University of New Hampshire
University of New Hampshire Scholars' Repository

Doctoral Dissertations | Student Scholarship

Spring 2012

Provider-Controlled Bandwidth Management for HTTP-based Video Delivery

Kevin J. Ma
University of New Hampshire, Durham

Follow this and additional works at: https://scholars.unh.edu/dissertation

Recommended Citation
Ma, Kevin J., "Provider-Controlled Bandwidth Management for HTTP-based Video Delivery" (2012). *Doctoral Dissertations*. 650.
https://scholars.unh.edu/dissertation/650

This Dissertation is brought to you for free and open access by the Student Scholarship at University of New Hampshire Scholars' Repository. It has been accepted for inclusion in Doctoral Dissertations by an authorized administrator of University of New Hampshire Scholars' Repository. For more information, please contact nicole.hentz@unh.edu.

Provider-Controlled Bandwidth Management for HTTP-based Video Delivery

BY

Kevin J. Ma

M.S., University of New Hampshire, 2004
B.S., University of Illinois, 1998

DISSERTATION

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

in

Computer Science

May 2012

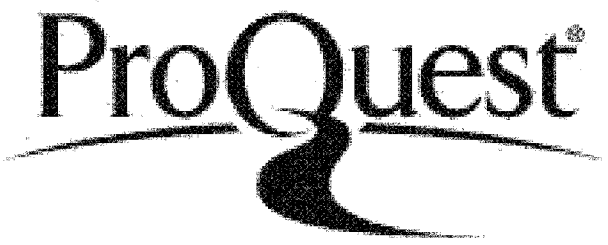
UMI Number: 3525058

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI 3525058
Published by ProQuest LLC 2012. Copyright in the Dissertation held by the Author.
Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code.



ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

This dissertation has been examined and approved.

Dissertation director, Dr. Radim Bartos
Associate Professor of Computer Science

Dr. Michel Charpentier
Associate Professor of Computer Science

Dr. Philip J. Hatcher
Professor of Computer Science

Dr. Raj Nair
CTO Azuki Systems

Dr. Ted M. Sparr
Professor of Computer Science

Dr. Craig E. Wills
Professor of Computer Science

Date

Dedication

For Mom, Dad, and Kathy.

Acknowledgments

I would like to thank my parents, Dr. Joseph K.H. Ma and Dr. Jane Y.C. Ma for instilling in me at an early age an understanding of the importance and power of education, to us as individuals, as well as to the community at large. They cultured in me the desire to begin this journey and the ambition to see it through, though always with the understanding that scholarship is not conferred at graduation, it is a lifetime pursuit. I am thankful for their continuous encouragement and constant example.

I would also like to thank my adviser, Dr. Radim Bartoš for his many years of guidance, understanding, and friendship. Throughout my long and unconventional course of study, Professor Bartoš has been a willing collaborator and unwavering champion. I am grateful for all the insightful discussions, cheerful advice, and limitless enthusiasm.

Contents

Dedication iii

Acknowledgments iv

List of Tables ix

List of Figures xii

Abstract xiv

1 Introduction **1**

1.1 Video Delivery Paradigms 2

1.1.1 Video Delivery Prioritization 7

1.1.1.1 Segment Caching 7

1.1.1.2 Static Frame Reordering 9

1.1.1.3 Packet Prioritization and Frame Dropping 10

1.2 Rate Adaptation 12

1.2.1 Video Encoding 14

1.2.1.1 Multiple Sender Encodings 16

1.2.1.2 Content Encryption 18

1.2.2 Playout Rate Adaptation 20

1.3 OTT Video Delivery 22

1.4 Outline of Dissertation 24

2 HTTP Progressive Download Server **26**

2.1 Pacing Algorithm 27

2.2 Intelligent Bursting 28

2.3 Experimental Configuration 29

2.4 Experimental Results 29
2.4.1 Initial Playback Latency Comparison 30
2.4.2 Total Download Time Comparison 33
2.4.3 Server Bandwidth Usage Comparison 36
2.4.4 Small File Download Comparison 38
2.5 Summary of Server-side Pacing Contributions 40

3 HTTP Adaptive Bitrate Client **42**
3.1 Stitched Media Files 44
3.2 Stitched Media Player 46
3.2.1 Rate Adaptation Latency 47
3.3 Stitched Media File Encryption 50
3.3.1 Pseudo-segment Download and Decryption 51
3.3.2 Cipher Performance 53
3.3.3 Rate Adaptation Timing 56
3.4 RTP Segment Files 58
3.5 Server-side Pacing Transparency 60
3.6 Summary of Client Rate Adaptation Contributions 60

4 HTTP Adaptive Streaming Bitrate Selection Override Proxy **62**
4.1 Network Proxy Controlled Rate Adaptation 65
4.1.1 Optimized Bandwidth Usage Testbed 66
4.1.2 Optimized Bandwidth Usage Results 69
4.2 Network Proxy Capped Rate Adaptation 72
4.2.1 Dynamic Rate Capping Testbed 73
4.2.2 Dynamic Rate Capping Results 75
4.3 Session Detection 77
4.4 Proxy-based CoS Enforcement 78
4.4.1 CoS Rate Capping Results 81

4.4.2 Rate Selection Transparency 83
4.5 Summary of Bitrate Selection Override Proxy Contributions 83

5 HTTP Adaptive Streaming Rate Adaptation Algorithm 86

5.1 Segment-based Queueing Model 86
5.2 Rate Adaptation Algorithm 88
5.3 Segment Download Timeouts 89
5.3.1 Random Up-switch Initiation 91
5.3.2 Random Backoff 92
5.4 Rate Adaptation Callback Procedures 93
5.5 Rate Distribution Numerical Estimation 95
5.6 Multi-Client Rate Adaptation Simulation Environment 96
5.6.1 Simulation Configuration 97
5.6.2 Simulation Evaluation Metrics 99
5.7 Multi-Client Rate Adaptation Simulation Results 99
5.7.1 Multi-Client Rate Adaptation CoS Differentiation 100
5.7.2 Random Backoff and Up-switch Initiation Performance 104
5.7.3 Numerical Estimation Comparison 108
5.7.4 Manifest-based CoS Differentiation Comparison 111
5.7.5 Bitrate Consistency Comparison 115
5.8 Network Throughput Dependency 116
5.9 Summary of Rate Adaptation Algorithm Contributions 117

6 Conclusion 119

6.1 Server Pacing 121
6.2 Client Rate Adaptation 121
6.3 Network Proxy Bitrate Override 123
6.4 Distributed CoS Enforcement 124
6.5 HTTP Adaptive Streaming 126

Bibliography **127**

A Abbreviations **134**

List of Tables

2.1 HTTP streaming server pacing variables. 28

3.1 Stitched media file rate switch variables. 45

3.2 Stitched media file encryption variables. 50

4.1 Rate selection algorithm variables. 78

5.1 Rate adaptation algorithm variables. 87

5.2 Throughput percentage comparison for random backoff, random up-switch initiation, and latching, $W = 3$. 107

5.3 Throughput percentage comparison for random backoff, random up-switch initiation, and latching, $W = 5$. 108

5.4 Bandwidth usage percentage comparison for single CoS unbounded manifest files, per-CoS bounded manifest files, and multiple CoS unbounded manifest files, $W = 3$. 113

5.5 Bandwidth usage percentage comparison for single CoS unbounded manifest files, per-CoS bounded manifest files, and multiple CoS unbounded manifest files, $W = 5$. 113

List of Figures

1-1 RTSP/RTP/RTCP connection setup packet flow diagram. 4

1-2 Video delivery methods: (a) streaming, (b) straight download, (c) server-paced download, (d) client-paced download. 5

1-3 Time-shift latency for Loopback cache chain. 8

1-4 Frame reordering can minimize the probability of late arrivals. 9

1-5 Frame interleaving can reduce the impact of burst losses. 10

1-6 HLS segment-based HTTP adaptive streaming architecture. 13

1-7 Video encodings: (a) individual bitrate monolithic files, (b) multi-layer encoding, (c) multiple description codings, and (d) per-bitrate segmented files. 15

