwnd_{i+1}$ is:

$$cwnd_{i+1} = cwnd_n + cwnd_i/2 = 1.5\, cwnd_i$$

The congestion window will therefore grow exponentially until it becomes equivalent to $ssThresh$ bytes[3] after n_t RTTs. Therefore, n_t can be derived as follows:

$$1.5^{n_t - 1} \cdot cwnd = ssThresh \rightarrow n_t = \log_{1.5}\left(\frac{ssThresh}{cwnd}\right) + 1 \tag{6.2}$$

The amount of data a sender transmits during slow-start can be captured with a geometric sum:

$$\begin{aligned} size &= cwnd + \lambda \cdot cwnd + \lambda^2 \cdot cwnd + \cdots + \lambda^{(n-1)} \cdot cwnd \\ &= cwnd \cdot \frac{\lambda^n - 1}{\lambda - 1} \\ &= 2 \cdot cwnd(1.5^n - 1) \quad \text{when } \lambda = 1.5 \end{aligned}$$

Therefore, to deliver $size$ bytes in slow-start, the number of RTTs n required is:

$$n = \log_{1.5}\left(\frac{size}{2 \cdot cwnd} + 1\right) \tag{6.3}$$

From equations 6.2 and 6.3, the total data transmitted D_{ss} from the beginning of slow-start

[3]If we are at the very beginning of a connection, and there is no measure of ssThresh, then we cap ssThresh it at $10^6 \times RTT$.

until cwnd equals ssThresh is:

$$D_{ss} = 3 \cdot ssThresh - 2 \cdot cwnd$$

Combining these together, we can compute the completion time α_s of transmitting $size$ bytes of data on subflow s:

$$\alpha_s = \begin{cases} \frac{RTT}{2} + n \cdot RTT & \text{if } cwnd < ssThresh \text{ and } size \leq D_{ss} \\ \frac{RTT}{2} + n_t \cdot RTT + \frac{(size - D_{ss})}{B} & \text{if } cwnd < ssThresh \text{ and } size > D_{ss} \\ \frac{RTT}{2} + \frac{size}{B} & \text{if } cwnd \geq ssThresh \end{cases}$$

We find the minimum α_s over the subflows using function ESTIMATECOMPLETION. The minimum completion time α determines which subflows are viable for the transmission of the next segment. Algorithm 8 shows how to compute the completion time α_s for each subflow programmatically in light of the above expressions.

Algorithm 8 Estimate Completion

```
1: function ESTIMATESUBFLOWCOMPLETION(size, subflow)
2:     RTT ← GETRTT(subflow)
3:     w ← GETWINDOW(subflow)
4:     totalTime ← RTT/2
5:     remainingBytes ← size
6:     while remainingBytes > 0 do
7:         if w < ssThresh then
8:             if remainingBytes >= w then
9:                 totalTime ← totalTime + RTT
10:            else
11:                B ← w/RTT
12:                totalTime ← totalTime + remainingBytes/B
13:            remainingBytes ← remainingBytes − w
14:            w ← w + w/2
15:        else
16:            B ← w/RTT
17:            totalTime ← totalTime + remainingBytes/B
18:            remainingBytes = 0
19:    return totalTime
```

Naturally, the completion estimator is bound to make mistakes. At the point where it computes the completion time of a resource, it cannot predict whether the resource will experience loss, or whether the connection itself will time out. Additionally, the estimator cannot guess whether the RTT of the connection will grow beyond the current sampled value as more packets are placed on the network. Given two subflows, losses are more likely on the worse subflow than the better one, so failures take us closer to vanilla MPTCP striping behaviour than to vanilla TCP single-path behaviour.

To understand how well the completion estimator performs, let us look at a particular example, namely downloading amazon.co.uk using single-path TCP. For every packet we transmit

for a particular resource, we record the estimator's prediction of the time it will take to transmit the remaining bytes of that resource to the client. Given we know the *actual* completion time of the resource, we plot the real remaining durations versus the estimated ones as the scatter plot shown in Figure 6.20(a). Packets from the same resource have the same color, and they have different symbols based on the TCP congestion state. If we zoom in (Figure6.20(b)), we see that most points lie on the $y = x$ line, which indicates that the real and estimated completion times are nearly equivalent for most resources.

If a point lies below the black line, the estimator has overestimated how long a resource will take to transmit on the path, and a resource will in reality arrive at the client sooner than predicted. We do not worry much about these points, because at worst, they do not increase PLTs but rather represent missed opportunities for using a second subflow. Conversely, if a point lies above the $y = x$ line, the estimator has underestimated the completion time. Underestimation is problematic, since it may prompt the scheduler to place a resource on a subflow where it completes much later than predicted, therefore increasing PLT. Most of these outliers are bounded within 2$\times$ of the black line. The factor of 2 is pertinent because TCP halves its window when a loss is detected, meaning the estimated bandwidth we use in our computation of α is off by a factor of two. On the other hand, because the amount of buffering used in our experiments is equivalent to a BDP, we expect the RTT to at most double due to queuing.

The resources marked by 1 and 2 in the figure are examples where the estimator does particularly poorly (values outside of the bounds) due to early losses in slow-start. The x's above the black line represent packets from Resource 1 and 2 that are in slow-start. As for the squares and circles below the line, they represent packets from the same resources that are undergoing fast recovery or additive increase. Resource 1 actually times out due to loss, after which it grows its window using Additive Increase. The x's above the red line represent underestimating the completion time during slow-start: the estimator believes that the flow will continue growing its window exponentially and transmit the bytes sooner, rather than abruptly switching to additive increase due to a loss. The squares marked by 1 are packets from the same resource undergoing Additive Increase from a small starting window. Because the window is so small, we tend to underestimate the speed of the subflow (recall we use $B = \frac{w}{RTT}$ as the bandwidth value). But, for this case the estimate is within 2$\times$, since we halved the window. Resource 2 exhibits similar behaviour, although it does not experience a timeout.

Resource 3, on the other hand, represents a case where the estimator predicts completion time rather well, except for several packets around a loss event where the RTT grows beyond our estimate. Finally, Resource 4 is a case where we have errors in estimation because of a large burst of losses in slow-start, which are recovered using NACKs.

Packet loss, as demonstrated, causes serious problems for the estimator. But, even though the completion estimator does not take into account loss probability, we've discovered in our experiments that it provides a good enough estimate for α_s to help significantly. If more ac-

curacy is needed, previous work on modelling TCP latency [146] provides a similar model for estimating completion time, which also incorporates loss probability.

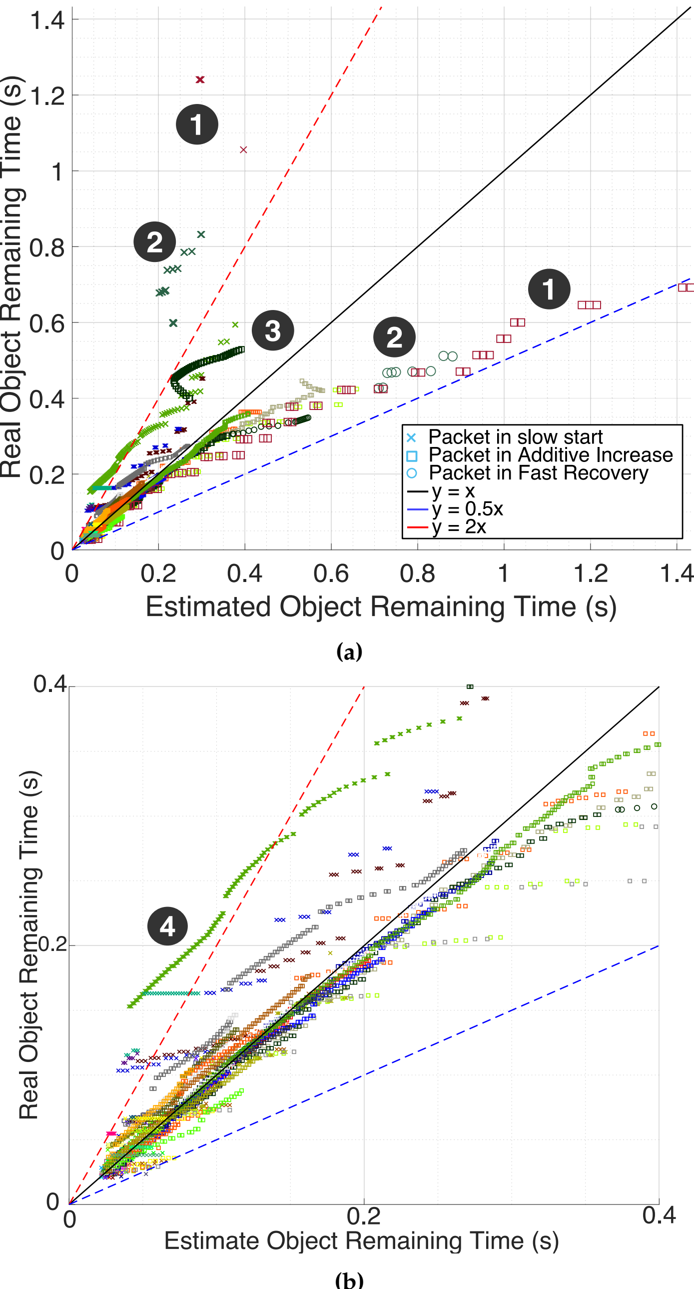


Figure 6.20: Real *vs.* Estimated resource completion time for amazon.co.uk downloaded using single-path TCP on a connection with bandwidth of 10Mbps and RTT of 40ms.

6.4.1.3 Retransmissions

The previous section has shown that packet losses reduce the accuracy of the completion estimator, and therefore diminish the efficiency of the Stream Scheduler. As a solution, we experimented with retransmitting lost bytes from a particular sublfow on the other subflow in order to improve resource completion times. Whenever we receive a NACK with a particular sequence number, we attempt to retransmit the missing segment on the other subflow. The lost

segments are in essence replicated on both subflows in accordance with MPTCP's semantics.

Schedulers that retransmit segments on the other path have in the past been explored in the context of mitigating the effects of the receive window closing [76, 85]. But in our case, preliminary results indicated that retransmitting a second subflow's segments indiscriminately delays the current subflow's delivery of its own resource segments, increasing completion time. Therefore, we only schedule retransmissions on a second subflow if that subflow is idle and has completed sending its data. We apply the same check for viability to retransmits on an alternate subflow, so that we never perform these retransmissions if we expect them to increase a resource's completion time.

6.4.2 Preemption

In our experiments in Section 6.3, we used HTTP/2 with prioritisation based on an omniscient critical path analysis of a web site's dependency graph. We found that to get the best wins, we needed to apply resource prioritisation to the MPTCP socket buffer. We opted not to preempt resources in the socket buffer if a resource with a higher priority was subsequently added. In other words, we allowed the resource at the front of the queue to be transmitted fully even if a higher priority resource was later added.

As an optimisation to multipath web, we enable preemption in the transmit socket buffer. If a critical resource is added to the buffer, it will push the untransmitted bytes of a lower priority object further back in the buffer, *even if* we had begun transmitting the beginning of the resource.

Enabling preemption is an optimisation which benefits all regimes, single or multipath. Nevertheless, recall that under the Stream Scheduler, the subflow with higher RTT looks ahead to find a lower priority segment if it cannot send the current one. Therefore, a correct ordering of resource bytes in the transmit buffer is crucial for its correct operation, rendering preemption particularly important in the context of the Stream Scheduler.

6.4.3 Out of Order Delivery

TCP's in-order delivery causes head-of-line blocking for web traffic if there are losses on the connection. A completed resource may be forced to wait for the gap in TCP's receive buffer to fill before it can be delivered to the application layer. With MPTCP, if two paths have mismatched RTT's, gaps will also exist in the receive buffer due to segments being delayed on the slower path. Additionally, recall that the Stream Scheduler allows the higher delay subflow to send a segment from a lower priority resource, and rearranges the transmit buffer so that the segment has a lower sequence number. As a result, the scheduler ends up mixing segments of the same resource, where some are sent earlier on the slower subflow due to lookahead, and the rest are sent on both subflows some time later. Lower priority resource segments on the slower subflow may consequently block other resources transmitted in the interim, further exacerbating the HOL blocking.

Although an increase in head-of-line blocking is an argument against rearranging the

transmit buffer, we opt to keep this simpler design, adopting instead a different policy with respect to gaps in the sequence space. We find that QUIC is ideally suited to tackle these HOL blocking delays due to a crucial design feature: it delivers out-of-order segments from different streams to the application immediately. To use QUIC's out-of-order feature, we implemented QUIC-lite for MPTCP (as described in Section 4.3). QUIC-lite annotates each TCP segment with its contents' stream IDs, offsets and sizes. These allow for correct multiplexing and reassembly of resources in the application layer without HOL blocking in the transport.

6.4.4 Disabling Linked Increase

Linked Increase was designed to grow the congestion windows of two MPTCP subflows jointly so as to be fair to TCP flows sharing a common bottleneck link if one exists [32, 73]. In Section 6.3.3, we saw that in low bandwidth regimes, when a subflow suffers early loss and leaves slow-start with a tiny cwnd, LIA may slow the growth of the congestion window leaving the link under-utilised for much longer than TCP.

Figure 6.8 shows that even when LIA hurts performance, MPTCP generally still does well overall. Some may argue that taking a small hit in performance is a reasonable price to pay for ensuring fairness. However, if our goal is to minimise completion times in comparison to single path TCP, we should consider disabling LIA for web traffic and using additive increase on each subflow independently, or perhaps BBR [68].

As opposed to uncoupling the subflows entirely, a partial solution is to relax sublfow coupling by using Opportunistic LIA (OLIA) [147]. Nevertheless, we would like to explore the extremes of the possible solutions to slow window growth. As a result, we choose maximum uncoupling by disabling LIA entirely, so that the subflows are independent in their congestion control.

We can argue that for most use cases of MPTCP, *i.e.* mobile devices, the LTE and WiFi paths are independent. The only links they share should be at the server, which we expect to be sufficiently provisioned as to not be the bottleneck. On the other hand, if a user subscribes to an ISP that supplies them with both broadband and 3G/LTE services, a shared bottleneck link close to the user will most likely exist. For these scenarios, it is probably best not to disable LIA. We note that with legacy HTTP/1.1 web servers, web browsers typically initiate six TCP connections per domain when downloading. In other words, web flows were already unfair to other traffic and disabling LIA for HTTP/2 traffic will actually be less aggressive. We later compare MPTCP both with and without LIA to illustrate this effect.

6.4.5 Tail Loss Recovery

Splitting a resource across two subflows exacerbates tail loss, as illustrated in Section 6.3.3. In particular, for bandwidth mismatch regimes where one path suffers a high loss rate relative to the second path, but both paths have sufficient bandwidth so as to be application limited, tail losses may lead to an increase in PLT with respect to the best single path, since the "worse" path times out frequently in order to recover missing segments at the end of a window. The

effect is especially punishing for short RTTs, *e.g.* 8ms, since a sender must wait a minimum RTO of 200ms before retransmitting the lost segments.

To mitigate the effect of tail loss on page completions, we use a modified version of Tail Loss Probes (TLP), as presented in [145] and evaluated in [148]. The idea behind Tail Loss Probes is quite simple: if a sender has not heard an ACK for the duration of a probe timeout (PTO), then it will assume a tail loss has occurred. In response, it will attempt to trigger fast retransmit by sending a probe segment to a SACK enabled receiver. Essentially, TLP allows a sender to avoid waiting an RTO before determining an ACK is overdue and attempting to query the receiver for more information.

A subflow checks whether it should schedule the PTO timer at two different points in its operation: whenever it is transmitting new data, and whenever it receives an ACK while the PTO timer is already running. The check is performed according to Algorithm 9, which also computes the duration of the PTO. The algorithm operates as follows: if the subflow is in the OPEN state, schedule a timeout if there aren't enough segments in flight to trigger fast retransmit, or the connection is application limited. The value of PTO is generally set to $2 \times SRTT$ where $SRTT$ is the smoothed RTT estimate computed using [149]. Therefore, PTO will usually be smaller than an RTO.

When operating with delayed ACKs, the PTO must also account for the fact that ACKs are transmitted in response to every other segment. As such, if there is only one segment in flight, PTO will include the delayed ACK timeout as well to prevent sending probes when an ACK is in fact en route.

Algorithm 9 Scheduling the PTO

```
1: function SCHEDULEPTO(Subflow)
2:     if (GETSTATE(Subflow) == OPEN) &&
3:         (ISAPPLICATIONLIMITED(Subflow) || flightSize < dupAckThresh) then
4:         if flightSize == 1 then
5:             PTO ← min(2 × SRTT, 10ms) + delayedAckTimeout
6:         else
7:             PTO ← min(2 × SRTT, 10ms)
8:         SCHEDULE(PTO, PTOHandler)
```

When the PTO fires, the subflow will either retransmit the highest transmitted sequence number, or the next untransmitted segment in the socket buffer. The goal of this "probe" segment is to elicit an ACK from the receiver with a SACK block, which subsequently allows the sender to determine whether a loss has in fact occurred and thus trigger Fast Retransmit using Forward ACKs (FACK) [150] or Early Retransmit [151] as the scenario dictates.