1-8 Client playback/jitter buffer diagram. 20

1-9 Federated CDN-based OTT video delivery architecture. 23

2-1 Comparison of Apache and Zippy process architectures. 27

2-2 Playback latency for 100 concurrent sessions (1 MB File). 30

2-3 Playback latency for 1000 concurrent sessions (1 MB file). 31

2-4 Playback latency for 100 concurrent sessions (1 MB vs. 6 MB file). . . . 32

2-5 Playback latency for 1000 concurrent sessions (1 MB vs. 6 MB file). . . 33

2-6 Download time for 100 concurrent sessions (1 MB file). 34

2-7 Download time for 1000 concurrent sessions (1 MB file). 35

2-8 Download time for 1000 concurrent sessions (1 MB vs 6 MB file). 36

2-9 Bandwidth usage for 100 concurrent sessions (10 KB file). 37

2-10 Bandwidth usage for 1000 concurrent sessions (10 KB file). 38

2-11 Small file download latency for 100 consecutive sessions. 39

2-12 Small file download latency for 1000 consecutive sessions. 40

3-1 Client-side rate adaptation proxy architectures. 43
3-2 Stitched media file (three bitrates): rate adaptation seek. 44
3-3 Stitched media file rate adaptation latency. 48
3-4 Stitched media file initial playback latency. 49
3-5 Stitched media file (three bitrates): cipher block overlay. 50
3-6 Discretized timing diagram for downloading, decrypting, and rendering. 52
3-7 Stream cipher performance comparison. 54
3-8 Fixed key cipher performance comparison. 55
3-9 HC128 stitched media file playback rate switch trace. 57
3-10 RC4 stitched media file playback rate switch trace. 58
3-11 RTP time-based serialization for segment creation. 59

4-1 Rate selection override network proxy deployment diagram. 63
4-2 Standard HTTP Live Streaming delivery. 64
4-3 Standard HTTP Live Streaming delivery request flow. 65
4-4 Network proxy controlled HTTP Live Streaming delivery. 66
4-5 Network proxy controlled HTTP Live Streaming delivery request flow. . 67
4-6 Bitrate override test network configuration. 68
4-7 Native rate adaptation segment requests. 69
4-8 Network controlled rate adaptation segment selections. 70
4-9 Native vs. network controlled rate adaptation bandwidth consumption. . 71
4-10 Network proxy bandwidth capped HTTP Live Streaming delivery. . . . 73
4-11 Network proxy bandwidth capped HTTP Live Streaming delivery request flow. 74
4-12 Bandwidth cap test network configuration. 74
4-13 Network proxy capped segment requests. 76
4-14 Network proxy capped segment requests (multiple CoS). 82

5-1 Average client bitrate for: (a) single Cos ($W = 1$), (b) multiple CoS ($W = 3$), and (c) multiple CoS ($W = 5$). 101

5-2 Network throughput for: (a) single CoS, (b) multiple CoS $W = 3$, and (c) multiple CoS $W = 5$. 103

5-3 Network throughput percentage for: (a) no random backoff, random up-switch initiation, or latching, (b) with random backoff, but no random up-switch initiation or latching, (c) with random backoff and random up-switch initiation, but no latching, and (d) with random backoff, random up-switch initiation, and latching, $W = 3$. 105

5-4 Network throughput percentage for: (a) no random backoff, random up-switch initiation, or latching, (b) with random backoff, but no random up-switch initiation or latching, (c) with random backoff and random up-switch initiation, but no latching, and (d) with random backoff, random up-switch initiation, and latching, $W = 5$. 106

5-5 Average client bitrate for: (a) round robin bandwidth distribution vs. (b) $G(x)$ numerical estimate, $W = 3$. 109

5-6 Average client bitrate for: (a) round robin bandwidth distribution vs. (b) $G(x)$ numerical estimate, $W = 5$. 110

5-7 Per-CoS goodput distribution for: (a) single CoS unbounded manifest files, (b) per-CoS bounded manifest files, and (c) multiple CoS unbounded manifest files, $W = 3$. 112

5-8 Per-CoS goodput distribution for: (a) single CoS unbounded manifest files, (b) per-CoS bounded manifest files, and (c) multiple CoS unbounded manifest files, $W = 5$. 114

5-9 Average number of client rate switches per CoS for: (a) single CoS vs. multiple CoS $W = 3$, and (b) single CoS vs. multiple CoS $W = 5$. . . . 115

ABSTRACT

PROVIDER-CONTROLLED BANDWIDTH MANAGEMENT

FOR HTTP-BASED VIDEO DELIVERY

by

Kevin J. Ma
University of New Hampshire, May, 2012

Over the past few years, a revolution in video delivery technology has taken place as mobile viewers and over-the-top (OTT) distribution paradigms have significantly changed the landscape of video delivery services. For decades, high quality video was only available in the home via linear television or physical media. Though Web-based services brought video to desktop and laptop computers, the dominance of proprietary delivery protocols and codecs inhibited research efforts. The recent emergence of HTTP adaptive streaming protocols has prompted a re-evaluation of legacy video delivery paradigms and introduced new questions as to the scalability and manageability of OTT video delivery.

This dissertation addresses the question of how to enable for content and network service providers the ability to monitor and manage large numbers of HTTP adaptive streaming clients in an OTT environment. Our early work focused on demonstrating the viability of server-side pacing schemes to produce an HTTP-based streaming server. We also investigated the ability of client-side pacing schemes to work with both commodity HTTP servers and our HTTP streaming server. Continuing our client-side pacing research, we developed our own client-side data proxy architecture which was implemented on a variety of mobile devices and operating systems. We used the portable client architecture as a platform for investigating different rate adaptation schemes and algorithms. We then concentrated on evaluating the network impact of multiple adaptive bitrate clients competing for limited network resources, and developing schemes for enforcing fair access to network resources.

The main contribution of this dissertation is the definition of segment-level client and network techniques for enforcing class of service (CoS) differentiation between OTT HTTP adaptive streaming clients. We developed a segment-level network proxy architecture which works transparently with adaptive bitrate clients through the use of segment replacement. We also defined a segment-level rate adaptation algorithm which uses download aborts to enforce CoS differentiation across distributed independent clients. The segment-level abstraction more accurately models application-network interactions and highlights the difference between segment-level and packet-level time scales. Our segment-level CoS enforcement techniques provide a foundation for creating scalable managed OTT video delivery services.

Chapter 1

Introduction

Over the past decade, the role of on-demand video delivery has grown significantly. Increases in broadband Internet access and high-speed cellular data connections have countered what could previously be considered prohibitively large amounts of data required by video applications. Initial forays into Internet-based video delivery were limited to low resolution clips, much of which was user generated content (UGC), through services like YouTube. This eventually evolved into standard definition professional content, distributed through content aggregators like Hulu. Today, premium content providers are going direct to consumers with full-length high-definition content, delivered in an over-the-top (OTT) manner. OTT delivery bypasses traditional broadcast television providers, multiple system operators (MSOs) and mobile network operators (MNOs), and delivers video over generic data connections. This revolution in video delivery paradigms has empowered users, allowing them to tailor their viewing schedules around other aspects of their busy lives.

The dramatic shift in consumer viewing habits, from a rigid in-home linear television model to a more flexible mobile on-demand model, has caught the attention of traditional television and Web-based video distribution services. Two of the most significant contributing factors have been the popularity of the Apple iPhone and iPad and the popularity of the Netflix streaming service. Though Apple was not the first to manufacture a smart phone or tablet, nor were they the first to define a segment-based HTTP delivery protocol, i.e., the HTTP Live Streaming (HLS) protocol, they were the first to integrate a segment-based HTTP delivery protocol into a mobile device that became instantly popular with tens of millions of consumers. In the same way, Netflix was not the first to deliver content to

televisions using an OTT distribution model, but they were the first to do so with great commercial success. The ubiquitous availability of video to mobile users provided by smart phones and tablets, combined with the on-demand convenience of OTT streaming services has created a higher level of expectation among viewers seeking both quality and flexibility in their content delivery.

To fully comprehend this shift in video delivery paradigms, we must first investigate the video streaming protocols which came before, but with the appreciation that the assumptions which motivated both their design and optimization may require re-evaluation. In the following sections we follow the migration of streaming technologies as they have kept pace with advances in the server, networking, and storage technologies on which they rely. We can then see the incremental addition of features that eventually led to a reassessment and simplification of the delivery paradigm. From there, we can begin to address new enhancements to OTT delivery methods, as well as consider the modification and reapplication of previous streaming protocol optimizations.

1.1 Video Delivery Paradigms

Beyond the many orders of magnitude increases in bandwidth available to consumer devices, what may be considered the primary factor in the migration to OTT delivery is the acceptance that, in many cases, *near-real-time* delivery (or *time-shifted* delivery) of content is as good as, if not better than, real-time delivery of the same content. Though interactive video applications like tele-conferencing are very sensitive to real-time latencies, video on demand (VoD), linear broadcast television, and even live event broadcasts, e.g., news and sports, are more delay tolerant. The uni-directional, non-interactive nature of these video streams allow delays, on the order of tens of seconds, to be less discernible and therefore more tolerable. Such delays are standard in television broadcasts to allow network censors to review content, to provide notification for regionalized advertisement replacement, and to reformat video content to the specification of individual MSOs/MNOs. Understanding that certain content, though it appears to be delivered in real-time, is actually tape-delayed,

and accepting that delivery delays on the same order would be acceptable, has resulted in a change in the way video delivery protocols are defined.

The primary delivery protocol for OTT video services is the Hyper-Text Transfer Protocol (HTTP) [1]. The evolution of HTTP-based video delivery had been slow, until recently, as TCP, the underlying transport protocol for HTTP, had long been considered to be unsuitable for video applications, due to the latency and overhead of TCP's sliding window and retransmissions. Video delivery had long relied on protocols like the Real-time Transport Protocol (RTP) [2]. RTP uses UDP as an underlying transport protocol, which does not have the overhead or latency of blocking on retransmissions. RTP uses a just-in-time (JIT) frame-based delivery model; individual video frames are sent out in a paced manner, and late or lost frames are ignored by the receiver. The paced delivery evenly distributes bandwidth over time and is inherently conducive to supporting real-time, multicast and interactive applications like live event broadcasts and tele-conferencing. One of the primary disadvantage of RTP, however, is that network interruptions which cause frames to be late or lost are likely to cause video artifacts and distortion, lowering the quality of experience (QoE) for the user. RTP-based delivery is also more complex. It typically relies on the Real-Time Streaming Protocol (RTSP) [3] as a control plane protocol, and requires not only separate UDP connections for each audio and/or video channel, but also additional UDP connections for Real-time Transport Control Protocol (RTCP) channels, one per RTP connection.

Figure 1-1 shows a packet flow diagram for a typical RTSP/RTP streaming session. The RTSP session sets up separate video and audio RTP (and corresponding RTCP) connections over which the content is streamed. Through the RTCP control channel, the server provides audio/video synchronization information to the client, and the client provides feedback on network jitter and packet loss. Where an HTTP-based approach would only require a single connection, the extra UDP connections for RTP/RTCP consume additional server resources, impacting streaming server scalability. The UDP connections also require dynamic firewall "fixup" support, since the UDP connections are server-initiated, as opposed to the client

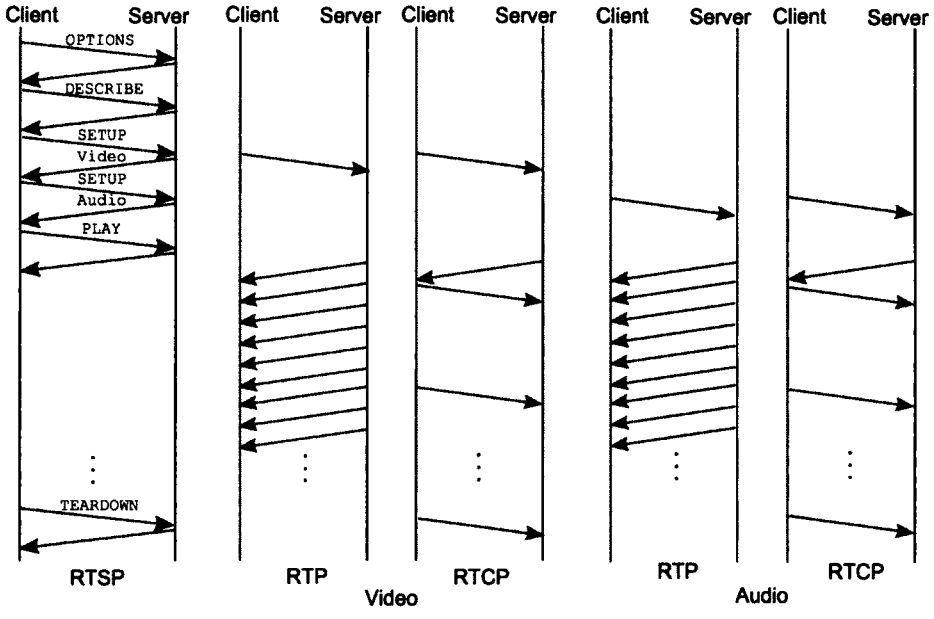


Figure 1-1: RTSP/RTP/RTCP connection setup packet flow diagram.

initiated RTSP (or HTTP) connection. In addition, frame-based delivery requires the RTP streaming server to parse the video file in order to extract the frames, which adds additional overhead and impacts scalability. Though the session initiation protocol (SIP) [4] may also be used as a control protocol in lieu of RTSP [5], we are primarily concerned with the data delivery portion, i.e., RTP and RTCP), and specifically understanding how it compares with HTTP. Figures 1-2 (a)-(d) compare the basic traffic pattern for streaming RTP delivery, with three methods for HTTP-based delivery.

Figure 1-2 (a) depicts a simplified view of the RTSP/RTP streaming session (a condensed version of Figure 1-1), concentrating on the data delivery aspect of individual frames being paced out by the server. In direct contrast, Figure 1-2 (b) depicts a straight HTTP as-fast-as-possible download with no delivery pacing. Comparing Figures 1-2 (a) and (b), there is an obvious bandwidth distribution advantage to using streaming, especially for large files or long-lived streams. However, straight download is not the only method of using HTTP for delivery. Figures 1-2 (c) and (d) show server-side and client-side approaches, respectively, for paced HTTP progressive download. In Figure 1-2 (c), the server paces out bursts of

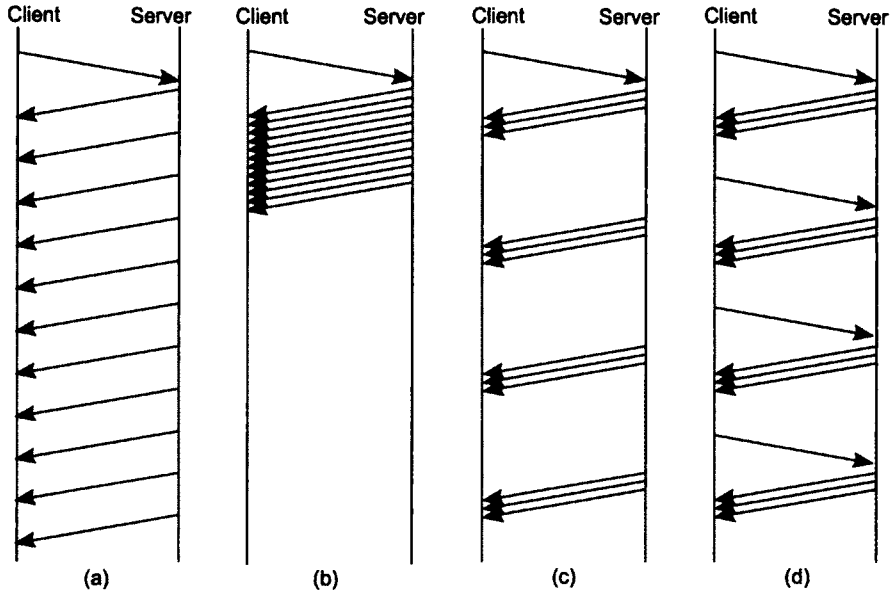


Figure 1-2: Video delivery methods: (a) streaming, (b) straight download, (c) server-paced download, (d) client-paced download.

packets, while in Figure 1-2 (d), the client paces out requests for partial content downloads. In both cases, the individual bursts or partial content requests are larger than an individual frame, and the inter-burst and inter-request gaps are larger than the inter-frame gap of the streaming case shown in Figure 1-2 (a). The larger burst and/or request sizes are used intentionally to create larger inter-burst and inter-request gaps, which provide larger time windows for performing any necessary TCP retransmissions. The burst of packets also means that all the frames should arrive well before their playout time, i.e., no frames should be late. Assuming sufficient time and bandwidth exist to perform retransmissions, the use of a TCP-based protocol, with retransmissions, prevents the playback artifacts commonly seen with RTP-based delivery. Using HTTP-based delivery with paced bursts, frames should never be late, and lost frames should be retransmitted, therefore playback should never exhibit distortion and viewers should always have a high QoE.