With the basic operation of Tail Loss Probes in hand, we performed several modifications to allow TLP to operate correctly with our implementation of MPTCP in the face of the following issues:

Absence of SACK implementation : Because our ns-3 TCP implementation uses NACKs for loss detection and recovery, we cannot use FACK to trigger Fast Retransmit, since FACK requires knowledge of the highest received sequence number as reflected back to the sender in a SACK block. Nevertheless, given that our experiments experience no packet reordering and no ACK losses, we can perform the following modification to allow us to use TLP in the absence of a SACK-enabled receiver. When a subflow's PTO fires, the subflow enters a new congestion state which we will call TAIL_LOSS. If a subflow receives any duplicate ACK whilst in this TAIL_LOSS state, it will immediately go into Fast Retransmit, without having to wait for 3 duplicate ACKs to arrive. This behaviour is very similar to Early Retransmit, and will not perform any better than the current proposed TLP mechanism, which, with the use of FACK, would have enough information from a SACK block to trigger Fast Retransmit from just one returning ACK. We do not advocate for making this simplification for a real system, but for our purposes it is an adequate change to allow us to prototype TLP's effect on page completion times.

Accounting for Delayed ACKs takes too long : Our initial experiments with TLP for multipath highlighted several scenarios where the PTO timer takes too long to fire, thus rendering any improvement to web page completions minimal when compared to waiting a full RTO in event of a tail loss. The culprit in these scenarios is the inclusion of the delayed ACK timeout when computing the value of PTO. Recall that we set the delayed ACK timeout to 100ms for our experiments, a relatively large value when compared to an RTT of 8ms.

For TLP to provide sufficient improvements in page completion for bandwidth and RTT regimes in the top left corner of Figure 6.8, we should eliminate the need to wait a delayed ACK timeout before concluding that a tail loss has occurred. Because we are dealing with web traffic, we know that MPTCP subflows are application limited at object boundaries, which is where we expect a receiver to time out before sending a delayed ACK for the very last segment. Normally, the sender uses the PSH flag to avoid this extra delay. PSH signals to the receiver that it should deliver the object to the upper layer at once, and in response send an immediate ACK outside of the delayed ACK mechanism.

In theory, the use of the PSH flag should allow us to remove Line 5 from Algorithm 9. While this strategy works well for TLP on single path connections, for multipath connections the PSH flag is transmitted on only *one* subflow. The other application limited subflow will still need to wait for a delayed ACK timeout before getting the final ACK. Figure 6.21 illustrates this issue: for an object consisting of eight packets, the sender sets the PSH flag on the last packet (packet 8), which it schedules on Subflow 1. Subflow 1, as a result, receives an ACK immediately outside of the delayed ACK mechanism. Conversely, packet 5 is the last packet of the object transmitted on Subflow 2. Since it does not carry a PSH flag, the corresponding ACK on that subflow is only transmitted after a delayed ACK timeout.

What we need is a mechanism for triggering ACKs at the object boundaries on both sub-

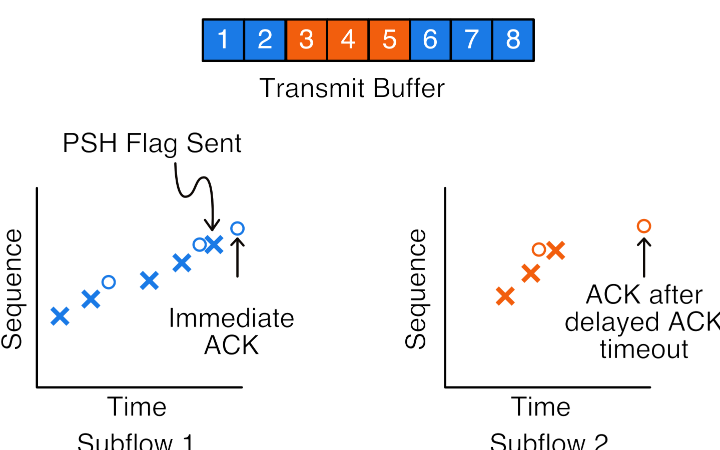


Figure 6.21: Division of a resource across two subflows, and the effect of the PSH flag.

flows. Unfortunately, there is no way to determine this boundary at transmission time. In our example, the sender cannot know that packet 5 will be the last packet transmitted on Subflow 2 from the current object. It has no way of predicting that the next three packets will be transmitted on Subflow 1, due incoming ACKs freeing up space in its congestion window.

One possible solution would be to send immediate ACKs on both subflows when *either* one receives a PSH flag at the receiver. But, if the subflows have different RTTs, it is possible for packet 5 to arrive at the receiver after packet 8. If so, the receiver once again needs to wait for a delayed ACK timeout before sending the corresponding ACK.

Instead, the sender must communicate to the receiver on *both* subflows that it has encountered the end of an object. When the sender transmits a PSH flag on one subflow, it also transmits a special control packet on the second. This packet is an ACK with the PSH flag enabled. In our MPTCP implementation, we use this PSH/ACK packet as a signal to the receiver that it must send an immediate ACK on the second subflow in response. With this mechanism in hand, the end of an object can be ACKed on both subflows without delay. Additionally, it is now safe to remove the delayed ACK timeout duration from the PTO estimation.

Yedugundla *et al.* introduce an optimisation to TLP for MPTCP in [152], in order to tackle higher completion times when tail losses occur on the higher delay path under a "worse" path first scenario. They suggest replicating the probe packet onto the second path as well, using a mechanism similar to opportunistic retransmission. Unfortunately, although this mechanism can plug the hole in the sequence space if the retransmitted packet is in the fact the one that is lost, it doesn't really aid the recovery of the original subflow experiencing tail loss in the first place. For this reason, we do not implement a similar mechanism for our experiments.

6.4.6 Interface Heuristics

As we've seen in Section 6.3.4, if MPTCP chooses to initiate new connections on the higher delay path, it will experience high PLTs across the board. It is just as crucial that the OS makes the correct choice of interface to initiate connections on for single-path TCP. Both TCP and MPTCP will suffer if the higher delay path is selected, resulting in an avoidable increase in

PLT for web traffic. The criteria for selecting the best interface on which to start do matter, but they are outside the scope of this thesis: we assume that the operating system will make the right choice when deciding between interfaces based on observable measurements of the interfaces' delay, bandwidth and loss profiles.

If a choice between interfaces cannot be made at the outset, a browser can possibly make the correct decision after it probes the interfaces on the very first connection it initiates. In other words, if after downloading a web page's main HTML file, a browser determines that the second interface is in fact the best one, it can initiate all subsequent MPTCP connections on that interface.

Finally, we've seen from Section 6.3.4 that one major cause of delay when MPTCP initiates connections on the worst path is the cost of the 3-way handshake on this slower path. Ideally, we should reduce the number of RTTs before the second "better" subflow is up and running. For example, the sender may choose to transmit the `MP_JOIN` option and the 3-way handshake before receiving a DSS option, transmitting data only after the DSS option arrives in the future. Similarly, mechanisms like TCP Fast-Open or QUIC's 0-RTT handshake present an attractive answer if adapted for MPTCP.

Although we propose these mechanisms as possible solutions, a thorough examination of their viability is outside the scope of this thesis, and we leave it to future work. In the next section, we compare single-path TCP over the best interface with MPTCP where the first subflow starts on the best interface.

6.5 MPTCP Improvements Evaluation

In this section we evaluate the impact of the modifications to MPTCP described above. In particular, we wish to know whether the changes are sufficient to fix the problems seen in Section 6.3. Are there cases where modified MPTCP performs worse than unmodified MPTCP? We also wish to understand which modifications have the greatest impact. To answer these questions we repeat the delay and bandwidth mismatch experiments from Section 6.3 using the modified version of MPTCP. The simulation setup is the same, with the same ranges of bandwidth and delay.

6.5.1 RTT mismatch

We begin by repeating the RTT mismatch scenarios from 6.3.2 where the second path has 5x the delay of the first. Figure 6.22 shows how Figure 6.6 changes when we enable the Stream Scheduler (Section 6.4.1), disable Linked Increase (Section 6.4.4), perform QUIC-like out-of-order delivery (Section 6.4.3), use preemption (Section 6.4.2), and incorporate Tail Loss Probes (Section 6.4.5). We compare against single-path TCP with TLP enabled.

From the figure, we see that the dark region of poor PLT at the lower right corner in Fig. 6.6 has greatly improved. For example at 50Mbps/100ms, only 36% of web sites using baseline MPTCP beat or equalled single-path TCP, whereas with the changes 77.9% of sites do so.

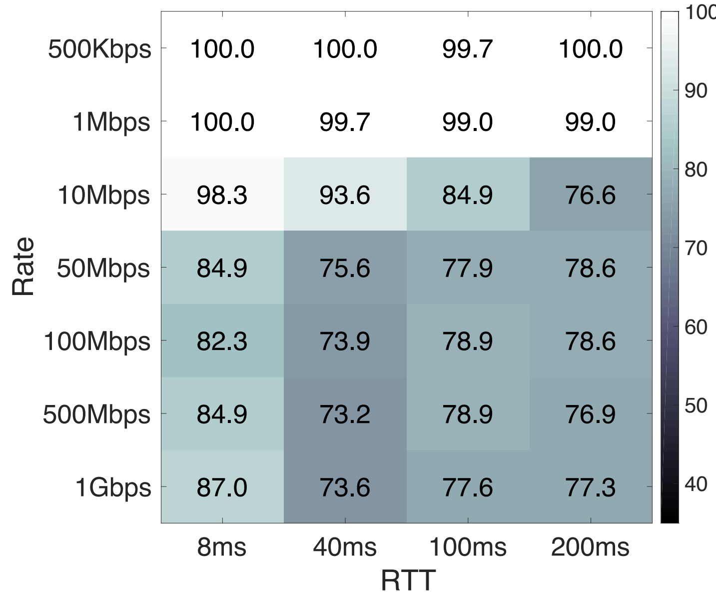


Figure 6.22: 5× RTT on second path: heatmap indicating the percentage of web sites with equal or better PLT using modified MPTCP *vs.* single-path TCP.

Enabling TLP for the delay mismatch case in Figure 6.22 offers only minor improvements when compared to the other optimizations, and only for cases with low bandwidth and RTT, where tail losses are more likely to occur. TLP in general has very little effect on page completion when the two paths have equal bandwidths. Placing resources on a second path which experiences a similar amount of tail loss will not reduce performance compared to single-path TCP. We therefore exclude TLP from our further analysis of the delay mismatch case.

Let us now examine how the MPTCP improvements fare with our example web site pages.github.com from Figure 6.7(b). With the new enhancements to MPTCP, we obtain the timeline plot shown in Figure 6.23. Immediately, we can see that using the Stream Scheduler with preemption produces a PLT of 4.0s, an improvement over both baseline MPTCP and even single-path TCP, which had a PLT of 4.71s. The improvement comes from two sources: first, where baseline MPTCP's RTT scheduler placed packets from critical Resource 43 on the slower path, the Stream Scheduler transmits Resource 43 in entirety on the faster subflow, preventing the increase in PLT. Second, lower priority Resource 10 is preempted by the more critical Resource 2 as shown in the figure. Since Resource 2 is a parent of Resource 43, downloading it before the larger Resource 10 will allow Resource 43 to download earlier as well, thus further reducing PLT.

The example above indicates that the different modifications to MPTCP offer separate contributions to the overall improvement in PLT. As such, we would like to quantify which of the MPTCP modifications are responsible for the largest improvements, and under which *(Bandwidth, RTT)* regimes. To that end, we've selected four representative regimes from the delay mismatch heatmap. Under each regime, we compute the mean PLT across 50 runs for each web site in our corpus, downloaded using single-path HTTP/2. Using these means as a

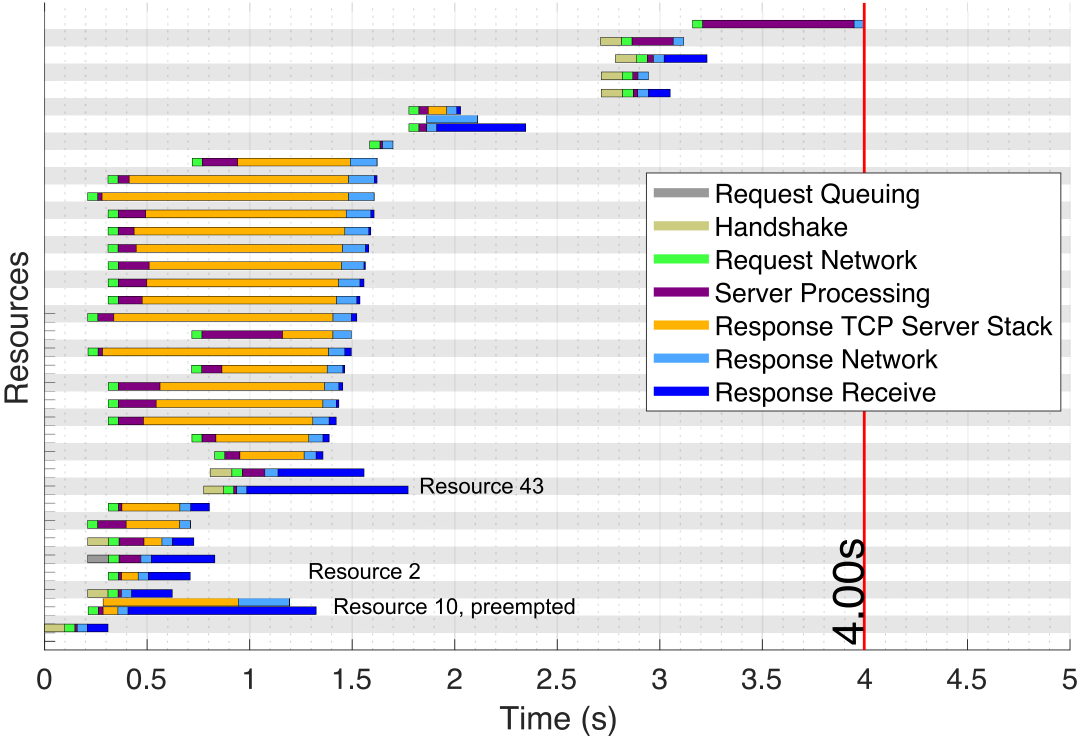


Figure 6.23: Timeline plot for pages.github.com using MP QUIC with Stream Scheduler and preemption on path with 50Mbps bandwidth, 100ms RTT, and $5\times$ RTT on second path.

baseline, we then compute the difference in PLT between the baseline and MPTCP, with our modifications incrementally applied. Figure 6.24 shows CDFs of this difference for the four regimes. Anything to the right of the vertical line at 0s has a larger completion time than single-path TCP on the better path. We discuss the different regimes below.

6.5.1.1 Latency Bound Regimes

Consider first the 50Mbps/100ms regime shown in Figure 6.24(c), which was particularly bad in Fig. 6.6. Under this combination of bandwidth and RTT, most web sites are application limited since the network is fast enough to complete transmitting resources faster than the time it takes to perform other activities like parsing, computation or server processing.

Since downloads are dominated by latency chains, it is unsurprising that MPTCP with HTTP/2 using the default "lowest RTT first" scheduler (labelled *MPTCP HTTP/2 RTT*) does really poorly: 63.9% of the websites do worse than single path TCP. As we incrementally add optimizations, PLT steadily reduces.

The first mechanism we enable is QUIC-lite (*i.e.* out-of-order delivery) but continue using the MPTCP RTT scheduler. As the *MP QUIC RTT* curve shows, the ability to deliver complete resources to the application without waiting for in order delivery at the socket level gives a small win over HTTP/2.