Comparing the server-side pacing scheme shown in Figure 1-2 (c) to the client-side pacing scheme shown in Figure 1-2 (d), the only visible difference in the traffic patterns is the additional overhead for each individual partial request in the client-side pacing approach.

Clients typically issue partial content requests using either HTTP Range GETs, or by retrieving pre-processed file segments. In the server-side approach, however, the server itself requires content awareness (i.e., what the encoded rate of the video is, and what the pacing rate should be), though not as much as an RTSP/RTP server. The primary advantage of the client-side pacing approach, is that ceding control to the client, allows the client to make decisions on what content it retrieves, without the need for alternate control channels, e.g., RTSP and RTCP. Client-side decision-making, specifically client-side rate adaptation algorithms (see Section 1.2), are a key factor in the popularity of client-side pacing schemes, as they allow the client to adjust to changes in available bandwidth, to prevent playback stoppages.

The primary criticism of HTTP, historically, was its inability to stream data, as HTTP is an as-fast-as-possible file download protocol. However, progressive download paradigms, e.g., server paced delivery [6] and the client paced delivery [7, 8] have clearly emerged and exhibited commercial success. Though critics continued to cite TCP retransmission latency as an issue, studies have shown that TCP is quite suitable for video delivery [9, 10] and that multiple TCP connections can significantly improve delivery efficiency [11, 12]. HTTP-based segmented delivery provides inherent methods for combatting late and lost frames, and the inclusion of rate adaptation provides a method for avoiding playback stoppage. Over the past few decades, though, many techniques have been developed and applied to RTP-based delivery, in order achieve similar QoE goals. Being predicated on the idea that bandwidth is scarce and that JIT delivery is a primary solution to addressing bandwidth scarcity issues (as opposed to JIT delivery being the solution to latency issues in real-time interactive communications), most RTP optimizations revolve around the philosophy of limiting, rather than preventing video distortions, though many also address initial playback latency reduction.

1.1.1 Video Delivery Prioritization

Many schemes have been proposed, to improve the quality and predictability of RTP-delivered video. Standard techniques like caching and packet prioritization have been adapted to support higher quality delivered video in challenged (lossy) networks. The goal of all of these optimizations is to temporally position delivered video frames, within the context of JIT delivery, with the highest possible rendering quality. In this section we look at three such techniques: segment caching, frame reordering, and packet prioritization.

1.1.1.1 Segment Caching

Caching schemes are typically used to: reduce latency by storing content at individual points of presence (POPs), which are typically closer to clients than the content origin server, and reduce load on wide area network (WANs) and the Internet core. While common for HTTP-based delivery, which typically rely on highly scalable and commoditized content delivery networks (CDNs) built on top of hierarchical caching infrastructures, caching is less prevalent in real-time frame-based delivery.

The proxy prefix caching scheme proposed by Sen et al. was an approach for live stream caching, where the proxy stores recently transmitted frames and uses them as a cached "prefix" for newly initiated streaming session [13]. The proxy essentially extends the buffering capabilities of the client, and pre-buffers content prior to the client requesting it. It must be noted that the buffer, in essence, is a time-shifting buffer which shifts the real-time nature of the source stream to a near-real-time delivery stream. Time-shifted delivery in RTP streaming has been accepted for well over a decade. Kalapriya and Nandy apply a similar scheme to peer-to-peer (P2P) streaming [14], while Kusmierek et al. take time shifting literally to the N^{th} degree with their "Loopback" caching scheme [15].

Kusmierek at al. combine caching and P2P by chaining individual P2P clients together, where each P2P client acts as a cache for all other clients downstream of it in the chain [15]. Each client buffers up to δ seconds of data, and any new session initiated within δ seconds of the previous session start, are added to the end of the chain. Given a chain of N clients, with

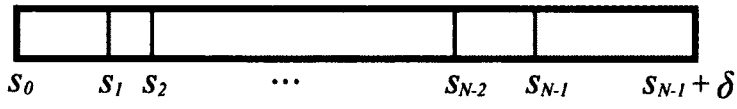


Figure 1-3: Time-shift latency for Loopback cache chain.

start times $S = \{s_0, \ldots, s_{N-1}\}$, their scheme ensures that: $\forall i \in 1 \ldots N-1 : s_{i+1} - s_i <= \delta$. The last client in the chain always buffers δ seconds worth of data; the other clients in the chain buffer $s_{i+1} - s_i$ seconds worth of data (i.e., they buffer only enough to service the next client in the chain). Figure 1-3 shows the compounded near-real-time latency of $s_{N-1} - s_0$, for the Loopback chain. In the worst case, the N^{th} client could experience a time-shift latency of up to $(N-1) \cdot \delta$ seconds. Deshpande and Noh use a similar P2P caching scheme with temporally proximate peers caching different portions of data [16], though their approach lacks the chaining structure and initial start time enforcement. Deshpande and Noh allow the client to select any start time and then find the node with the desired start time cached.

Others have extended the prefix caching concept, though most are more focused on optimizing cache efficiency (in challenged caches with limited capacity) for VoD, as opposed to live streams. Shen et al. extended the definition of a prefix for VoD to account for popularity-based random starting points within a VoD asset [17]. They allow for multiple "prefixes", one per common starting point. Li et al. further extended popularity-based caching using variable segment sizes, by applying segment coalescence for adjacent segments [18]. Because coalescence reduces starting point flexibility, they apply a popularity-aware coalescence and eviction scheme. Tu et al. take a similar popularity-based approach, but at a much finer key-frame granularity [19].

Segment-based caching has also been extended beyond just prefix caching, for VoD assets. Ma and Du proposed a scheme where every other segment was cached, to reduce the impact of WAN latency by half [20]. Chen et al. produced a series of work which combine popularity-based segment prefetching and eviction to lower cache miss latency,

which they refer to as "proxy jitter" [21, 22, 23]. Though these generic segment caching schemes do not directly impact frame-based delivery, they do act as a coarse level (segment level) prioritization of content to be delivered. More importantly, they highlight the general acceptance of segment-based video formatting and storage in RTP streaming environments.

We can see from these prefix and segment caching approaches that time-shifted delivery and segment-based representations, both of which are critical requirements for HTTP-based video delivery, and both of which have been traditionally cited as disadvantages of HTTP-based video delivery approaches, have firm foundations in RTP-based delivery. Though the context in which these schemes were initially applied to RTP-based delivery may differ from the context in which they add value to HTTP-based delivery, the fundamental acceptance of these concepts should be universal.

1.1.1.2 Static Frame Reordering

Frame reordering comes in many forms. In this section we look at static methods for performing frame reordering in the encoded source. Static frame reordering seeks to select a frame order which is statistically less susceptible to distortion when frame losses or delays occur. Wee et al. proposed a model of measuring the distortion caused by late frames, where, unlike lost frames, late frames may still be used in the decoding of future frames. They proposed moving frames up so that they would arrive earlier, reducing the probability that they arrive too late [24]. Figure 1-4 shows an example of advancing frame f to position $f - k$, and delaying the k frames which had to be shifted out.

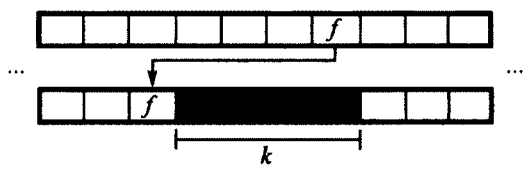


Figure 1-4: Frame reordering can minimize the probability of late arrivals.

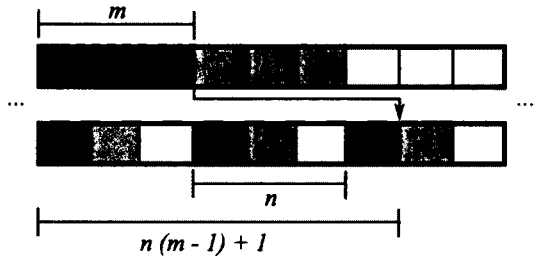


Figure 1-5: Frame interleaving can reduce the impact of burst losses.