If we also allow higher priority resources to preempt lower priority ones in MP QUIC's transmit buffer (*MP QUIC RTT Preempt*), we see another small reduction, as delivering the

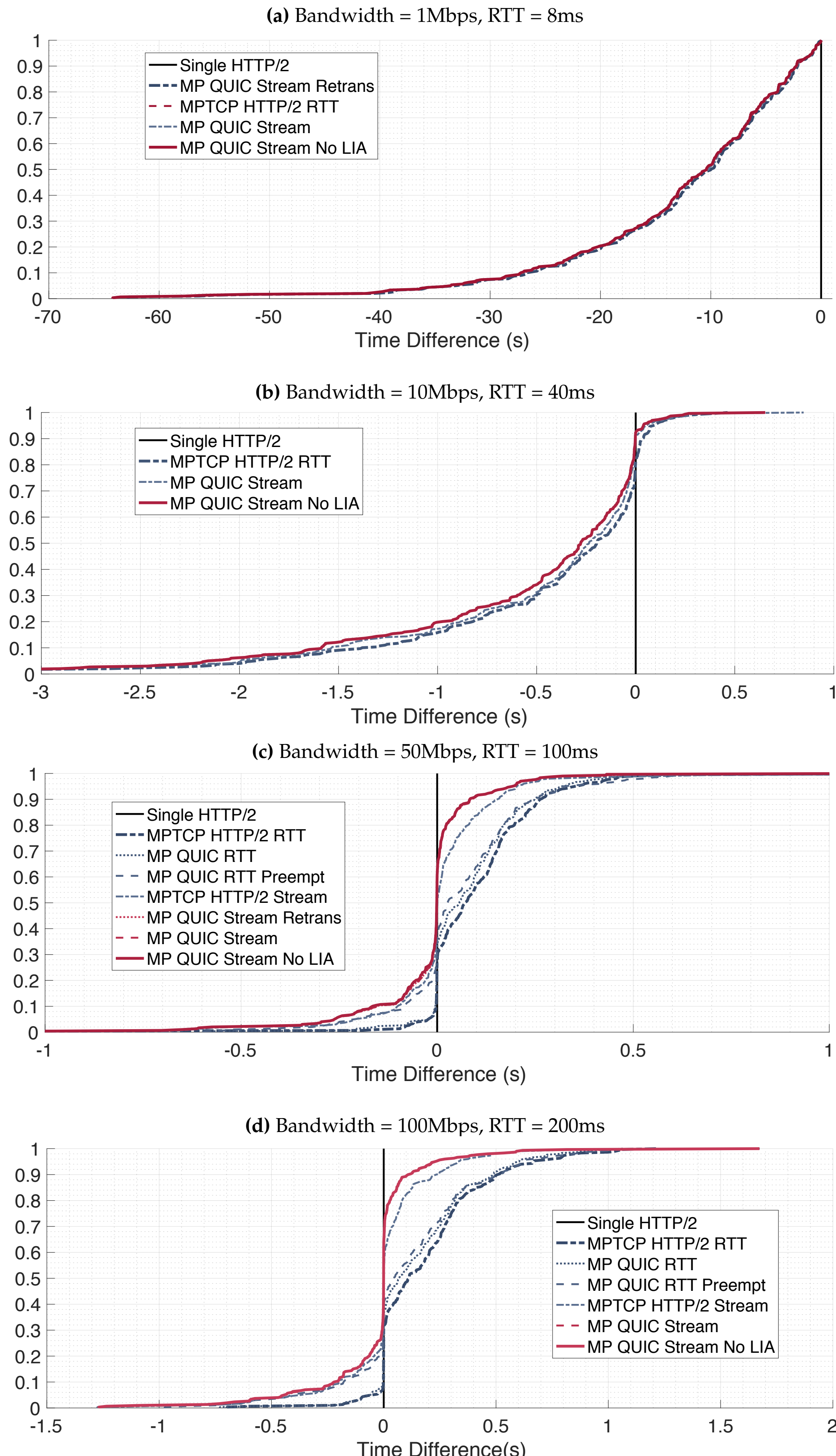


Figure 6.24: CDFs of the difference between the mean PLTs of 300 web sites under the depicted MPTCP regime and single-path TCP. The second path has $5\times$ the RTT.

critical objects earlier improves PLT.

It is only when we apply the Stream Scheduler (with preemption) that we begin to see a sizable improvement. We first look at the case where we use pure HTTP/2 without out-of-order delivery with the Stream Scheduler (*MPTCP HTTP/2 Stream*). Although we get a sizable improvement in PLT, about 50% of cases still do worse than single path.

Finally, we find that QUIC's out-of-order semantics are crucial for making the Stream Scheduler more useful (*MP QUIC Stream*). Enabling out-of-order delivery allows the Stream Scheduler to performs as well as or better than single path TCP in around 78% of the cases.

From the figure, we see that enabling the final two improvements makes very little difference for 50Mbps/100ms, since the two curves for Retransmissions and No LIA lie directly on top of the *MP QUIC Stream* curve. Using *MP QUIC Stream* and allowing retransmissions to be sent on the second path has little effect on PLT, since at a bandwidth of 50Mbps the web downloads experience very little loss. Similarly, disabling LIA only has a minor effect in this configuration, since the bandwidth is large enough that few losses occur, and subflows spend most of their time in slow-start rather than AIMD.

Figure 6.24(d) shows that, once we are in the latency bound regimes, increasing the bandwidth and/or RTT will still produce similar trends in terms of the PLT difference CDF. Once again, using out-of-order delivery, preemption and the Stream Scheduler provide incremental improvements over *MPTCP HTTP/2 RTT* and the optimizations captured by the *MP QUIC Stream* scheme produce the best benefit.

6.5.1.2 Bandwidth Limited Regimes

We have established that for lower bandwidth regimes, simply adding a second path produces a large improvement in PLT, despite the path's higher RTT. Unmodified MPTCP with its default RTT scheduler allows the web download to take advantage of the second path and drives down latency. As Figure 6.24(a) shows, enabling the optimizations encapsulated in *MP QUIC Stream* produces a very small improvement over the status quo. Disabling LIA also produces little visible difference. Therefore we can conclude that enabling the MPTCP improvements by default will help in the latency bound regimes, and will not actively hurt the bandwidth limited regimes.

From the figure, we can see a minor decrease in performance for the *MP QUIC Stream Retrans* curve. Although we expected to achieve a decrease in PLT for high loss regimes if we retransmit lost packets on an alternative subflow, the results indicate that this optimization causes MPTCP to perform even worse than the unmodified status quo. Unfortunately, although the Stream Scheduler checks whether a subflow is idle before using it to retransmit packets, we cannot guarantee that the subflow will remain idle until all retransmissions complete. A new resource may be scheduled on the subflow in the intermittent time, and the retransmissions will in fact delay the delivery of the new resource, therefore increasing PLT.

Examining the full range of bandwidths and RTTs (not shown), we note that *MP QUIC*

Stream Retrans generally reduces the improvement brought about by MP QUIC Stream. Therefore, we can conclude that this optimization fails to improve PLT values with respect to single-path TCP.

6.5.1.3 Boundary Regimes

Ultimately, Section 6.5.1.1 has shown that our improvements prevent MPTCP from hurting PLTs for the 50Mbps/100ms scenario, but even with all improvements added, MPTCP still does not *beat* single-path TCP. This should not be a surprise, since the second subflow's RTT is 400ms greater than the first. With this much delay mismatch, it takes a really large object to be worth placing it on the higher RTT path. Let's examine a scenario where the difference is less extreme.

Fig. 6.24(b) shows the 10Mbps/40ms scenario, where the delay difference between the two paths is 160ms. Although MPTCP with the default RTT scheduler performs reasonably well in this regime, using using the Stream Scheduler with out-of-order delivery improves PLT even further, with a higher percentage of web sites to the left of the black line.

Disabling LIA in this case achieves a significant decrease in PLT. Upon examination, we note the difference in loss rates for subflows on the two paths in this regime. Figure 6.12 shows that because we size the network buffers proportional to the BDP, the first path has a higher loss rate, whereas the second with an RTT of 200ms suffers no losses on average. Therefore, subflows on the first path grow their windows with LIA after encountering loss, whereas subflows on the second path spend most of their time in slow-start. As a result, the congestion window of a subflow on the first path will often be much smaller than its counterpart on the second path, which continues to grow exponentially under slow-start.

As we've seen in Section 6.3.3, LIA grows the window on the first path proportionate to its share of the total window. As such, the "better" low latency path in fact grows its window much slower than regular AIMD because the window on the second path is larger. LIA's goal is to shift traffic onto the path with the lower loss probability. Unfortunately, this strategy overlooks the fact that the lossless path is in fact the one with the larger latency, causing an increase in PLT. Disabling LIA eliminates this issue.

Finally, we show the difference in mean PLT improvement between modified and baseline MPTCP in Table 6.1. For each *(Bandwidth, RTT, website, run)* tuple, we find $(PLT_{TCP} - PLT_{MPTCP})$, and then obtain $\bar{x}$ by computing the overall mean of the difference across the $300 \times 50 = 15000$ samples. σ is the sample standard deviation. The percentage improvement of mean multipath PLT over mean single-path PLT is computed using $\frac{\sum PLT_{TCP} - PLT_{MPTCP}}{\sum PLT_{TCP}}$ for the samples. We found that the difference between modified MPTCP and single path TCP is significant (computed using confidence intervals), likewise between baseline MPTCP and single path TCP. Using one way ANOVA to compare modified and baseline MPTCP, we determined the p-values shown in the table. Each p-value indicates the likelihood of the null hypothesis, *i.e.* that the means of both groups are the same for the given bandwidth and RTT. If

Mode		PLT_{tcp}(s)	MP QUIC Stream No LIA TLP			MPTCP HTTP/2 RTT			p-value
Rate	RTT		$\bar{x}$ (s)	σ	% Imp.	$\bar{x}$	σ	% Imp.	
500Kbps	8ms	54.407	26.583	21.959	48.9	26.601	22.074	48.9	0.944
	40ms	54.872	25.799	21.783	47.0	25.173	21.606	45.9	0.016
	100ms	56.490	25.284	22.309	44.8	24.352	21.606	43.1	0.001
	200ms	57.867	22.917	21.474	39.6	21.777	20.673	37.6	0.000
1Mbps	8ms	27.527	13.107	10.913	47.6	13.023	10.955	47.3	0.512
	40ms	28.264	12.692	11.063	44.9	12.159	10.719	43.0	0.000
	100ms	29.260	11.452	10.720	39.1	10.837	10.330	37.0	0.000
	200ms	30.650	9.209	9.975	30.0	8.473	9.687	27.6	0.000
10Mbps	8ms	3.993	0.905	1.020	22.7	0.829	0.959	20.8	0.002
	40ms	4.558	0.561	0.835	12.3	0.435	0.769	9.6	0.000
	100ms	6.103	0.366	1.016	6.0	0.193	0.995	3.2	0.000
	200ms	8.622	0.273	1.635	3.2	0.066	1.646	0.8	0.000
50Mbps	8ms	2.460	0.035	0.138	1.4	0.018	0.130	0.7	0.469
	40ms	3.106	0.024	0.257	0.8	-0.028	0.223	-0.9	0.030
	100ms	4.469	0.019	0.236	0.4	-0.107	0.277	-2.4	0.000
	200ms	7.071	0.021	0.337	0.3	-0.190	0.383	-2.7	0.000
100Mbps	8ms	2.376	0.004	0.095	0.2	-0.006	0.092	-0.3	0.655
	40ms	2.994	0.004	0.120	0.1	-0.042	0.127	-1.4	0.059
	100ms	4.404	0.022	0.167	0.5	-0.103	0.159	-2.3	0.000
	200ms	7.022	0.025	0.278	0.4	-0.185	0.314	-2.6	0.000
500Mbps	8ms	2.329	-0.006	0.051	-0.3	-0.009	0.051	-0.4	0.895
	40ms	2.958	0.001	0.084	0.0	-0.048	0.099	-1.6	0.040
	100ms	4.381	0.019	0.164	0.4	-0.106	0.160	-2.4	0.000
	200ms	7.004	0.021	0.255	0.3	-0.187	0.288	-2.7	0.000
1Gbps	8ms	2.325	-0.007	0.045	-0.3	-0.009	0.044	-0.4	0.922
	40ms	2.957	0.000	0.084	0.0	-0.048	0.101	-1.6	0.044
	100ms	4.379	0.019	0.164	0.4	-0.106	0.160	-2.4	0.000
	200ms	7.002	0.021	0.254	0.3	-0.187	0.288	-2.7	0.000

Table 6.1: PLT difference of modified and baseline MPTCP with respect to single-path TCP for the delay mismatch scenario. PLT_{TCP} is the mean PLT in seconds for the single-path case downloaded on the best path. $\bar{x}$ is the mean over all values of corresponding $(PLT_{TCP} - PLT_{MPTCP})$ in seconds, σ is the sample standard deviation, and the % Imp is $\frac{\sum(PLT_{TCP}-PLT_{MPTCP})}{\sum PLT_{TCP}} \times 100\%$. The p-value represents the probability that the mean PLTs of modified and baseline MPTCP are not significantly different, and is the result of computing one-way ANOVA across the two groups.

we choose a significance level of 5%, modified MPTCP generally achieves a statistically significant reduction in PLT when compared to the baseline. The exception occurs for RTTs of 8ms, where modified and baseline MPTCP are not significantly different.

From the table, we can see that adding a second, albeit higher latency path produces a large improvement in PLT for both modified and unmodified MPTCP when bandwidths are lower then 50Mbps, very much like the equal path case. But the improvement is slightly larger for modified MPTCP because of the beneficial effect of the Stream Scheduler. As soon as we move to bandwidths where web downloads are application limited, *i.e.* larger than 50Mbps, the improvements diminish. We know from the equal path case that the best we can do in these high bandwidths is an improvement of about 5-6% over baseline TCP. Instead, baseline MPTCP actually increases PLT over the single-path case, by up to about 3% for long RTTs.

Only by adding our modifications can we bring PLT down for these high bandwidths. When the RTT factor is much larger on the second path, the best we can do is avoid using the bad path, so the percentage improvement goes to zero as MPTCP becomes equivalent to single-path TCP on the faster path.

With the exception of 10Mbps, modified MPTCP does not provide a significant win over the baseline when the RTT is 8ms, confirming our earlier results. In fact, when the bandwidths are 500Mbps and 1Gbps, modified MPTCP does worse than single path, since tail losses come into play and TLP fails to mitigate them sufficiently.

6.5.1.4 RTT Ratios

In our experiments, we have investigated the behaviour of multipath web downloads when the second path had five times the RTT. Although using a factor of five may seem rather extreme, especially when we have RTTs of 200ms on the first path and 1s on the second, we had chosen this RTT ratio to illustrate the problems of sending critical resources on a significantly slower path. We have shown that our MPTCP modifications, namely the Stream Scheduler, improve PLTs in this scenario. We explore whether these improvements continue to hold as we vary the RTT ratios between the two paths.

In Figure 6.25(e), we choose several representative *(Bandwidth, RTT)* regimes and examine the effect of downloading web pages on two paths with varying ratios between their RTTs. For each RTT factor, we plot the percentage improvement in mean PLT with respect to single-path TCP. We compute this percentage across all web sites for for both modified and baseline MPTCP running over the chosen path configuration. A factor of one indicates that the paths have equal RTT. Although the RTT factors on the higher end of the scale are rather unrealistic, we wish to test the boundaries of our design.

From the figures, we observe that modified MPTCP generally outperforms the baseline for most RTT ratios between the two paths. Modified MPTCP produces a larger reduction in PLT even for cases where the RTT factor is small, *e.g.* with the second path having two or three times the RTT of the first path. We also note that the percentage improvement of both modified and baseline MPTCP over single-path TCP declines as the RTT factor grows. The larger the delay on the second path, the less likely the path scheduler will choose it for transmitting packets. The decline is steeper when the web downloads are not bandwidth limited, for example when the bandwidth is 10Mbps (Figures 6.25(c) and 6.25(d)) in contrast to 1Mbps (Figure 6.25(a) and 6.25(b)).

If the latency is quite large and dominates the web download, as the RTT factor increases, baseline MPTCP will perform worse than the single-path case, *i.e.* the percentage improvement dips below zero. We can see this effect clearly when the RTT is 200ms on the faster path, as shown in Figure 6.25(e). Note that when the RTT factor becomes really large in this setting, the baseline MPTCP curve tends towards zero again. Here, the RTT on the second path becomes large enough that sending a 3-way handshake on the second path is prohibitive, and can con-

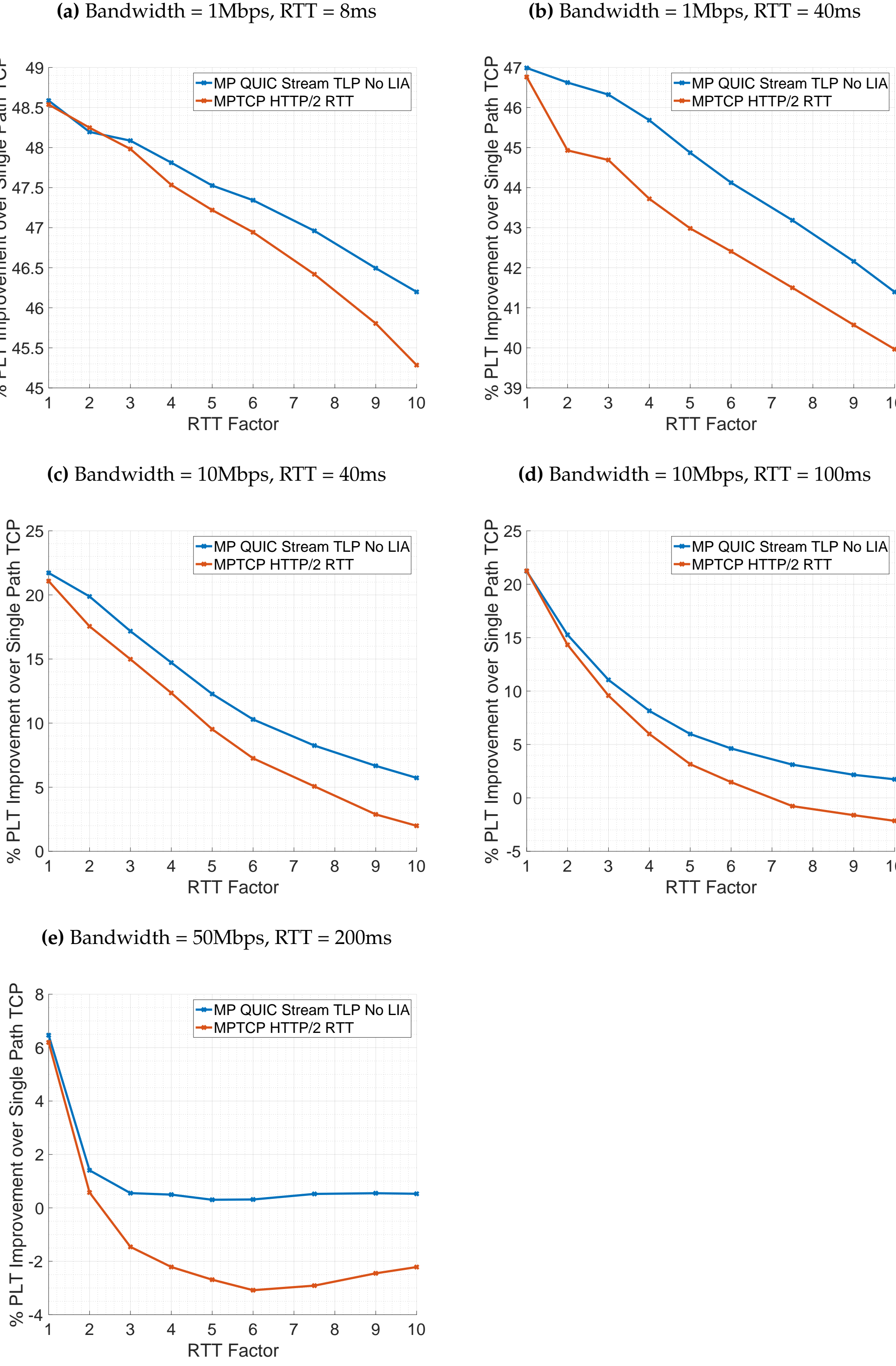


Figure 6.25: The percentage improvement in mean PLT over single path when using modified or baseline MPTCP with the RTT mismatch case. The factor multiplying the RTT on the slower path is shown on the x-axis. Each figure's title shows the RTT on the faster path.

sume the entire download time. As for modified MPTCP, the best its Stream Scheduler can do is prevent any packets from using the higher RTT path. By ignoring the slow path entirely, the exchange is reduced to the single-path case. As a result, we can see from the figure that modified MPTCP does not improve the PLT over the single-path case when the RTT is very large – as the RTT factor increases, the PLT improvement goes to zero as expected. As a general result, even as the ratio of RTTs between the paths grows, the modifications we've applied to MPTCP prevent it from performing worse than single-path TCP.