Liang et al. also looked at the interdependencies of frames but propose a different solution which relies on live encoding [25]. Their approach relies on detecting frame loss through a client feedback mechanism. When a key frame is lost, future frames are encoded to a different key frame which was not lost and is likely to still be in the client's playout buffer. The scheme relies on a low latency feedback channel, as well as storage and maintenance of the distortion prediction tables. They expanded their investigation to include the impact of burst losses [26]. They then proposed using a frame interleaving scheme to prevent key frame loss in the presence of burst losses [27]. Figure 1-4 shows an example of n blocks of m frames each being interleaved, which reduces the temporal correlation of adjacent frames, but introduces an addition delivery latency of $n \cdot (m-1)+1$, for each block.

1.1.1.3 Packet Prioritization and Frame Dropping

When network congestion occurs, routers will naturally drop packets as their limited queue capacity is exceeded. Generally, individual applications do not have the ability to directly influence which packets get dropped in those situations. Assuming the application can detect when congestion occurs, however, it may be able to reprioritize its packets prior to sending them and pre-emptively drop lower priority packets, never actually sending them into the network. Though these types of schemes are most prevalent in peer-to-peer (P2P) applications, where application routing overlays are common, they may be generalized and applied to more traditional network routers with video routing intelligence.

Baccichet et al. proposed a P2P packet prioritization scheme based on a distortion factor

D, where frames which would result in greater distortion, if lost, are deemed to be of higher priority [28]. Setton et al. used a similar calculation in their P2P packet prioritization, however they consider the cumulative impact of packet loss on downstream clients. The distortion factor D is weighted by the number of client N that would be affected by the packet loss [29]. J. Li et al. further augment the calculation by considering the processing time as a penalty for each packet. In their P2P model, the distortion factor D is weighted by $\frac{N}{T}$ where T is the time to process the given packet [30]. In the degenerate case where all packets have equal processing cost, J. Li's scheme is strictly proportional to Setton's, though, it is conceivable that larger packets (e.g., those containing key frames) will result in greater distortion if lost, however, the larger size may make it more susceptible to loss and may prevent many other smaller packets from being processed.

One issue with P2P based application routing overlays is that they have no control over intermediate network routers and do not necessarily have insight into actual network health. Though protocols like ALTO [31] are under development to provide network information to P2P system, the level of aggregation may not be sufficient for packet level decision making. Chakareski and Frossard proposed a scheme for implementing distortion estimation and packet prioritization in the network nodes [32]. Y. Li et al. proposed a packet marking scheme which uses distortion factor prioritized marks, so that intermediate routers can use that information for packet dropping [33]. Tan et al. proposed an alternate packet marking scheme where marks generated by the streaming server contain estimates of client buffer fullness, so that intermediate routers can use that information for packet dropping [34]. Coordinated packet dropping has also been proposed in conjunction with content-aware playout. Li et al. proposed the use of motion intensity estimation for reordering and prioritization of packets, where frame dropping is performed in conjunction with playout rate reduction [35]. Playout rate reduction is discussed further in the rate adaptation section below.

What becomes evident from the survey of frame reordering and packet prioritization schemes is that the JIT nature of the streaming paradigm complicates video delivery with

concerns about frame lateness and frame loss. The arrival time of any individual frame is not typically a concern with HTTP-based delivery paradigms, as segment completion (not frame arrival) is the measure of success, and frame loss is handled through TCP retransmission. Intentionally dropping frames does not make sense with TCP-based protocols, and the difference in time scale between segments and frames render packet-based prioritization inconsequential. Because RTP-based schemes operate at the frame level, RTP optimizations act at the frame level. From this, we can draw that for HTTP-based schemes, which operate at the segment level, it makes sense for HTTP optimizations to act at the segment level.

1.2 Rate Adaptation

The segment-based client pacing scheme, as depicted in Figure 1-2 (d), has become the preferred method for HTTP-based video delivery. The HTTP Live Streaming (HLS) [36] and Dynamic Adaptive Streaming over HTTP (DASH) [37] protocols are examples of open-standard segment-based HTTP adaptive streaming protocols which use client-side pacing. Microsoft® SilverLight™ Smooth Streaming and Adobe® HTTP Dynamic Streaming are also examples of segment-based HTTP adaptive streaming protocols, though they are proprietary in nature.

HTTP adaptive streaming protocols use segment boundaries, not only for paced progressive download, but also for initiating bitrate adaptation. Manifest files are used to convey bitrate and segment information to the clients. Figure 1-6 shows the components of a typical HLS delivery configuration. A master manifest file contains a list of individual bitrate manifest files locations, providing encoding information for each representation. The individual bitrate manifest files contain lists of segment file locations. The segments for each bitrate are synchronized so that segment boundaries are aligned, making switching between bitrates seamless when performed on segment boundaries. While frame-based RTP streams have a minimum real-time latency of a single frame time (i.e., $\frac{1}{30}^{th}$ of a second, for a typical video), segment-based streams have a minimum real-time latency of a single segment duration (e.g., 10 seconds, for a typical HLS deployment). Segments can be produced

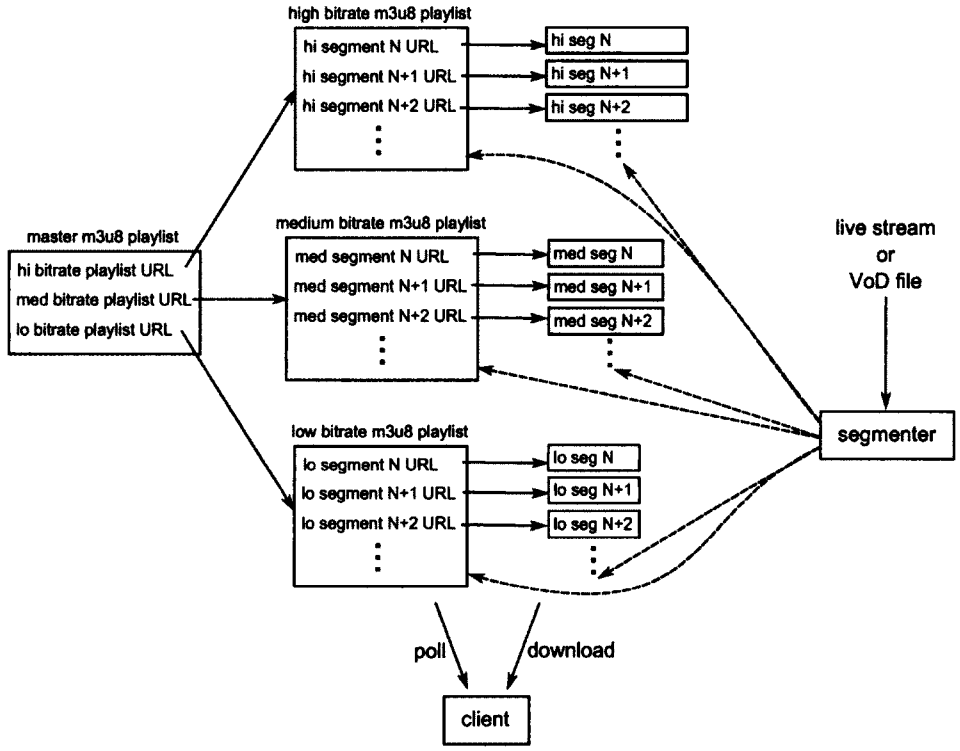


Figure 1-6: HLS segment-based HTTP adaptive streaming architecture.

from either a live input stream or a VoD source file. For live inputs, the near-real-time latency is a multiple of the segment duration, rather than a multiple of the frame time. As we have seen though (see Section 1.1.1.1), increasing near-real-time latency is a side-effect of many streaming delivery optimization schemes. For non-interactive video applications, the impact of segment boundary time-shift latency is typically negligible.

HTTP adaptive streaming clients automatically adjust to changes in available bandwidth, though, different clients exhibit various levels of aggressiveness [38]. More aggressive clients often exhibit bitrate oscillation, while less aggressive clients may not achieve bitrate parity. HTTP adaptive streaming clients estimate available bandwidth (e.g., by measuring segment download speed or tracking segment download buffer occupancy) and use that information to select new bitrates based, as network conditions change [39]. HTTP adaptive streaming media players typically buffer a small number of segments as a jitter buffer. Given that this can amount to tens of seconds worth of buffered data, bandwidth esti-

mations can be smoothed over many seconds without significant impact playback or QoE. Understanding the difference in time scales between RTP frame-based delivery and HTTP segment-based delivery is important when comparing rate adaptation approaches. Packet-level traffic management schemes (like the ones surveyed in Section 1.1.1.3) can adversely affect QoE for segment-based delivery. Similarly, packet-level rate estimation schemes which act on instantaneous bandwidth measurements can also impact QoE. TCP-based rate estimation [40] and TCP-friends segment-size optimizations [41] have been proposed for HTTP adaptive streaming schemes. However, because the typical TCP window size is so much smaller than both the segment and jitter buffer sizes, it does not accurately reflect the time scale on which rate adaptation measurements should be made. When analyzed over a longer time scale, segment download burst interleaving for paced segment requests can naturally smooth overall bandwidth usage, even though individual clients may monopolize bandwidth allocations in the short-term [42]. Amortizing bandwidth measurements over a time scale relative to the segment durations [43] is generally a better abstraction.

1.2.1 Video Encoding

In order to perform bitrate adaptation, a video must be encoded at multiple bitrates. Though methods have been proposed which rely on dynamic transcoding [44, 45, 46, 47], the processing resources required for these types of approaches make them unsuitable for VoD applications due to the inherent scalability issues. Most VoD adaptive bitrate streaming approaches use a discrete set of pre-selected bitrates at which videos are pre-encoded. Figure 1-7 shows a variety of different encoding video methods. Figure 1-7 (a) shows individual monolithic files, one per bitrate. Figure 1-7 (b) shows a cumulative layered encoding scheme, where a base encoding may be progressively enhanced by one or more higher layer encodings, though each layer is dependent upon all the layers below it [48]. H.264 scalable video coding (SVC) is one of the most commonly used layered encodings [49]. Figure 1-7 (c) shows a multiple description coding (MDC) scheme [50], where each description is independently playable (unlike layered encodings), but when combined, multiple descriptions can be

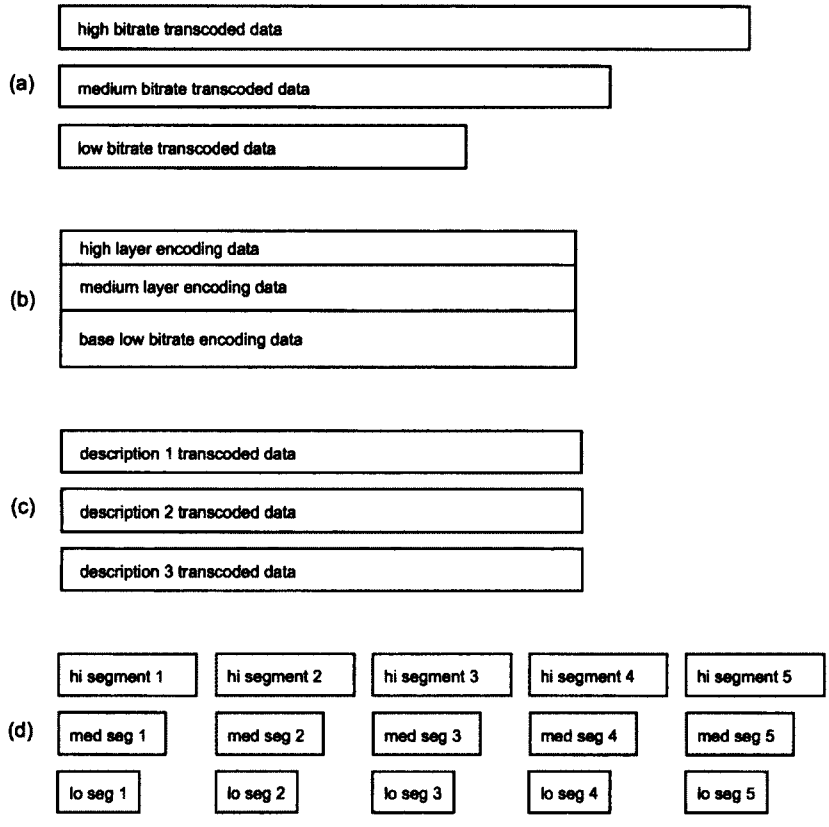


Figure 1-7: Video encodings: (a) individual bitrate monolithic files, (b) multi-layer encoding, (c) multiple description codings, and (d) per-bitrate segmented files.

used to enhance video quality. MDC schemes have the added advantage of error resiliency, but often pay a penalty for data redundancy [51]. Figure 1-7 (d) shows a segment-based format, suitable for use with HTTP adaptive streaming protocols like HLS or DASH.

Segments can be created from any of the encodings shown in Figures 1-7 (a)-(c), though, for HLS and DASH, segments are typically created from individual monolithic bitrate files, as shown in Figure 1-7 (a). Individual monolithic files are also used by RTSP streaming servers in file switching rate adaptation schemes [52, 53]. In the RTSP case, the streaming server is parsing monolithic files to enable frame-based delivery. Switching files incurs a processing penalty for seeking to the current location in an alternate file, but the general penalty for parsing out frames is already high. RTP-based schemes are also inherently reactive, rather an proactive. Rate adaptation is typically performed in response to high

frame loss being reported by clients, via RTCP receiver reports. The QoE has already been unpredictably degraded at that point. Though rate adaptation does reduce quality, it does so in a controlled manner, with the individual encodings typically having gone through rigorous quality control evaluations.

Though HTTP adaptive streaming, to this point, has concentrated on single layer, single description encodings, a fair amount of work on MDC and SVC for legacy streaming video has been done. The use of multiple layers or descriptions often incurs an overhead penalty for the use of parallel connection. In most cases, this is justified by ensuring path diversity between connections, using different servers to source each stream [54]. Reliance on multiple senders, however, introduces a new problem where client feedback needs to be distributed to all servers, and rate selection needs to be coordinated between all servers. In the following sections we look at a couple of these schemes.

1.2.1.1 Multiple Sender Encodings

Chakareski and Girod proposed a scheme where clients measure the download rates of "media packets" at fixed intervals. After each interval, a "transmission opportunity" occurs where the client must issue requests to each server from which it wishes to receive data [55]. In their scheme each server distributes a different layered encoding, however, the striking similarity of this scheme to the segment-based client rate selection of HTTP adaptive streaming should be noted. Chakareski and Frossard proposed an alternate scheme in which clients report bandwidth estimates back to the servers, allowing the servers to make streaming transmission decisions, rather than allowing the client to make rate selection decisions [56]. The media is again "packetized" into segment-like structures, and simple packet partitioning is performed between servers, rather than layer partitioning. The distributed scheme relies on clients sending bandwidth estimates for all paths to all servers (the overhead of which is not discussed). The servers all run the same algorithm to determine the packet partitioning and sender scheduling. In their follow up work, Chakareski and Frossard included support for sending different MDC encodings from each server [57].

Nguyen and Zakhor had previously proposed a slightly different approach whereby clients estimate available bandwidth based on round trip time (as reported by the server) and compute the packet partitioning and sending schedules for the servers [58]. In their follow up work, Nguyen and Zakhor include packet loss probability in their scheduling calculations [59].

In a different approach, Guo et al. attempt to estimate path diversity among many servers using standard networking characteristics, e.g., round trip time (RTT) and hop count. Then, given a fixed number of MDC encodings, retrieves each encoding from a different server [60]. While Guo et al. focused on a CDN scenario, De Mauro et al. applied similar techniques to selecting peers in a P2P scenario [61]. Apostolopoulos et al. also looked at the performance of MDC encodings in conjunction with CDN-based delivery [62]. They also investigated the effects of burst losses in with path diversity [63]. They then compared different hop count-based and distortion estimation-based algorithms for selecting a subset of servers from which to retrieve MDC encodings [64].