6.5.2 Bandwidth Mismatch

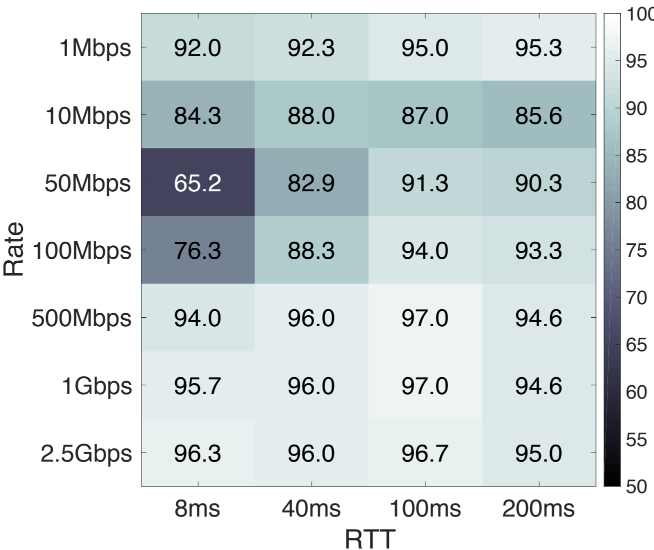


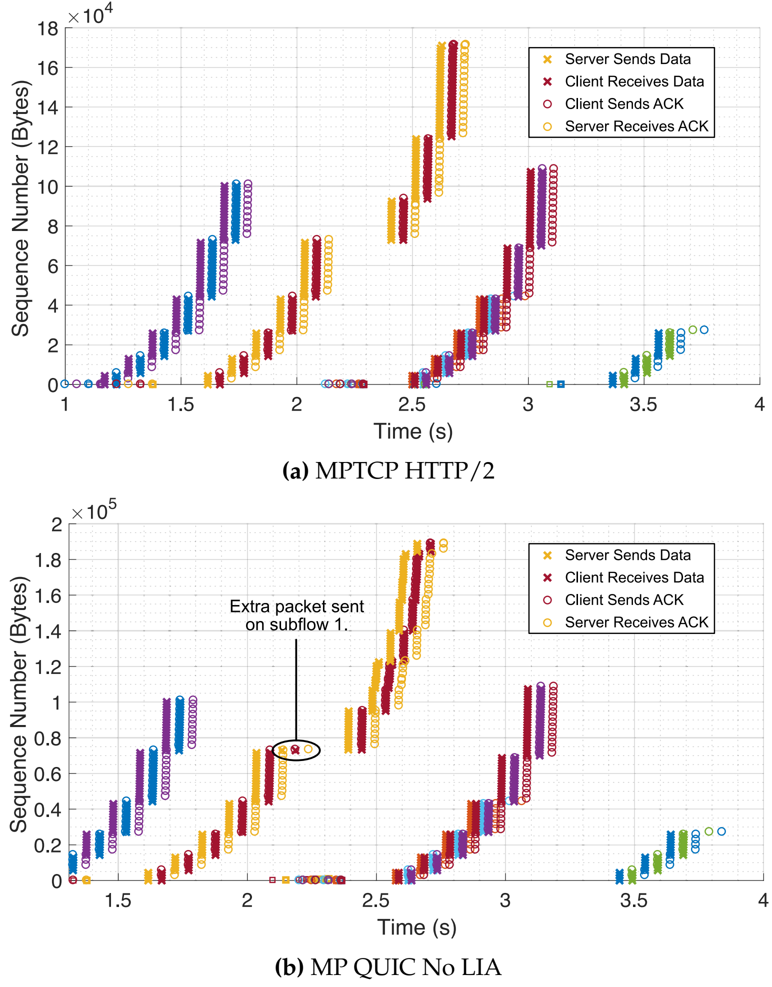
Figure 6.26: $\frac{1}{5}$ bandwidth on second path: heatmap indicating the percentage of web sites with equal or better PLT with modified MPTCP vs single-path TCP (with TLP). We use MP QUIC with no LIA and TLP.

What about the bandwidth mismatch case from Figure 6.8? We repeat the corresponding experiments with our modified MPTCP implementation, disable LIA and enable TLP. We show the results in Figure 6.26.

There are noticeable improvements in the upper right region of the heatmap, mainly attributed to disabling LIA. Additionally, the top half of the RTT=8ms column shows improvement in page completion time due to enabling TLP, and reducing the penalty of tail loss. Unfortunately, even with TLP enabled, MPTCP still performs relatively poorly in the 50Mbps/8ms regime. Remember, this regime is particularly problematic because we are comparing a case where we place some resources on a path which experiences tail loss, with a single-path TCP case that experiences zero tail loss. The key here is recognizing that tail loss has not been eliminated — tail loss probes only manage to mitigate its effects. The TLP algorithm still requires that a subflow wait a PTO timeout before transmitting a probe packet, and Fast Retransmit itself also consumes a non-negligible amount of time. These delays accumulate to increase the PLT.

We observe a slight dip in performance for modified MPTCP compared to the baseline

for some high bandwidth and high RTT regimes. The cause of this dip can be best explained using an example. Figure 6.27 shows a time sequence plot of the flows on the first path for suning.com downloaded at 50Mbps/100ms, one of the problematic regimes. In the figure, we compare two scheduling schemes: *MPTCP HTTP/2 RTT* in Figure 6.27(a) and *MP QUIC Stream No LIA* in Figure 6.27(b). At this bandwidth, flows spend all of their time in slow-start. The discrepancy between the two schemes manifests as a single extra packet sent on the first path by the Stream Scheduler, as opposed to on the second path under MPTCP's RTT scheduler. This packet is the last packet of the resource.



(a) MPTCP HTTP/2

(b) MP QUIC No LIA

Figure 6.27: Time/sequence plot of flows on Path 1 for suning.com downloaded on 50Mbps/100ms with $\frac{1}{5}$ bandwidth on second path.

The Stream Scheduler first attempts to transmit the labeled packet when some space frees up for the subflow on Path 2. According to the scheduling algorithm, it compares $\frac{\text{RTT}_1}{2}$ with $\frac{\text{RTT}_2}{2}$, and finds the second value to be larger, since the second subflow experiences more queuing delays due to its lower bandwidth. Therefore, the scheduler chooses to wait until Subflow 1 is available before it can send the selected packet. Unfortunately, MPTCP must wait a full RTT before Subflow 1 receives an ACK, frees up some window space, and becomes available again. The wait for an extra RTT causes delays and extends PLT — the higher the RTT, the

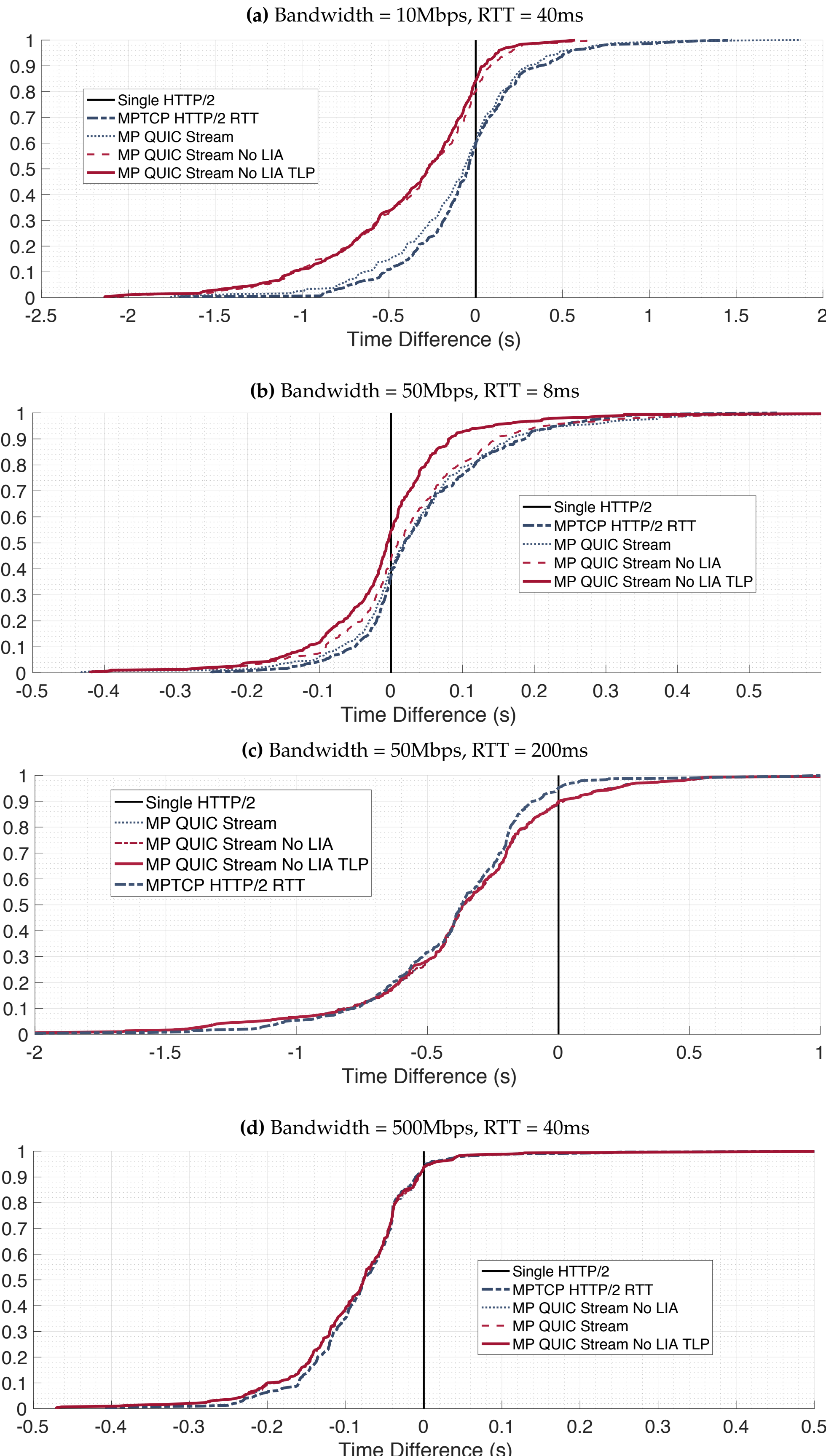


Figure 6.28: CDFs of the difference between the mean PLTs of 300 web sites under the depicted MPTCP regime and single-path TCP. The second path has $\frac{1}{5}$ the bandwidth.

worse the delay. The scheduler has made the wrong choice because it doesn't know when the next ACK will arrive on Subflow 1.

The simplicity of the scheduler is therefore detrimental for these scenarios where flows remain in slow-start, and one subflow has a slightly larger RTT. We could modify the scheduler to keep track of when the left hand side of the congestion window has been transmitted in slow-start, and use this information in the completion estimator to improve its accuracy, but at the cost of increasing the complexity of the design. Figure 6.28(c) shows the CDF of the difference between mean PLTs for MPTCP and single-path TCP, for web sites downloaded with 50Mbps/200ms regime. Although, the CDF shows that more web sites have lower PLTs than single-path TCP under *MPTCP HTTP/2 RTT* than *MP QUIC Stream No LIA TLP*, the two curves are nearly congruent for about 70% of the web sites. The dip in performance only appears for the remaining 30%.

For cases where we see an improvement in PLT, we now zoom into three representative *(Bandwidth, RTT)* regimes to see where these improvements come from, shown in Figure 6.28. We discuss each regime in the following sections.

6.5.2.1 High Loss

Figure 6.28(a) shows a case where the web downloads are bandwidth limited, but experience different loss on each subflow, such that LIA may cause the window of the faster subflow to grow too slowly. Unlike in the delay mismatch scenario (Figure 6.24), using the Stream Scheduler with QUIC-like out-of-order-delivery only offers very small improvements to PLT. As expected, disabling LIA (*MP QUIC Stream No LIA*) offers the best improvement in PLT, as it helps MPTCP's congestion window grow faster. Enabling TLP only has a small effect in these regimes, because the subflows experience little tail loss.

6.5.2.2 Tail Loss

When subflows are application limited and experience loss, they are particularly susceptible to losing the tail of a transmitted resource. We examine the case of 50Mbps/8ms, which we have seen to be particularly problematic. Figure 6.28(b) shows the corresponding CDF, which indicates that although *MP QUIC Stream* and *MP QUIC Stream No LIA* offer incrementally minor improvements, the biggest win over baseline MPTCP comes from enabling TLP.

6.5.2.3 No Loss

When both paths are sufficiently provisioned, and when neither of them experience any losses, we obtain the behaviour depicted in Figure 6.28(d). Unmodified MPTCP provides an improvement in PLT over single-path TCP for nearly 96% of the web sites. The MPTCP optimizations can offer very minor improvements here, since the problems they are designed to tackle are minimal for this regime.

Finally, we show the difference in mean PLT improvement between modified and baseline MPTCP in Table 6.2, with values obtained using the same techniques used for Table 6.1. This

Mode		PLT_{tcp}(s)	MP QUIC Stream No LIA TLP			MPTCP HTTP/2 RTT			p-value
Rate	RTT		$\bar{x}$ (s)	σ	% Imp.	$\bar{x}$	σ	% Imp.	
1Mbps	8ms	27.527	3.867	3.873	14.0	3.693	4.004	13.4	0.416
	40ms	28.264	4.096	3.921	14.5	3.786	3.981	13.4	0.151
	100ms	29.260	4.186	3.921	14.3	3.514	3.939	12.0	0.002
	200ms	30.650	4.250	4.026	13.9	2.313	3.670	7.5	0.000
10Mbps	8ms	3.993	0.322	0.494	8.1	0.196	0.446	4.9	0.000
	40ms	4.558	0.390	0.558	8.6	0.057	0.571	1.2	0.000
	100ms	6.103	0.599	1.135	9.8	0.296	1.237	4.9	0.000
	200ms	8.622	0.870	2.101	10.1	0.703	2.069	8.2	0.001
50Mbps	8ms	2.460	-0.001	0.158	-0.0	-0.050	0.159	-2.0	0.031
	40ms	3.106	0.095	0.317	3.1	0.085	0.281	2.8	0.679
	100ms	4.469	0.193	0.303	4.3	0.198	0.266	4.4	0.846
	200ms	7.071	0.374	0.486	5.3	0.400	0.405	5.7	0.498
100Mbps	8ms	2.376	0.005	0.104	0.2	-0.017	0.112	-0.7	0.322
	40ms	2.994	0.075	0.174	2.5	0.080	0.150	2.7	0.826
	100ms	4.404	0.201	0.210	4.6	0.200	0.179	4.5	0.992
	200ms	7.022	0.400	0.417	5.7	0.401	0.346	5.7	0.969
500Mbps	8ms	2.329	0.019	0.065	0.8	0.017	0.062	0.7	0.928
	40ms	2.958	0.088	0.109	3.0	0.080	0.100	2.7	0.755
	100ms	4.381	0.220	0.198	5.0	0.209	0.170	4.8	0.687
	200ms	7.004	0.414	0.394	5.9	0.405	0.335	5.8	0.798
1Gbps	8ms	2.325	0.019	0.048	0.8	0.018	0.051	0.8	0.986
	40ms	2.957	0.089	0.107	3.0	0.081	0.098	2.7	0.742
	100ms	4.379	0.222	0.198	5.1	0.210	0.172	4.8	0.666
	200ms	7.002	0.415	0.388	5.9	0.404	0.336	5.8	0.773
2.5Gbps	8ms	2.324	0.020	0.047	0.9	0.020	0.049	0.9	1.000
	40ms	2.956	0.090	0.106	3.1	0.082	0.100	2.8	0.717
	100ms	4.378	0.223	0.196	5.1	0.210	0.171	4.8	0.629
	200ms	7.001	0.416	0.386	5.9	0.404	0.336	5.8	0.764

Table 6.2: PLT difference of modified and baseline MPTCP with respect to single-path TCP for the bandwidth mismatch scenario. PLT_{TCP} is the mean PLT in seconds for the single-path case downloaded on the best path. $\bar{x}$ is the mean over all values of corresponding to $(PLT_{TCP} - PLT_{MPTCP})$ in seconds, σ is the sample standard deviation, and the % Imp is $\frac{\sum(PLT_{TCP}-PLT_{MPTCP})}{\sum PLT_{TCP}} \times 100\%$. The p-value represents the probability that the mean PLTs of modified and baseline MPTCP are not significantly different, and is the result of computing one-way ANOVA across the two groups.

time, adding a second path achieves a smaller improvement over single path, since the second path has $\frac{1}{5}$ the bandwidth of the first path. Disabling LIA for low bandwidths of 10Mbps and less produces a decrease in PLT, and a larger improvement over baseline MPTCP when compared to single-path.