The independent nature of MDC encodings make it better suited to multiple sender schemes, where loss cannot be predicted. If congestion causes packet loss, it does not matter which MDC encoding is lost since the other(s) will still be usable. The inter-dependence of layered encodings like SVC, however, require more care in selecting which layers are transmitted. If a base layer is lost, the rest of the transmitted data is useless, and sending an additional higher layer can cause congestion which may result in a base layer being lost. This makes MDC more versatile for unreliable delivery protocols like RTP [65]. Layered encodings have been of primary interest in multicast environments [66, 67], however, there are cases where individual bitrate simulcast out performs SVC multicast [68]. Layered encodings must typically also be combined with forward error correction (FEC), to reduce the probability of loss in multicast RTP-streaming environments [69, 70, 71, 72]. Though FEC is often used in environments with no explicit feedback, Zhu et al. proposed a scheme in which feedback is used to not only influence rate adaptation, but also to influence the level of FEC used [73]. Though FEC has been a topic of RTP-streaming research for a long

time, it does increase data transmission overhead and is as necessary for MDC encodings which have their own redundancy and overhead built in. There has also been work done on optimizing caching efficiency for layered encodings [74, 75, 76], though these are less delivery oriented.

Though HTTP-based retrieval of layered encodings does not suffer from the base layer loss concerns that occur with RTP-based delivery, the additional overhead of retrieving separate layers from the same server does not make sense in a client-side rate selection approach. And while layered encodings could still be used in path diverse multiple server environments, a partial segment striping approach like the one proposed by Gouache et al. works equally well for multiple servers but does not require the additional encoding and decoding complexity of a layered scheme [77]. The redundancy support offered by MDC encodings makes it a more interesting option for both RTP-based and HTTP-based delivery, but as with layered encodings, the overhead of MDC encoding and decoding is not makes the tradeoff less valuable for HTTP-based approaches. Though path diversity is an important aspect of OTT delivery, basic segment striping or partial segment striping [77] approach is usually sufficient.

1.2.1.2 Content Encryption

Content encryption and digital rights management (DRM) is an important consideration for premium content. Secure streaming protocols, e.g., the Secure Real-time Transport Protocol (SRTP) [78], use transport layer encryption where individual RTP frame payloads are encrypted. SRTP uses AES128 in either counter mode (CTR) or output feedback mode (OFB) as a stream cipher. Stream ciphers generate pseudo-random bit streams and XOR the random data with the frame data to produce an encrypted frame. For RTP streaming servers, which have to parse the source file in order obtain the frames, it is typically assumed that the streaming server can be trusted with unencrypted content and that client devices are implicitly authenticated by having a proper decryption key and will not store the decrypted content after playout. In most real-world deployments, it is assumed

that SRTP receivers are manufactured with pre-burnt-in encryption keys which should only be known to the operator who purchased and will be managing that device. Simplifying assumptions such as that allow most RTP optimization schemes to avoid key exchange concerns. As long as the RTP headers are unencrypted (as they are with SRTP), frame-based optimization (excluding those which involve transcoding) can continue to function.

Quite a few partial encryption (or scrambling) schemes have been proposed to reduce the processing cost of encryption [79]. Some schemes encrypt only certain frames [80]. Other schemes encrypt only portions of frames [81]. There are also proposals for the use of lower complexity ciphers [82] with partial frame encryption. Layered encodings often receive special treatment, as encryption of the base layer typically requires more rigorous treatment than encryption of enhancement layers [83], and key management for per-layer keys also requires consideration [84]. While it may be argued that full encryption is gratuitous for certain applications, it should be noted that the overhead required to isolate partial encryption boundaries is non-null, and the practical effort required to certify partial encryption schemes for commercial deployment is not insignificant.

For HTTP adaptive streaming delivery, where content is typically distributed and accessible through public CDNs, the files in the CDN must be encrypted. HLS uses AES128 in cipher block chaining (CBC) mode to encrypt each segment file [36]. CBC mode relies on a feedback mechanism which first does an XOR of the video data and the previously encrypted data before performing the encryption operation. This creates inter-dependencies within the encrypted data to prevent decryption of file fragments. Though stream cipher-based approaches are typically less processing intensive than CBC mode [8], most modern devices have hardware support for CBC mode decryption. Though transport layer encryption could be used with HTTP (i.e., SSL or TLS) it requires that secure client authentication. While assumptions, similar to the ones made for SRTP clients, could be made whereby every client device needs to be pre-burned with an SSL/TLS client authentication certificate, in reality this is not the case in OTT video delivery. Though the transport layer encryption could prevent third party eavesdroppers from intercepting the data; it would not guarantee

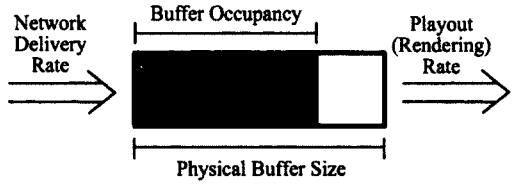


Figure 1-8: Client playback/jitter buffer diagram.

that the requestor actually had the right to access the data. Encryption key management is typically handled outside of the delivery path as part of a separate DRM key exchange protocol [85]. It is important to note the difference in assumptions between RTP-based delivery and HTTP-based delivery, as some delivery optimizations can be complicated by the presence of encryption.

1.2.2 Playout Rate Adaptation

While bitrate adaptation can react to network congestion and proactively reduce load on the network by switching bitrates, playout rate adaptation reacts to variations in network conditions by changing the rate at which the content is being rendered [86]. Figure 1-8 shows a client playback buffer and depicts the relationship between the network delivery rate, i.e., the buffer fill rate, and the playout (or rendering) rate, i.e., the buffer drain rate. Where bitrate adaptation affects both the future network delivery rate and future playout rate, playout rate adaptation only affects the current playout rate. Assuming a constant bitrate b for all video in the playback buffer, the buffer occupancy o, in bits, can be expressed in terms of playout duration d, based on the current playout rate r, such that $d = \frac{o}{r}$. Under normal circumstances, $r = b$. Adjusting playout rate r effectively adjusts the client playback buffer duration d. The playback buffer duration d is inversely proportional to the playout rate r, therefore lowering the playout rate increases the effective buffer occupancy, while increasing the playout rate decreases the effective buffer occupancy.

Videos are typically encoded at a constant frame rate, e.g., 24 or 30 frames per second (fps). Similarly, audio is typically encoded at a constant sample rate, e.g., 22.05, 44.1, or

48 kHz. Playout rate adjustments adjust the rendering rate with respect to the nominal frame rates and sample rates. Rate reduction can be used to reduce the buffer drain rate in response to temporary drops in the buffer fill rate, while increasing the playout rate can be used to increase the buffer drain rate, in order to "catch-up" after a playout rate reduction [87]. Li et al. proposed a scheme in which playout rates are reduced to compensate for variations in delivery latency in wireless network. When collisions occur in wireless networks, packets transmissions fail, but are automatically retried. Though the packet is not lost, it can introduce latency variations. Li et al. propose only reducing playout rate in low motion scenes [88]. Li et al. extended their model to include playout rate reduction for power conservation [89]. Collisions and retransmissions in wireless networks can waste power; adjusting playout to delay transmissions which are likely to collide can save power. As alluded to in Section 1.1.1.3, Li et al. go on to include a frame dropping component for frames that, even with playout rate reduction, will not be delivered on time [35]. It is assumed that the wireless router is both stream aware and client session aware, in order to coordinate playout rate reduction, delayed transmission, and frame dropping.

Kalman et al. use playout rate reduction, not only to adjust to variations in delivery latency, but also to improve initial playback latency [87]. Client media players typically require a minimum amount of data to be buffered before playback starts. Using playout rate reduction to effectively increase the current buffer occupancy (as measured in playout duration), allows the media player to reduce the amount of data (in bits) required to begin playback, assuming a constant network delivery rate.

Hui and Lee proposed a scheme for multiple sender environments which uses playout rate reduction instead of bitrate adaptation [90], though the two do not need to be mutually exclusive. Argyriou proposed a playout rate adaptation scheme as part of a TCP-based client-side pacing scheme [91], though a slightly more aggressive buffering scheme might negate the need for playout rate adjustments. In general, however, playout rate adaptation can be used in conjunction with any of the other bitrate adaptation schemes that we have surveyed. Playout rate can be considered largely orthogonal to the delivery rate. As our

work focuses on bitrate adaptation and not playout rate adaptation, references to rate adaptation should be understood to refer to bitrate adaptation, unless playout rate adaptation is explicitly mentioned.