For larger bandwidths, the improvement diminishes as expected due to web downloads becoming application limited. The mean PLTs of modified and baseline MPTCP are no longer significantly different. Since the RTTs of the two paths are equivalent, the percentage improvement is more in line with the equal paths case. MPTCP with tail loss probes mitigates the decrease in performance experienced by baseline MPTCP for bandwidths 50Mbps and 100Mbps, and an RTT of 8ms.

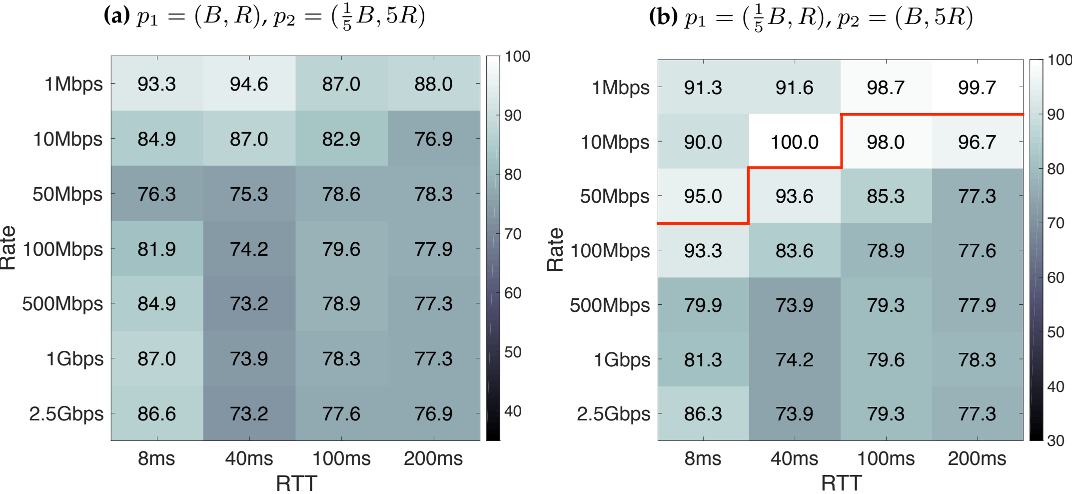


Figure 6.29: Percentage of web sites with equal or better PLT using improved MPTCP *vs.* single-path TCP on the "best" path. Above the red line, $(B, 5R)$ is the best path. Below the read line, $(\frac{1}{5}B, R)$ is best. We use *MP QUIC Stream* with LIA and TLP.

6.5.3 Combination Paths

We finally turn our attention to the combination paths case, *i.e.* scenarios where both paths have a bandwidth and/or an RTT multiplier. Once again, we will only consider the cases where MPTCP initiates connections on the "best" path.

For the bandwidth/RTT regime shown in Figure 6.29(a), which combines pathologies experienced by both RTT mismatch and bandwidth mismatch scenarios, the proposed MPTCP enhancements offer a significant improvement over the status quo. Similarly, for Figure 6.29(b), MPTCP's enhancements improve PLT performance for a larger proportion of web sites. Above the red line, disabling LIA provides the best benefit, and below the red line the Stream Scheduler prevents critical resources from being scheduled on the high latency path.

6.6 Discussion

We set out to examine how multipath affects PLT in today's web. While beneficial in general, we discovered and presented multiple cases where the current state of the art in multipath transport (MPTCP) actually hurts PLT. Instead of giving up on multipath transport, we demonstrated that it is possible to address all of the discovered issues and modify MPTCP to make it more suitable for today's internet.

Despite the measurable benefits of our proposed optimisations, there is still room for improvement. In the following sections, we highlight several areas which require some additional focus.

6.6.1 Completion Estimation Inaccuracies

Using the Stream Scheduler reduces PLTs, but from Figure 6.22, we observe that for around 20-25% of web sites that experience high RTTs, MPTCP still performs worse than single-path TCP. Errors in the Completion Estimator are mostly to blame. If the estimator incorrectly computes

the completion time for a resource on a particular path, the Stream Scheduler may place traffic on the high delay path even though doing so would cause a resource to complete at a later time. From Section 6.4.1.2, we deduce that the most common causes of estimator error are losses, and incorrect estimation of path RTT.

Packet Loss is problematic for several reasons:

- it unpredictably delays resource completion time despite our use of NACKs (emulating SACK) for faster recovery.
- it may cause timeouts if tail losses occurs.
- it can cancel the effect of prioritisation and preemption if a critical resource is scheduled earlier, and unluckily falls at the end of slow-start where a large proportion of packets are lost.

A better predictor like [146] may help incorporate packet loss into the estimator model, but at the cost of greater complexity, since it requires some measure of loss probability to estimate TCP latency.

Another solution is to use a transport protocol that recovers the congestion window quickly from loss, such as TCP CUBIC. Nevertheless, a well defined load balancing algorithm for CUBIC to work with MPTCP does not exist, and flows with losses will still experience delays due to retransmissions. BBR, on the other hand, seems well suited for latency sensitive traffic like web downloads. BBR generally experiences fewer packet losses, because its control loop always tries to send at the bottleneck bandwidth rate, and never more than a BDP worth of data. Using BBR's reasonably accurate measure of a subflow's fair share of the bottleneck bandwidth B_{btl} may prove a better estimate of the throughput to be used in the completion estimator.

Additionally, BBR's startup behaviour attempts to avoid growing the congestion window beyond the available buffers, and actively drains any potential queues, therefore reducing the probability of experiencing a burst of packet losses at the end of slow-start. Given these properties, BBR seems like a good candidate for multipath web downloads. Unfortunately, the BBR specification has only been made public recently, towards the conclusion of this work. Additionally, BBR as of yet does not have any load balancing multipath component either. We leave the design and evaluation of multipath BBR for web traffic to future work.

If packet drops are an obstacle to obtaining correct completion time estimates, with explicit congestion notification (ECN) [153, 154], a host can perform congestion control without using packet drops as a signal of congestion. ECN requires the support of an appropriate active queue management (AQM) scheme. When queues in the network detect incipient congestion, they set the correct bits in the TCP header to signal congestion to the receiver. The receiver in turn echoes the ECN bits to the sender, which will accordingly cut its transmission rate. ECN is an attractive solution for minimising packet losses, thereby reducing the inaccuracies

in the completion time estimator. Although many modern operating systems support ECN, the underlying network infrastructure must be correctly configured with AQM such as RED queues as well, which may not always be the case.

Finally, since retransmitting lost packets takes too long and throws off the estimator, we may be better off forgoing retransmissions altogether. A reasonable amount of FEC on transmitted packets may be a win if it allows a receiver to reconstruct missing packets without incurring retransmission delays. FEC would be especially helpful at the end of web resources, in order to recover from tail loss. QUIC's simple XOR-based FEC scheme, where an FEC packet contains the parity over packets in an FEC group, may be a good start. But packet losses often occur in bursts, and QUIC's FEC only allows the recovery of one packet loss per FEC group [21].[4]

Perhaps a better but more ambitious forward error correction scheme would be to use rateless codes [155], which would allow a server to transmit a continuous stream of coded packets to the client until the client can manage to decode them. But rateless codes require a complete redesign of the transport protocol, and a better understanding of the trade-off between the resulting coding overhead and latency.

Incorrect RTT Estimation occurs when an RTT value reported by TCP to the completion estimator at a particular instant poorly reflects its future value, which may increase due to the buildup of network queues. Network buffers filling during slow-start and after a loss cause the RTT to grow beyond the estimates. Once again, BBR may help reduce this phenomenon and improve the estimator's operation. BBR, by virtue of its design, avoids forming large queues, and growing the RTT. As soon as a queue starts building, and RTT growing, BBR clamps down on the send rate. Finally, applying QUIC's more accurate loss signalling and RTT measurements would improve the completion estimator as well.

6.6.2 Subflow Setup Delays

As we've seen in Section 6.3.2, after setting up a connection on an initial path, MPTCP cannot begin using the second path immediately, but must wait for another 3-way handshake on that path. When the second path has a high RTT, it may take a shorter amount of time to fully transmit an object on the first path than establish the 3-way handshake on the second. Waiting for a second handshake is wasteful, and represents a missed opportunity where we could have sent on the second subflow if it had been ready earlier during object transmission.

Ideally, after an initial handshake, we'd like the server to begin using the two subflows on the two available paths immediately. The full handshake is required in MPTCP because it uses TCP as a substrate, which needs to traverse middleboxes, and has a limited option space of 40 bytes. On the other hand, if we use UDP based transport like QUIC, we can forgo the 3-way handshake and use a similar mechanism to QUIC's 0-RTT handshake. The client would

[4] Google has removed FEC support from QUIC after noticing that in only achieved negligible improvements in search latency, but increases in video latency.

have to transmit all its valid IP addresses in its initial handshake message, or perhaps send a handshake on the two paths simultaneously. After authenticating the client in a previous round, the client now owns a server assigned token for each IP address advertised, and any future communications with the server would require 0-RTTs to begin receiving data on both subflows.

6.6.3 Content Sharding

With servers sharding their content across multiple origins, prioritising resources can only happen on the local scale within each server's transmit queue. Therefore, the Stream Scheduler may often have a limited list of resources to choose from when attempting to select a viable resource to transmit using Lookahead. We do not examine the effects of unsharding on web page completion times for MPTCP, but we do expect a general reduction in PLTs when sharded domains are unified, since unsharding allows a web-aware scheduler to reason about resource priorities more efficiently.

6.6.4 Stack Changes

As readers will be apt to point out, our study that had begun as a modification to the MPTCP design, has ultimately veered away from stream-based in-order delivery. In fact, we've shown that recent advances in web-oriented transport protocols (e.g., QUIC), even though designed for single path, are highly beneficial to multipath transport as well. This is encouraging—extending QUIC, which is a userland protocol, to use multiple paths is a viable and possibly easier alternative to modifying MPTCP to better handle web transfers.

Nevertheless, if we do insist on modifying MPTCP in the kernel, what necessary changes must we perform? First, the application layer must mark each resource it transmits to the transport layer with its stream/resource ID and its priority. These two will allow MPTCP to prioritise packets in its transmit socket buffer, and perform preemption as necessary. MPTCP must keep track of resource boundaries, to allow for the correct operation of the Stream Scheduler. MPTCP must also append a QUIC's Stream Frame to any segments it transmits to allow for reassembly in the application layer. At the client, we need to modify the receive buffer to pass packets up to the application layer immediately, in order to support out-of-order delivery. Because these MPTCP improvements must be applied in *both* the client and the server, the barrier to adoption becomes even higher.

It is therefore easier to use an application layer protocol like like QUIC, which has access to the necessary application layer information like resource IDs, boundaries, and prioritisation. QUIC's properties render it quite attractive as a blueprint for multipath web transport, not only for its its ease of adoption and extension, but also due to techniques like out-of-order delivery, 0-RTT handshakes, and FEC.

6.6.5 Energy

It is an open question as to whether improving PLT by using multipath transport also improves power consumption. While both radios operate, increasing power consumption, the user spends less time waiting with the screen on. Such concerns are outside the scope of this work, but should be examined in the future.

Chapter 7

Conclusions

Long web page load times can frustrate users and cause them to lose confidence in the web site's brand and possibly switch to a competing service. In response, there is a concerted interest in developing better web and transport protocols that facilitate faster page loads, and we provided an overview of some of these mechanisms in Chapter 2. At the same time, the spread of devices that support two or more interfaces has prompted the design of MPTCP as a transport protocol that pools bandwidth across these paths. MPTCP was originally evaluated for bulk transfers, which are bandwidth limited. In this work, we set out to examine MPTCP's impact on web page load time, which is often latency bound.

Our pursuit of a framework for evaluating a large number of web pages downloaded under different network conditions, server architectures, and web protocols culminated in the design of PCP, described in Chapter 3. In addition to allowing reproducible experiments, PCP combines dependency graphs generated from real web traces with an emulator to model web page downloads. PCP incorporates server processing delays sampled from our real traces, and models JavaScript timers and events in web pages, to closer approximate the operation of real web pages.

We confirmed previous observations about current web and transport protocols [51–53, 56] in Chapter 4. In particular, we showed that HTTP/2 using one connection and a parallel server architecture (that returns objects out of order) provides the best reduction in PLT when compared with HTTP/1.1. But HTTP/2 will still suffer under high-loss environments due to HOL blocking in the transport layer. Delivering packets out of order to the application layer in the style of QUIC delivers small gains in light of packet loss. Although resource prioritisation alleviates some of HTTP/2's shortcomings, its benefits are not great because prioritisation only happens locally at each server, and web page designers already prioritise their resources to optimise "above-the-fold" page speeds.

Unsharding web domains has been hailed as a possible solution for improving HTTP/2's performance [23]. We investigated the effect of this recommendation is in Chapter 5. We discovered that the advice to unshard is sound, yet, through a series of case studies, we showed that unsharding does not always produce a win. By coalescing several connections to multiple servers into one connection to a single unsharded server, HTTP/2 becomes more susceptible

to the pitfalls of transmitting all of its data on a single, possibly lossy, connection. Once again, prioritisation may help in these cases, in addition to a transport protocol that recovers quickly from loss.

Finally, to fully quantify the effect of delivering web pages on multiple paths, in Chapter 6 we presented a thorough investigation of downloading 300 popular web sites using HTTP/2 over MPTCP. Our goal was to validate whether MPTCP can reduce web PLTs when compared with single-path TCP running on the best path. By examining the case where both paths have equal bandwidth and RTT, we have presented the maximum possible reductions in PLT when enabling MPTCP. Most notably, in the range of 1Mbps to 50Mbps, which represents a typical working range for WiFi and 3G deployments, enabling MPTCP appears to be beneficial since it produces user-perceivable reductions in PLT. Conversely, we uncovered several limitations of downloading web pages using MPTCP on paths asymmetric in bandwidth and/or RTT, which we summarise here:

- When the RTTs on the two paths are mismatched, MPTCP's default RTT scheduler may delay the transmission of a critical resource by assigning some of its packets to the path with the longer RTT.
- Linked Increase is too conservative in the case where bandwidths are mismatched, and may delay congestion window growth on the faster subflow after a loss has occurred.
- MPTCP experiences higher tail loss overall since it stripes traffic across two paths. Moreover, moving any traffic to a path that experiences higher tail loss will cause an increase in PLT in the case where bandwidths are mismatched. This problem worsens when the RTT of the subflow in question is shorter than the minimum retransmission timeout.
- The choice of interface to initiate the connection on is crucial. When MPTCP uses a path with a longer RTT to set up a new connection, web page completion times suffer greatly.

After fully understanding these limitations, we applied several strategies to address them. First, we designed and evaluated the Stream Scheduler, which uses knowledge of resources and their priorities to allocate packets to subflows so that it can minimise the page's overall completion time. The Stream Scheduler prevents packets at the end of critical resources from being scheduled on the slower path. If a subflow has space in its congestion window but cannot send the current critical resource because doing so will delay it, the scheduler looks ahead in the transmit buffer to find a lower-priority resource to send. When we apply the Stream Scheduler for RTT-mismatched topologies, MPTCP can move from reducing PLT for only 32–38% of the web sites examined, to reducing PLT for about 78% of the web sites.

Additionally, we allowed out-of-order delivery of packets to the application layer, disabled Linked Increase and implemented tail loss probes to further improve PLT when using MPTCP. Since our modifications require application-layer knowledge, the simplest method for achieving them would involve redesigning the multipath transport such that it becomes an

application-layer protocol. QUIC is a good substrate to build upon and augment with our proposed design, since it offers transport-layer functionality in the application layer. Nevertheless, with these mechanisms in place, we have shown that it is advantageous to turn MPTCP on by default, since it will improve PLT in many cases, and in the remaining ones perform as well as a single path in the worst case scenario.

7.1 Future Research

Since our investigation of multipath web browsing was primarily motivated by taking advantage of multiple interfaces on mobile devices, the most immediate next step for extending this work is modelling WiFi and LTE in the PCP emulator. After we have presented the basic scenarios for bandwidth and RTT in this work, future work should investigate scenarios where bandwidth varies over time, and where there may be 3G or LTE outages. Taking the investigation further, future research can evaluate the mechanisms for improving MPTCP with real mobile devices communicating with local servers modified to support a multipath transport. With the recent development and release of MPQUIC [83], changes can be applied to implement the Stream Scheduler and our other improvements on top of MPQUIC and use it as the substrate for transport in real world experiments.

In various points in this work, we have indicated that TCP (New Reno) exhibits suboptimal loss recovery behaviour, and recommended using BBR as a possible alternative. A full study of single-path web browsing with BBR as transport is a logical future direction to explore. Going further, developing a load-balancing multipath BBR would be a worthwhile endeavour, not only for reducing web page loads, but for general network traffic as well.

One of the key components of our work is the generation of representative web page dependency graphs. The browser uses these to assign dependency-based priorities to the resources. We've seen that MIME-type prioritisation does reasonably well, but in the end it falls short of achieving the best possible gains, and we argued that dependency-based prioritisation was better. Nonetheless, we made no indication as to how the browser may arrive at omniscient dependency graphs, especially since computing them online would require downloading and constructing the full web page first. Future work can investigate several strategies for dependency graph construction:

- The first proposed solution would be to construct the dependency graph at a proxy located close to the origin server, and combine multiple accesses to the same web page in a manner similar to Klotski [101].[1] The proxy could compute an invariant dependency graph with placeholders for URLs that are different across multiple samples. It could then convey these priorities to the browser when future clients request the same web page. Without browser support, the system could use a mechanism similar to Polaris [95] and push a JavaScript scheduler to the client that will assign the correct resource priorities to

[1] The proxy could use Scout here as well [95], but Scout attempts to eliminate sources of non-determinism from web page loads, and this may not be a desired side effect.

the HTTP/2 requests. Note that the scheduler would not prioritise the requests themselves, but rather set the correct dependency-based priorities in the HTTP/2 headers. On the other hand, a solution that is more in line with how content publishers typically design web pages would be to add browser and HTML support, such that each resource can specify its priority directly within its HTML tag.

- Instead of generating dependency graphs at the server, the browser may generate them locally for the first access to the web page, and use the cached version for future accesses. This scheme is beneficial because it will generate a dependency graph tailored specifically to the client given its local state. Of course, the web page structure will change as time progresses, but having a cached copy of a previous dependency graph may offer useful insights about the priorities of objects in future accesses to the same web page. Unfortunately, both these techniques require a learning period before viable dependency graphs become available. As a stop gap, the browser could use MIME-type prioritisation then transition to dependency-based priorities at a later point.

- Finally, it may be easier to construct smaller dependency sub-graphs at either the server or client. A browser can co-opt the preloader and use it to build small parts of the dependency graph as it goes along. The caveat is that the preloader will not evaluate JavaScript, and therefore will miss any dependencies that appear as a result of executing JavaScript code. On the other hand, web page developers may already know whether certain child resources should be downloaded as a result of downloading a particular parent resource, and incorporate this dependency at the server end.

Of the three techniques we have proposed, none of them is ideal, or will provide the browser with a fully omniscient dependency graph at a particular instant. It is worthy investigating whether the complexity of the proposed schemes is justified by the possible improvements in PLT they can achieve.

In fact, given that continuing to add bandwidth and paths will only achieve diminishing returns as web downloads become more application limited, server push emerges as the likely best option for continuing to reduce PLT. Server push will flatten out dependencies, and eliminate extra RTTs required for sending requests from the client. Unfortunately, server push trades off shorter page loads with larger volumes of traffic on on the network, and may defeat caching.

Further investigation of server push is necessary. One can envision a system where the client notifies the server of some subset of its cached resources on the first request, encoding resource fingerprints using a mechanism similar to that employed by Silo [45]. The server, or reverse proxy, could then push only the uncached resources to the client, applying dependency-based prioritisation as it sees fit. To expand on server push even further, future work could investigate how this scheme behaves over multiple paths as well.

In the end, web page downloads are made up of diverse technologies and protocols that interact, sometimes in unexpected ways. We explored how the use of multiple paths may reduce page completion time under different regions of operation, and examined the phenomena that had emerged from the interactions of the application and transport layers in detail. What remains is to design deployment-ready techniques that use the right mechanisms for the specified regions of operation, and do so seamlessly. We have shown which techniques are the most promising and in which regimes, and now others can hopefully put these observations into practice in deployment.

Appendices

Appendix A

DOM Event Dependencies

The following tables list the most common types of JavaScript event activities we have encountered in popular web pages, and the parents of these events. By determining the event's parents, we can add them correctly to the web page's dependency graph. The tables are grouped by the event's DOM object target.

Target : Document	
Name	
DOMContentLoaded	**Description** Fires when a frame's HTML document has been completely loaded and parsed, without waiting for other resources and subframes to finish loading. **Parents** The event's predecessor in the frame completion events chain (Figure 3.9).
readystatechange	**Description** Fires when the Document changes its `readyState` attribute. Possible values are "loading", "interactive", or "complete". See Section 3.5.2 for more details. **Parents** The event's predecessor in the frame completion events chain, Figure 3.9.

Table A.1: Events which target a DOM Document, and their dependency information.

Target : Element	
Name	
load	**Description**
	Fires when the resource referenced by one the following tags completes downloading: `<img>`, `<input type="image">`, `<link>`, `<script>` or `<style>`.
	Parents
	Resource downloaded by tag.
load	**Description**
	The event targets the `<iframe>` element, and represents a frame load event (see Section 3.5.2 for details). Frame load events may fire twice: first when the frame's HTML file completes downloading, and second when all the iframe's resources and subframes have completed. We distinguish between these two cases accordingly in the dependency graph.
	Parents
	Depending on whether the event is fired for the first second time, the parent is either the Resource corresponding to the `iframe`'s HTML file. Otherwise, the event's parents are all the resources and subframes recursively requested by the iframe in addition to the event's predecessor in the frame completion events chain (Figure 3.9).
error	**Description**
	Fires if an error occurs while downloading the resource referenced by the following tags: `<img>`, `<input type="image">`, `<object>`, `<script>` or `<style>`.
	Parents
	Resource referenced by element, triggering the error.
*	**Description**
	Any event which targets a particular DOM Element, but is fired as a direct result of executing a JavaScript. For example, a script might add an extra Element into the DOM by calling `element.appendElement()` which will trigger DOMNodeInserted event. Other examples include the focus event, which can fire when a developer programmatically draws focus to an element by calling `element.focus()` from JavaScript.
	Parents
	The Computation activity (*e.g.*Script Eval) corresponding to the JavaScript that triggered the event.

Table A.2: Events which target a DOM Element, and their dependency information.

Target : XMLHTTPRequest	
Name	
readystatechange	**Description** Fires when the `readyState` attribute of an XMLHTTPRequest changes. An XMLHTTPRequest is a browser scripting language API that allows a client to incrementally request resources from a server without having to perform a full web page refresh. The possible XMLHTTPRequest `readyState` values are: • 0: Request not initialized. • 1: Server connection established. • 2: Request received. • 3: Processing request. • 4: Request finished and response is ready. After creating an XMLHTTPRequest, developers can use the methods `open()` and `send()` to respectively initialise and then send the request. **Parents** The event handler which fires when XMLHTTPRequest `readyState == 1` depends on evaluating the Computation calling the `send()` method. Subsequent events with an increasing state number depend on downloading the resource referenced by the `send` method. Each event also depends on calling the previous readystatechange event handler in order of state.
Target : MessagePort	
Name	
message	**Description** A MessagePort is an interface in the Channel Messaging API, which allows two separate scripts running in different browsing contexts attached to the same document (*e.g.*, two iframes, or the main document and an iframe) to communicate directly, passing messages between one another through two-way channels (or pipes) with a port at each end. The event fires when a script calls the `port.postMessage()` method, and is captured by the target's event handler. **Parents** The Computation activity corresponding to the JavaScript that called the `postMessage()` function.

Table A.3: Events with XMLHTTPRequest and MessagePort targets, and their dependency information.

Target : Window	
Name	
DOMContentLoaded	**Description** Corresponds to the Document DOMContentLoaded event after it bubbles up to the Window and triggers its event handler. **Parents** The event's predecessor in the frame completion events chain (Figure 3.9).
load	**Description** The Window load event is declared by setting an HTML frame's `<body onload=loadFunction()>`. Fires when the browser has completed downloading and parsing the entire HTML document, in addition to all the resources requested by the document. It also recursively depends on the completion of all the document's subframes and their resources as well. See Section 3.5.2 for more details. **Parents** All the frame's downloaded resources, subframes and their resources, in addition to the event's predecessor in the frame completion events chain (Figure 3.9).
message	**Description** Fires when a JavaScript calls the `window.postMessage()` function on a recipient Window. Normally, scripts on different pages can only communicate if they have the same origin, *i.e.*the same protocol (https), port number and domain. `postMessage` allows safe cross-origin communication between two scripts on different windows. The recipient Window declares an event handler which deals with the message received. **Parents** The Computation activity calling the `window.postMessage()` function.
error	**Description** A runtime JavaScript error will trigger the global `window.onerror` event handler. **Parents** The Script Eval activity corresponding to the faulty JavaScript.
error	**Description** An error event that has been captured from an element. **Parents** Either the Script Eval or the Resource the original element references.

Table A.4: Events that target a DOM Window, and their dependency information.

Appendix B

Negative Acknowledgements

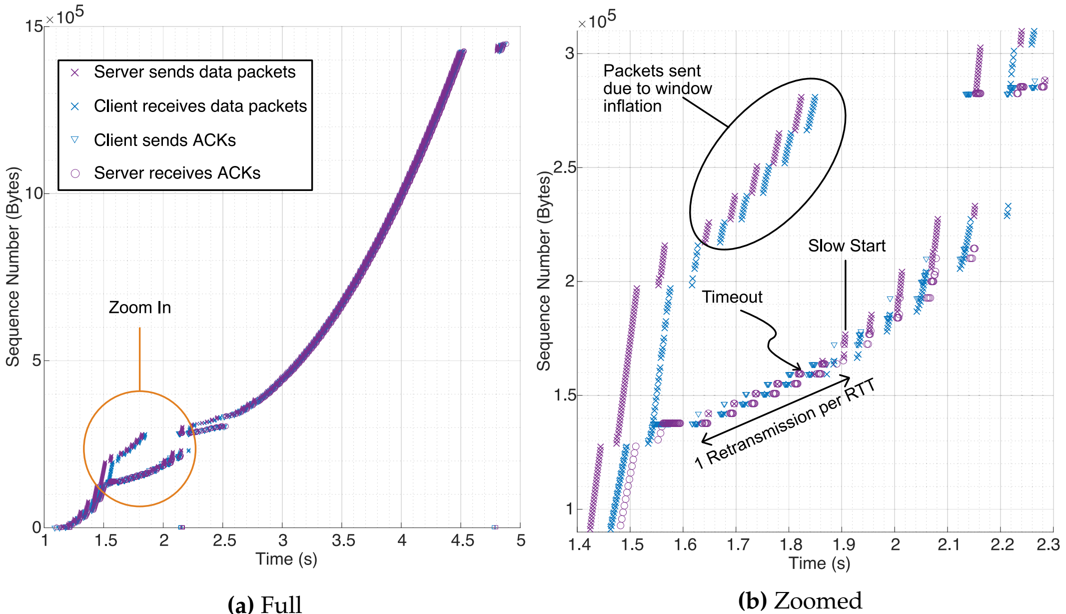
TCP New Reno, combined with TCP's basic congestion control behaves as follows [132, 144]: when a TCP sender receives three duplicate ACKs, it infers a loss, halves its congestion window cwnd, and enters the fast recovery state. It also records the highest transmitted sequence number in the RecoveryPoint variable.

In the fast recovery state, the sender immediately retransmits the lost packet marked by the duplicate ACKs. The sender continues to receive duplicate ACKs for any packets that have been transmitted in the same window with higher sequence numbers than the lost packet, until it finally receives a cumulative ACK for all the data up until RecoveryPoint. Because a duplicate ACK is an implicit signal that a data packet has left the network, the sender artificially inflates its cwnd in response to any incoming duplicate ACK. When the inflated cwnd becomes larger that the unacknowledged bytes in flight, then the sender can transmit the next untransmitted data packets from the front of the transmit buffer.

New Reno tackles the case where there are multiple losses in the same window, for example at the end of slow-start. It allows the sender to deflate the window by the amount of data acknowledged if it receives a "partial acknowledgement", *i.e.* an ACK that covers new data, but not all the data up to RecoveryPoint. The partial ACK marks the next gap in the sequence space. As such, a New Reno sender will retransmit the packet marked by this ACK upon its reception. As an optimisation, ns-3 does not rearm the retransmission timeout timer after any duplicate ACK is received during fast recovery, which has been shown to perform better than allowing the recovery period to continue indefinitely. As a result, if a retransmission timeout elapses from the moment the first loss is detected, the sender will timeout, and begin slow-start from the missing packet, transmitting all the consecutive segments in order.

Figure B.1 shows an example of TCP New Reno in action. In Figure B.1(a), we plot the time/sequence for a single connection to a particular domain of the web site imgur.com downloaded using PCP at 10Mbps and an RTT of 40ms. Other connections for this web site (not shown) compete with this connection at the bottleneck link and cause losses. We can see that the connection spends nearly a full second recovering from a burst of losses that happen within the marked circle. Zooming in, we obtain Figure B.1(b), which illustrates a fundamental property of New Reno's loss recovery mechanism: the sender can only retransmit one lost packet

per RTT. Despite the fact that sender tries to keep the pipe busy by sending data packets from the top end of the congestion window using artificial window inflation, retransmission takes too long. The sender cannot recover all the lost packets within a retransmission timeout, and accordingly times out and begins slow start again. If there are too many losses in the window, or there are not enough bytes in flight to inflate cwnd past the RecoveryPoint, then the sender cannot transmit any packets past the right hand side of the congestion window during fast retransmit. It will under-utilise the link when it only sends one retransmission per RTT and nothing else.



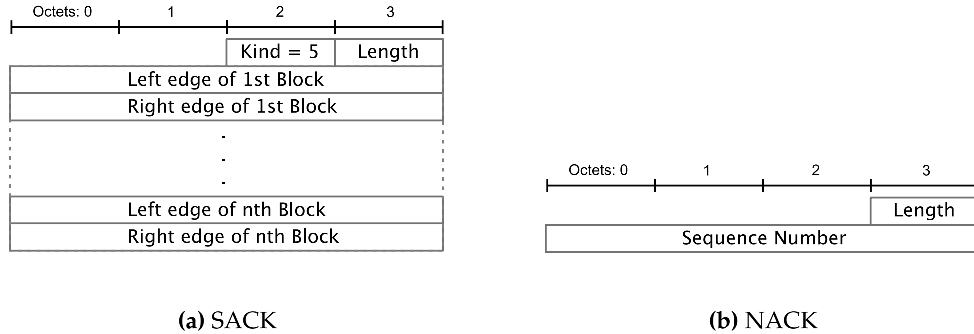
(a) Full

(b) Zoomed

Figure B.1: Time/sequence plot for a connection to one of the domains of imgur.com, downloaded at 10Mbps and 40ms RTT using TCP New Reno.

Modern operating systems use selective acknowledgements (SACK for short) to address TCP New Reno's deficiencies. SACK is a TCP option which allows a receiver to acknowledge the exact sequence ranges which it has received [64]. A SACK option, shown in Figure B.2(a), marks the left and right side of contiguous blocks of sequence numbers delivered to the receiver. With a maximum option size of 40 bytes, the TCP SACK option can contain up to four ACKed blocks, or three blocks if the timestamp option is used. We added an ACK signalling mechanism to ns-3, which behaves in a similar fashion to the SACK option, allowing the sender to determine which sequence numbers have been lost without resorting to the step behaviour of New Reno.

Before we delve into the into the particulars of the scheme we implemented, let us first give an overview of a conservative TCP loss recovery mechanism using SACK [156]. A TCP sender maintains a scoreboard of received SACK information, and updates it using incoming SACK blocks within cumulative ACKs. Using the scoreboard, the sender determines a sequence number to be lost if three non-contiguous SACKed sequences have arrived above the sequence number in question, or if more than two segments greater than the sequence number



(a) SACK **(b)** NACK

Figure B.2: Option format for SACK and NACK TCP options.

have been SACKed. With SACK blocks, a more accurate estimate of the bytes in flight can be computed: they are the number of bytes that have not been SACKed minus those determined to be lost, plus any retransmitted bytes. On a retransmission timeout, which might indicate that the receiver has reneged, the sender discards the SACK scoreboard, and slow-starts again.

With this information in hand, the SACK-enabled TCP sender performs the following steps after receiving an ACK during fast recovery:

1. If the ACK has a greater sequence number than RecoveryPoint, then the sender exits fast recovery, and clears the SACK scoreboard less than RecoveryPoint.

2. If the ACK is less than RecoveryPoint, the sender updates its bytes in flight estimate and the scoreboard. While cwnd is greater than the bytes in flight, it transmits a packet as follows:

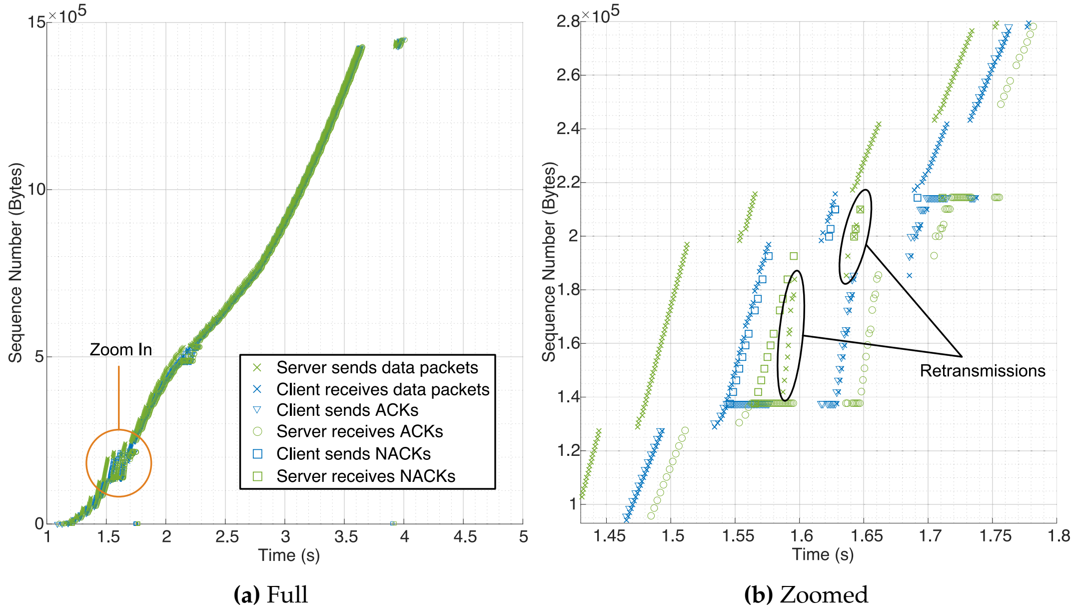
   (a) It retransmits the next packet determined to be lost according to the criteria outlined above.

   (b) If all the lost packets have been transmitted, it sends the next unsent data in transmit buffer.

   (c) If neither of the above operations is viable, it retransmits as rescue packets any sequences it suspects have been lost, but do not meet the stringent loss criteria.

Our simulations do no suffer from packet reordering or ACK loss. As a result, we implemented a lightweight version of SACK in ns-3 which behaves similarly with respect to loss recovery. In contrast to sending a SACK block with received sequences explicitly marked, the receiver notifies the sender using negative acknowledgements (NACKs) exactly which sequences are missing. Using the same logic as SACK to determine which sequences have been lost, the receiver checks its receive buffer for any gaps in the sequence space. If there are more than two received segments after the gap, then the receiver appends a NACK option to a duplicate ACK if it hasn't transmitted it already. Figure B.2(b) shows the structure of NACK option — it contains the sequence number of the lost packet and its length.

On the other end, the sender stores a list of NACKed sequences and their length, keeps track of the highest NACKed sequence which has been retransmitted, and clears any NACKed sequences less than the cumulative ACK. Because the sender doesn't have SACK blocks which explicitly mark which segments have been received, it cannot decrease its estimate of the bytes in flight as duplicate ACKs arrive, which would have typically allowed it to transmit some new unsent packets. As a result, we choose to keep the mechanism of inflating the cwnd for each duplicate ACK received, because this implicitly marks when a packet leaves the network.



(a) Full

(b) Zoomed

Figure B.3: Time/sequence plot for a connection to one of the domains of imgur.com, downloaded at 10Mbps and 40ms RTT using TCP New Reno with NACKs.

A sender can, in theory, retransmit a NACKed sequence as soon as it receives it; it is "safe" to do so because the lost packet has technically left the network. Unfortunately, this scheme is not conservative because it does not halve the transmission rate after a loss, and hence outperforms SACK. We only send a retransmission when there is space in the cwnd. We use an algorithm similar to SACK: while the inflated cwnd is greater than the bytes in flight, the sender first attempts to retransmit any NACKed sequences which it has not retransmitted yet. Otherwise, if it has retransmitted everything, it will send new data from the transmit buffer. In event of a timeout, the sender clears its NACKed sequences list. It also modifies its bytes in flight definition: they are the unacknowledged data minus NACKed sequences plus retransmitted bytes.

The NACK mechanism suffers from a particular caveat. Sometimes, the receiver doesn't have more than two segments after a gap to indicate that a loss has occurred, *e.g.* the very last packet in window is lost. For this case, the sender will fall back to New Reno behaviour of using partial ACKs as a signal for retransmitting the missing sequences. This problem is not unique to the NACK mechanism — SACK will suffer in a similar manner.

Figure B.3 shows the behaviour of imgur.com downloaded with NACKs enabled. As we

can see, the download recovers from losses much faster (within 200ms), and doesn't need to time out.

Appendix C

Congestion Window Validation

The basic idea behind congestion window validation is to decay the congestion window proportional to the duration of the idle period, whilst using the slow start threshold ssThresh to store the previous value of cwnd. In addition, the TCP sender should not increase the congestion window when it is application limited. Whenever the sender starts up again after receiving new data, it will begin with a smaller cwnd and perform slow start up to ssThresh, the value of the previous cwnd, thereby tentatively probing the network for available capacity.

In PCP, we applied congestion window validation according to RFC 2861 [133], and as shown in Algorithm 10. Any time a sender attempts to send a data packet, it must first validate the congestion window if it has been idle for some time (Line 2). Of course, the sender can transmit a packet if it has data to send, and if its effective congestion window is larger than the number bytes in flight by at least one Sender Maximum Segment Size (SMSS). After transmitting any packets, the sender then checks whether it is application limited on Line 8, and performs any resulting actions. Note that the sender records the time the last packet was sent in the T_{last} variable.

Algorithm 10 Send pending data.

1: **function** SENDPENDINGDATA( )
2: VALIDATEWINDOW( )
3: $window \leftarrow$ MIN($cwnd$, $rwnd$)
4: $pipe \leftarrow$ BYTESINFLIGHT( )
5: **while** ($window - pipe \geq SMSS$) **and** HAVEDATA( ) **do**
6: SENDNEXTPACKET( )
7: $T_{last} \leftarrow$ NOW( )
8: CHECKAPPLICATIONLIMITED( )

To perform window validation, the basic algorithm defines three new state variables. T_{last} denotes the last time the sender has transmitted a packet. T_{prev} indicates the last time a packet was sent while the congestion window was full, *i.e.* the last time the flow was network limited. T_{prev} also records the last time the window was halved due to validation. Finally, for an application limited flow, W_{used} marks the maximum window size actually used since the sender was last network-limited.

Algorithm 11 shows how we decay the congestion window during idle periods using the VALIDATEWINDOW() function. If an RTO or more has elapsed since the sender last transmitted a packet, then it sets ssThresh to $\frac{3}{4}$ of cwnd, a conservative estimate of the previous window. The sender subsequently halves cwnd for every RTO elapsed since T_{last}, the beginning of the idle period. It limits the window reduction so that the window cannot become smaller than $icwnd$, the initial congestion window.

Algorithm 11 Validate window.

1: **function** VALIDATEWINDOW( )
2: $duration \leftarrow$ NOW( ) $-T_{last}$
3: **if** ($duration \geq RTO$) **and** ($TCPState == OPEN$) **then**
4: $ssThresh \leftarrow$ MAX($ssThresh$, $\frac{3}{4} \cdot cwnd$)
5: $window \leftarrow$ MIN($cwnd$, $rwnd$)
6: **for** $i = 1$ **to** $duration/RTO$ **do**
7: $cwnd \leftarrow$ MAX($icwnd$, $\frac{1}{2} \cdot window$)
8: $T_{prev} \leftarrow$ NOW( )
9: $W_{used} \leftarrow 0$

Algorithm 12 Check application limited

1: **function** CHECKAPPLICATIONLIMITED( )
2: pipe $\leftarrow$ BYTESINFLIGHT( )
3: **if** $cwnd - pipe < SMSS$ **then**
4: $T_{prev} \leftarrow$ NOW( )
5: $W_{used} \leftarrow 0$
6: **else if** No more data available to send **then**
7: W_{used} = MAX(W_{used}, pipe)
8: **if** ($T_{prev} \neq 0$) **and** (NOW( ) - $T_{prev} \geq RTO$) **and** ($TCPState == OPEN$) **then**
9: ssThresh $\leftarrow$ MAX(ssThresh, $\frac{3}{4} \cdot cwnd$)
10: window $\leftarrow$ MIN(cwnd, rwnd)
11: cwnd $\leftarrow$ MAX(icwnd, $\frac{1}{2} \cdot (window + W_{used})$)
12: $T_{prev} \leftarrow$ NOW( )
13: $W_{used} \leftarrow 0$

Algorithm 12 shows the CHECKAPPLICATIONLIMITED() function, which determines whether the flow is application limited after it sends a packet. If the flow is in fact network limited instead (Line 3), the function resets the variables T_{prev} and W_{used} accordingly. But if the flow still has space in its congestion window, yet has no more data to send, then it is application limited. In that case, the sender checks whether the number of bytes in flight are larger than W_{used} and if so sets W_{used} to that value. If T_{prev} is not zero and the elapsed time since T_{prev} is larger than an RTO, then VALIDATEWINDOW() has not just reduced the congestion window. The TCP flow has instead been application limited for at least an entire RTO interval. In this case, the function sets ssThresh to be $\frac{3}{4}$ of the current window. It also sets the current congestion window to be the average between it previous value, and the actual window last used when the flow was network limited.

After the window reduction is complete, the function resets the values of T_{prev} and W_{used}

so no further window reduction can take place until at least another RTO period has elapsed. Finally, we must note that we apply any window reduction during VALIDATEWINDOW() and CHECKAPPLICATIONLIMITED() only if the TCP socket is in the OPEN state, and is not experiencing any losses.

Bibliography

[1] CERN. The first web site. http://info.cern.ch.

[2] Internet live stats. http://www.internetlivestats.com/, 2017.

[3] Dependency graphs for alexa top 500 websites. https://github.com/kusoof/alexa_traces/tree/master/dependency_graphs/alexa_deps_24_3_2017, 2017.

[4] MPTCP implementation for ns-3. https://github.com/kusoof/ns-3-dev-git/tree/mptcp, 2016.

[5] Cisco. Cisco visual networking index: Forecast and methodology, 2016-2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html, September 2017.

[6] Cisco. The zettabyte era: Trends and analysis. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html, June 2017.

[7] Lucian Popa, Ali Ghodsi, and Ion Stoica. HTTP as the narrow waist of the future internet. In *Proceedings of 9th ACM Workshop on Hot Topics in Networks (HotNets '10)*, October 2010.

[8] Ofcom. The communications market report. https://www.ofcom.org.uk/__data/assets/pdf_file/0023/26393/uk_internet.pdf, 2016.

[9] Ramakrishnan Rajamony and Elmootazbellah (Mootaz) Elnozahy. Measuring client-perceived response time on the www. In *Proceedings of the 3rd* USENIX *Symposium on Internet Technologies and Systems*, USITS, March 2001.

[10] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. Demystifying page load performance with WProf. In *Proceedings of the 11th USENIX symposium on Networked Systems Design and Implementation (*NSDI '13*)*, 2013.

[11] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert G Greenberg, and Yi-Min Wang. Webprophet: Automating performance prediction for web services. In *Proceedings of the 11th USENIX symposium on Networked Systems Design and Implementation (*NSDI '10*)*, volume 10, pages 143–158, 2010.

[12] Matteo Varvello, Jeremy Blackburn, David Naylor, and Kostantina Papagiannaki. Eyeorg: A platform for crowdsourcing web quality of experience measurements. In *Proceedings of the 12th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*, pages 399 – 412, 2016.

[13] Nina Bhatti, Anna Bouch, and Allan Kuchinsky. Integrating user-perceived quality into web server design. *Computer Networks*, 33(1):1–16, 2000.

[14] Dennis F. Galletta, Raymond Henry, Scott McCoy, and Peter Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 5(1):1, 2004.

[15] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 297–304. ACM, 2000.

[16] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly Media, 2013.

[17] Jakob Nielsen. *Usability Engineering. Response times: the three important limits*. Academic Press Inc., 1993.

[18] Using site speed in web search ranking. https://webmasters.googleblog.com/2010/04/using-site-speed-in-web-search-ranking.html, 2010.

[19] Jake Brutlag. Speed matters for google web search. http://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009.

[20] Greg Linden. Make data useful. https://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf, 2006.

[21] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Joanna Kulik, Jim Roskind, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. The QUIC transport protocol: Design and internet-scale deployment. In *SIGCOMM*. ACM, 2017.

[22] Roy Fielding, J. Gettys, Jeff Mogul, H. Frystyk, L. Masinter, P. Leach, and Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616, June 1999.

[23] Ilya Grigorik. HTTP/2 is here, let's optimize! https://docs.google.com/presentation/d/1r7QXGYOLCh4fcUq0jDdDwKJWNqWK1o4xMtYpKZCJYjM/, 2015.

[24] Matteo Varvello, Kyle Schomp, David Naylor, Jeremy Blackburn, Alessandro Finamore, and Konstantina Papagiannaki. Is the web HTTP/2 yet? In *Passive and Active Measurements Conference (PAM)*, 2016.

[25] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540, May 2015. https://tools.ietf.org/html/rfc7540.

[26] Mike Belshe and Roberto Peon. SPDY protocol: Draft 3.2. http://dev.chromium.org/spdy/spdy-protocol/spdy-protocol-draft3-2.

[27] QUIC, a multiplexed stream transport over udp. http://www.chromium.org/quic.

[28] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2, Internet Draft*. IETF Internet Draft, draft-ietf-quic-transport, December 2017. https://tools.ietf.org/html/draft-ietf-quic-protocol-08.

[29] J. Iyengar and I. Swett. *QUIC Loss Recovery And Congestion Control, Internet Draft*. IETF Internet Draft, draft-ietf-quic-recovery, December 2017. https://tools.ietf.org/html/draft-ietf-quic-recovery-08.

[30] Statista. Share of global mobile website traffic 2015-2017. https://www.statista.com/statistics/277125/share-of-website-traffic-coming-from-mobile-devices/, 2017.

[31] Statista. Percentage of all global web pages served to mobile phones from 2009 to 2017. https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/, 2017.

[32] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, implementation and evaluation of congestion control for multipath TCP. In *Proceedings of USENIX symposium on Networked Systems Design and Implementation (*NSDI '13*)*, volume 11, 2011.

[33] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 6824, 2013. https://tools.ietf.org/html/rfc6824.

[34] Iljitsch van Beijnum. Multipath TCP lets Siri seamlessly switch between Wi-Fi and 3G/LTE. https://arstechnica.com/gadgets/2013/09/multipath-tcp-lets-siri-seamlessly-switch-between-wi-fi-and-3glte/, 2013.

[35] Ns-3 network simulator. http://www.nsnam.org/, 2015.

[36] Alexa top 500 global sites. http://www.alexa.com/topsites.

[37] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. The internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. ACM, 2014.

[38] Google. DNS pre-fetching. http://www.chromium.org/developers/design-documents/dns-prefetching.

[39] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246, August 2008.

[40] B. Briscoe, A. Brunstrom, A. Petlund, D. Hayes, D. Ros, I.-J. Tsang, Gjessing S., G. Fairhurst, C. Griwodz, and M. Welzl. Reducing internet latency: A survey of techniques and their merits. *IEEE Communications Surveys and Tutorials*, 18(3):2149–2196, 2016.

[41] Tim Berners-Lee, Roy Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945, May 1996.

[42] Jeffrey C. Mogul. The case for persistent-connection HTTP. In *Proceedings of ACM SIGCOMM*, pages 299–313. ACM, October 1995.

[43] Google. Pagespeed insights. https://developers.google.com/speed/docs/insights/rules.

[44] Google. Pagespeed module. https://developers.google.com/speed/pagespeed/module.

[45] James Mickens. Silo: exploiting JavaScript and DOM storage for faster page loads. In *Proceedings of 2010 USENIX Conference on Web Application Development, (WebApps '10)*, 2010.

[46] Sivasankar Radhakrishnan, Cheng Yuchung, Jerry Chu, Arvind Jain, and Barath Raghavan. TCP fast open. In *Proceedings of the Seventh Conference on Emerging Networking Experiments and Technologies (*CoNEXT '11*)*, page 21. ACM, 2011.

[47] Wenxuan Zhou, Qingxi Li, Matthew Caesar, and P. Godfrey. ASAP: A low-latency transport layer. In *Proceedings of the Seventh Conference on emerging Networking Experiments and Technologies (*CoNEXT *'11)*, page 20. ACM, 2011.

[48] Randall Stewart. *Stream control transmission protocol*. RFC 4960, September 2007.

[49] Preethi Natarajan, P. Amer, J. Leighton, and F. Baker. Using SCTP as a transport layer protocol for HTTP. *Work in progress as an internetdraft, draft-natarajan-http-over-sctp*, 2009.

[50] SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper.

[51] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. How speedy is SPDY? In *Proceedings of the 11th USENIX symposium on Networked Systems Design and Implementation (*NSDI '14*)*, April 2014.

[52] Yehia Elkhatib, Gareth Tyson, and Michael Welzl. Can SPDY really make the web faster? In *Networking Conference, 2014 IFIP*, pages 1–9. IEEE, 2014.

[53] Hugues de Saxcé, Iuniana Oprescu, and Yiping Chen. Is HTTP/2 really faster than HTTP/1.1? In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 293–299. IEEE, April 2015.

[54] Jitu Padhye and Henrik Frystyk Nielsen. A comparison of SPDY and HTTP performance. Technical Report MSR-TR-2012-102, Microsoft, July 2012.

[55] Kyriakos Zarifis, Mark Holland, Manish Jain, Ethan Katz-Bassett, and Ramesh Govindan. Modeling HTTP/2 speed from HTTP/1 traces. In *International Conference on Passive and Active Network Measurement (PAM)*, pages 233–247. Springer, 2016.

[56] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and KK Ramakrishnan. Towards a SPDY'ier mobile web? In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, CONEXT, pages 303–314. ACM, 2013.

[57] Matteo Varvello, Kyle Schomp, David Naylor, Jeremy Blackburn, Alessandro Finamore, and Kostantina Papagiannaki. To HTTP/2, or not to HTTP/2, that is the question. *arXiv preprint arXiv:1507.06562*, 2015.

[58] M. Bishop. *Hypertext Transport Protocol HTTP over QUIC*. IETF Internet Draft, draft-ietf-quic-http, December 2017. https://tools.ietf.org/html/draft-ietf-quic-http-08.

[59] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. How secure and quick is QUIC? Provable security and performance analyses. In *IEEE Symposium on Security and Privacy (SP)*, pages 214 – 231, May 2015.

[60] Adam Langley and Wan-Teh Chang. QUIC Crypto. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit.

[61] M. Thompson and S. Turner. *Using Transport Layer Security (TLS) to Secure QUIC*. IETF Internet Draft, draft-ietf-quic-tls, 2017.

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

[63] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. *ACM SIGCOMM Computer Communication Review*, 17(5):2–7, 1987.

[64] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgment Options*. RFC 2018, October 1996.

[65] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. HTTP over UDP: an experimental investigation of QUIC. In *Proceedings of ACM Symposium on Applied Computing, SAC '15*, 2015.

[66] Prasenjeet Biswal and Omprakash Gnawali. Does QUIC make the web faster? In *Global Communications Conference (GLOBECOM), 2016 IEEE*, pages 1–6. IEEE, 2016.

[67] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. How quick is QUIC? In *IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2016.

[68] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14(5):50, 2016.

[69] Jim Gettys and Kathleen Nichols. Bufferbloat: Dark buffers in the Internet. *ACM Queue*, 9(11):40, 2011.

[70] Neal Cardwell, Yuchung Cheng, Soheil Hassas Yeganeh, and Van Jacobson. *BBR Congestion Control*. IETF Internet Draft, July 2017. https://tools.ietf.org/html/draft-cardwell-iccrg-bbr-congestion-control.

[71] Yuchung Cheng, Neal Cardwell, Soheil Hassas Yeganeh, and Van Jacobson. *Delivery Rate Estimation*. IETF Internet Draft, July 2017. https://tools.ietf.org/html/draft-cheng-iccrg-delivery-rate-estimation.

[72] Janardhan R Iyengar, Paul D. Amer, and Randall Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Transactions on Networking*, 14(5):951–964, 2006.

[73] C. Raiciu, M. Handley, and D. Wischik. *Coupled Congestion Control for Multipath Transport Protocols*. RFC 6356, 2011. https://tools.ietf.org/html/rfc6356.

[74] Costin Raiciu, Sebastien Barre, Christopher Pluntke, Adam Greenhalgh, Damon Wischik, and Mark Handley. Improving datacenter performance and robustness with multipath TCP. *ACM SIGCOMM Computer Communication Review*, 41(4):266–277, 2011.

[75] Costin Raiciu, Christopher Pluntke, Sebastien Barre, Adam Greenhalgh, Damon Wischik, and Mark Handley. Data center networking with multipath TCP. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets '10)*, page 10. ACM, 2010.

[76] Costin Raiciu, Christoph Paasch, Sebastien Barre, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. How hard can it be? Designing and implementing a deployable multipath TCP. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI '12)*, pages 29–29. USENIX Association, 2012.

[77] Christoph Paasch, Gregory Detal, Fabien Duchene, Costin Raiciu, and Olivier Bonaventure. Exploring mobile/WiFi handover with multipath TCP. In *Proceedings of the 2012 ACM SIGCOMM workshop on Cellular networks: operations, challenges, and future design*, pages 31–36. ACM, 2012.

[78] Yung-Chih Chen, Yeon-sup Lim, Richard J. Gibbens, Erich M. Nahum, Ramin Khalili, and Don Towsley. A measurement-based study of multipath TCP performance over wireless networks. In *Proceedings of the 2013 Internet Measurement Conference (IMC '13)*, pages 455–468. ACM, 2013.

[79] Shuo Deng, Ravi Netravali, Anirudh Sivaraman, and Hari Balakrishnan. Wifi, LTE, or both? Measuring multi-homed wireless internet performance. In *Proceedings of the 2014 Internet Measurement Conference (IMC '14)*, pages 181–194. ACM, 2014.

[80] Ashkan Nikravesh, Yihua Guo, Feng Qian, Z Morley Mao, and Subhabrata Sen. An in-depth understanding of multipath TCP on mobile devices: measurement and system design. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 189–201. ACM, 2016.

[81] Bo Han, Feng Qian, Shuai Hao, and Lusheng Ji. An anatomy of mobile web performance over multipath TCP. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (*CoNEXT '15*)*, page 5. ACM, 2015.

[82] Bo Han, Feng Qian, and Lusheng Ji. When should we surf the mobile web using both WiFi and cellular? In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges*, pages 7–12. ACM, 2016.

[83] Quentin De Coninck and Olivier Bonaventure. Multipath QUIC: Design and evaluation. In *Proceedings of the 13th ACM Emerging networking experiments and technologies (*CoNEXT *'17)*, pages 160–166. ACM, December 2017.

[84] C. Paasch and S. Barre. Multipath TCP - linux kernel implementation. http://www.multipath-tcp.org/.

[85] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. Experimental evaluation of multipath TCP schedulers. In *Proceedings of the 2014 ACM SIGCOMM workshop on Capacity Sharing*, pages 27–32. ACM, 2014.

[86] Nicolas Kuhn, Emmanuel Lochin, Ahlem Mifdaoui, Golam Sarwar, Olivier Mehani, and Roksana Boreli. DAPS: Intelligent delay-aware packet scheduling for multipath transport. In *IEEE International Conference on Communications (ICC)*, pages 1222–1227. IEEE, 2014.

[87] Golam Sarwar, Roksana Boreli, Emmanuel Lochin, Ahlem Mifdaoui, and Guillaume Smith. Mitigating receiver's buffer blocking by delay aware packet scheduling in multipath data transfer. In *27th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 1119–1124. IEEE, 2013.

[88] Fan Yang, Qi Wang, and Paul D. Amer. Out-of-order transmission for in-order arrival scheduling for multipath tcp. In *28th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 749–752. IEEE, 2014.

[89] Simone Ferlin, Özgü Alay, Olivier Mehani, and Roksana Boreli. BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks. In *IFIP Networking Conference (IFIP Networking) and Workshops, 2016*, pages 431–439. IEEE, 2016.

[90] Thomas Dreibholz, Robin Seggelmann, Michael Tüxen, and Erwin Paul Rathgeb. Transmission scheduling optimizations for concurrent multipath transfer. In *Proceedings of the 8th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*, volume 8, 2010.

[91] Johan Eklund, Karl-Johan Grinnemo, and Anna Brunstrom. Efficient scheduling to reduce latency for signaling traffic using CMT-SCTP. In *IEEE 27th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–6. IEEE, September 2016.

[92] Yeon-sup Lim, Erich M. Nahum, Don Towsly, and Richard J. Gibbens. ECF: An MPTCP path scheduler to manage heterogeneous paths. In *Proceedings of the 13th ACM Emerging networking experiments and technologies (*CoNEXT *'17)*, pages 147–159. ACM, December 2017.

[93] HTTP archive. http://httparchive.org/.

[94] Chrome developer tools. https://developers.google.com/web/tools/chrome-devtools/.

[95] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16)*. USENIX Association, 2016.

[96] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. Mahimahi: accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC '15)*, pages 417–429, 2015.

[97] Debopam Bhattacherjee, Muhammad Tirmazi, and Ankit Singla. A cloud-based content gathering network. In *9th* USENIX *Workshop on Hot Topics in Cloud Computing (HotCloud '17)*. USENIX Association, 2017.

[98] Opera mini. http://www.opera.com/mobile/mini.

[99] Amazon silk. https://aws.amazon.com/documentation/silk/.

[100] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. Flywheel:

Google's data compression proxy for the mobile web. In *Proceedings of USENIX symposium on Networked Systems Design and Implementation (*NSDI '15*)*, volume 15, pages 367–380, 2015.

[101] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *Proceedings of USENIX symposium on Networked Systems Design and Implementation (NSDI '15)*, pages 439–453, 2015.

[102] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. Speeding up web page loads with shandian. In *Proceedings of the 13th USENIX symposium on Networked Systems Design and Implementation (*NSDI '16*)*, pages 109–122, 2016.

[103] WebKit, an open source web browser engine. https://webkit.org, 2017.

[104] Javad Nejati and Aruna Balasubramanian. An in-depth study of mobile browser performance. In *Proceedings of the 25th International Conference on World Wide Web*, pages 1305–1315. International World Wide Web Conferences Steering Committee, 2016.

[105] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

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

[107] Apache module mod_http2. https://httpd.apache.org/docs/2.4/mod/mod_http2.html.

[108] dpkt: fast, simple packet creation / parsing, with definitions for the basic TCP/IP protocols. https://pypi.python.org/pypi/dpkt.

[109] Kazuho Oku. HTTP/2 is much faster than SPDY thanks to dependency-based prioritization. blog.kazuhooku.com/2015/04/dependency-based-prioritization-makes.html, April 2015.

[110] H2O the optimized HTTP/1.x, HTTP/2 server. https://h2o.examp1e.net/configure/http2_directives.html#prioritization, 2017.

[111] Yu-Kwong Kwok and Ishfaq Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

[112] Webpagetest documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index.

[113] YUI team. Performance research, part 2: Browser cache usage - exposed! http://yuiblog.com/blog/2007/01/04/performance-research-part-2, 2007.

[114] Marja Hölttä and Daniel Vogelheim. New JavaScript techniques for rapid page loads. http://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html, 2015.

[115] Benjamin Erb. *Concurrent Programming for Scalable Web Architectures*. PhD thesis, Ulm University, 2012.

[116] Apache HTTP server project. http://httpd.apache.org/.

[117] Multitasking server architectures. http://www.fmc-modeling.org/category/projects/apache/amp/4_3Multitasking_server.html, 2004.

[118] Nginx. http://www.nginx.com/.

[119] Andrey Alexeev. *The architecture of open source applications*, volume 2. 2012. http://www.aosabook.org/en/nginx.html.

[120] Ofcom. UK home broadband performance. https://www.ofcom.org.uk/__data/assets/pdf_file/0015/100761/UK-home-broadband-performance,-November-2016-Technical-report.pdf, April 2017.

[121] Keith Winstein, Anirudh Sivaraman, Hari Balakrishnan, et al. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *Proceedings of USENIX symposium on Networked Systems Design and Implementation (*NSDI '13*)*, pages 459–471, 2013.

[122] R. Braden. *Requirements for Internet Hosts — Communication Layers*. RFC 1122, October 1989.

[123] M. Allman, H. Balakrishnan, and S. Floyd. *Enhancing TCP's Loss Recovery Using Limited Transmit*. RFC 3042, January 2001.

[124] D. Borman, B. Braden, V. Jacobson, and R. Scheffenegger. *TCP Extensions for High Performance*. RFC 7323, September 2014.

[125] Sally Floyd and Van Jacobson. On traffic phase effects in packet-switched gateways. *Internetworking: Research and Experience*, 3(3):115–156, 1992.

[126] Lixia Zhang and D Clark. Oscillating behavior of network traffic: A case study simulation. *Internetworking: research and experience*, 1(2):101–112, 1990.

[127] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking (ToN)*, 1(4):397–413, 1993.

[128] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet. *The FlowQueue-CoDel Packet Scheduler and Active Queue Management Algorithm*. IETF Internet Draft, March 2016. https://tools.ietf.org/html/draft-ietf-aqm-fq-codel-06.

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

[130] K. Nichols, V. Jacobson, A. McGregor, and J. Iyengar. *Controlled Delay Active Queue Management*. IETF Internet Draft, September 2017. https://tools.ietf.org/html/draft-ietf-aqm-codel-08.

[131] Random variables in ns-3. https://www.nsnam.org/docs/manual/html/random-variables.html, 2017.

[132] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582, April 2012.

[133] Mark Handley, Jitu Padhye, and Sally Floyd. *TCP Congestion Window Validation*. RFC 2861, 2000.

[134] G. Fairhurst, A. Sathiaseelan, and R. Secchi. *Updating TCP to Support Rate-Limited Traffic*. RFC 7661, 2015.

[135] Elixir — free electrons, embedded linux experts. http://elixir.free-electrons.com/, 2017.

[136] Linux kernel source tree. https://github.com/torvalds/linux, 2017.

[137] EasyList. https://easylist.to/, 2017.

[138] Akamai. Akamai's state of the internet, Q1 2017 report. https://www.akamai.com/fr/fr/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf, 2017.

[139] Douglas J. Leith, Robert N. Shorten, and Gavin McCullagh. Experimental evaluation of cubic-TCP. In *Proceedings of the 6th International Workshop on Protocols for Fast Long-Distance Networks (PFLDnet)*, 2008.

[140] HTTP/2 dashboard. http://isthewebhttp2yet.com/, 2016.

[141] Matthieu Coudron and Stefano Secci. An implementation of multipath TCP in ns-3. *Computer Networks*, 116:1–11, 2017.

[142] Morteza Kheirkhah, Ian Wakeman, and George Parisis. Multipath-TCP in ns-3. *arXiv preprint arXiv:1510.07721*, 2015.

[143] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. *Architectural Guidelines for Multipath TCP Development*. RFC 6182, March 2011. tools.ietf.org/html/rfc6182.

[144] Mark Allman, Vern Paxson, and Ethan Blanton. *TCP congestion control*. RFC 5681, September 2009.

[145] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. IETF Internet Draft, draft-dukkipati-tcpm-tcp-loss-probe, 2013. https://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.

[146] Neal Cardwell, Stefan Savage, and Thomas Anderson. Modeling TCP latency. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1742–1751. IEEE, 2000.

[147] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. MPTCP is not pareto-optimal: performance issues and a possible solution. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT '12)*, pages 1–12. ACM, 2012.

[148] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. Reducing web latency: the virtue of gentle aggression. *ACM SIGCOMM Computer Communication Review*, 43(4):159–170, 2013.

[149] Vern Paxson, Mark Allman, Jerry Chu, and Matt Sargent. *Computing TCP's retransmission timer*. RFC 6298, 2011.

[150] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: Refining TCP congestion control. *ACM SIGCOMM Computer Communication Review*, 26(4):281–291, 1996.

[151] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. *Early retransmit for TCP and stream control transmission protocol (SCTP)*. RFC 5827, 2010.

[152] Kiran Yedugundla, Per Hurtig, and Anna Brunstrom. Probe or wait: Handling tail losses using Multipath TCP. In *16th International IFIP TC6 Networking Conference*, 2017.

[153] Kadangode Ramakrishnan, Sally Floyd, and David Black. *The addition of explicit congestion notification (ECN) to IP*. RFC 3168, September 2001.

[154] Sally Floyd. TCP and explicit congestion notification. *ACM SIGCOMM Computer Communication Review*, 24(5):8–23, 1994.

[155] John W. Byers, Michael Luby, Michael Mitzenmacher, and Ashutosh Rege. A digital fountain approach to reliable distribution of bulk data. *ACM SIGCOMM Computer Communication Review*, 28(4):56–67, 1998.

[156] E. Blanton, M. Allman, L. Wang, I. Jarvinen, M. Kojo, and Y. Nishida. *A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP*. RFC 6675, August 2012.