1.3 OTT Video Delivery

HTTP adaptive streaming protocols (e.g., HLS and DASH) were initially developed to enable mobile clients to adjust to the dynamic nature of cellular networks [92], however, OTT video delivery has begun to emerge in more traditional video distribution outlets (e.g., MSOs and MNOs). HTTP adaptive streaming schemes have focused on maximizing QoE for individual clients, rather than considering overall network throughput, but as the ability to deliver content to multiple screens becomes increasingly important [93], content providers and network providers are becoming more concerned with the ability to provide fair access to content and enforcing minimum QoE levels for all clients. As smart TVs and smart phones and tablets converge on a single HTTP adaptive streaming format, converged media preparation and distribution infrastructures simplify workflow management and reduce operational expenditures. However, without the ability to manage QoE on a per-client basis, monetizing differentiated service levels is not possible.

Linear television typically consists of a combination of live events (e.g., sports or news) and non-live content (e.g., pre-recorded show broadcast at a scheduled time), all delivered in real-time over meticulously provisioned multicast delivery networks. The set-top-boxes (STBs) to which the television content is delivered are typically managed devices, fully under the control of the MSO/MNO. The STBs are registered and activated in a given location (i.e., the home) and are not mobile. The relatively static nature of these clients is what makes finely tuned network provisioning and monitoring possible. In OTT video delivery, however, content is retrieved on-demand by clients which are both unmanaged and mobile. The content is typically stored in and distributed through a CDN caching infrastructure. The on-demand nature OTT delivery removes the ability to take advantage of multicast which increases load on MSO/MNO networks and edge caching nodes. It

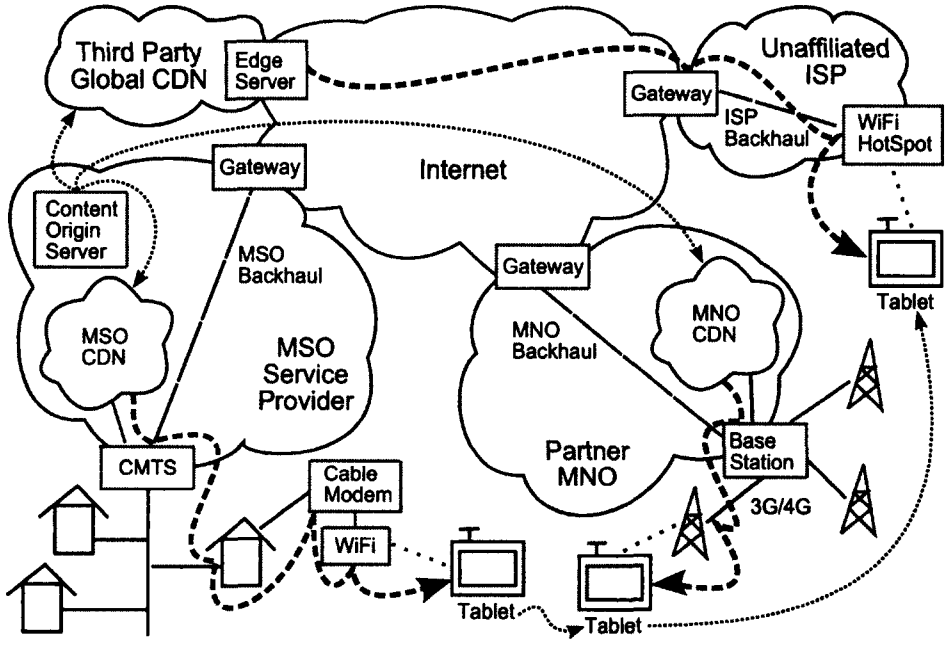


Figure 1-9: Federated CDN-based OTT video delivery architecture.

also makes network load unpredictable. The ability to support flash crowds of OTT clients requires a much greater network elasticity than traditionally provisioned broadcast networks typically have. Migrating to an OTT delivery model therefore requires significant changes to both physical infrastructure and network management paradigms.

OTT video delivery deployments often rely on CDN federation to increase the geographic footprint of delivery. An MSO may choose to operate their own internal CDN for in-network content delivery, but may also contract other third party global CDN for out-of-network content delivery. The internal CDN reduces delivery costs for the majority of content delivered to the home, while third party CDNs abstract the complexity of supporting a global footprint. It is assumed that out-of-network delivery represents a much lower volume of traffic than in-network delivery. Third party global CDNs can be expensive to maintain relationships with and may not always have the reach necessary to optimize delivery to roaming mobile clients. An MSO may also partner with other MSOs or MNOs, negotiating bilateral agreements to combine their internal CDNs and extend their footprints. Sharing

internal CDNs can further reduce costs by reducing reliance on third party global CDNs.

Fig. 1-9 diagrams an example of a federated OTT distribution scenario where the MSO originates content to its own internal CDN, its partner MNO's internal CDN, as well as a third party global CDN. When the mobile OTT client is in the home, content is sourced from the MSO's internal CDN. When the mobile OTT client is roaming on the partner MNO's network, content is sourced from the partner MNO's internal CDN. When the mobile OTT client is attached to the Internet through some unaffiliated Internet service provider (ISP), content is sourced from the global third party CDN. In this case, CDN federation reduces load on the MSO and MNO backhaul links and Internet gateways by taking advantage of the internal CDNs and not having to traverse the public Internet to get to the third party global CDN. The unaffiliated ISP, however, will still experience backhaul and Internet gateway load.

Federated delivery introduces new complexities for video delivery management. Either the mobile OTT client needs to understand which network it is currently connected to and what the CDN access rules are for different networks, or each affiliated partner network needs to be able to recognize the mobile OTT client and route its content requests to the appropriate CDNs. Though the practical aspects of routing and name resolution in federated CDN deployments extend beyond the simple models represented in Section 1.2.1.1, many of the basic principles for multiple server selection do apply.

1.4 Outline of Dissertation

This dissertation addresses the need for improved scalability in on-demand video delivery applications, specifically through the use of provider-controlled bandwidth management for HTTP adaptive streaming. As MSOs and MNOs evaluate HTTP adaptive streaming technologies as a means to address the on-demand expectations of their increasingly mobile customers, it is strikingly evident that the current state of technology lacks the level of control to which they have become accustomed. Our research seeks to answer the question of how can we provide monitoring and management capabilities for a large number of

independent OTT clients. We focus specifically on the question of how to enforce CoS-based bandwidth allocations in HTTP adaptive streaming clients.

The following chapters detail a series of techniques for intelligently applying video delivery pacing and bitrate adaptation in order to provide fair access to content in congested multi-client scenarios. The course of our research has followed the evolution of HTTP-based video delivery from single bitrate monolithic file download through multiple bitrate segment-based content retrieval to our current work in managed OTT video delivery. Each of these technologies builds upon the last as they seek to achieve functional parity with and ultimately exceed the capabilities of legacy video streaming protocols. It is our belief that HTTP adaptive streaming technologies, though based on fundamentally different assumptions than traditional RTP streaming techniques, provide a viable, efficient, and higher QoE approach for OTT video delivery.

Our initial work, discussed in Chapter 2, focused on server-side pacing in single bitrate progressive download applications, to improve server and data-center uplink scalability. Our follow-on work, discussed in Chapter 3, concentrated on client-side architectures for implementing data proxies, to enable rate adaptation in clients. As HTTP adaptive bitrate client support became more ubiquitous, our research refocused onto methods for providing network operators the ability to perform traffic management at the HTTP segment request level. Traditional low level traffic management techniques which are neither segment nor stream-aware can adversely impact QoE; migrating traffic management to the segment level can improve delivery continuity. Chapter 4 details a network proxy-based architecture for enforcing traffic policies on HTTP adaptive streaming sessions by overriding client rate selection decisions. The discussion then culminates in Chapter 5 with an evaluation of our segment-based rate adaptation algorithm that was applied in the client-side architecture described in Chapter 3, and which has been augmented to support class of service (CoS) differentiation and enforcement in distributed multi-client HTTP adaptive streaming environments.

Chapter 2

HTTP Progressive Download Server

One of the key components to any video delivery scheme is the pacing. Video is rendered at a constant rate (the video bitrate). Delivery of the video must be at least as fast as the video rendering rate. A delivery rate significantly greater than the video rendering rate does not necessarily benefit the client. An overly aggressive delivery rate can cause undue congestion in the network and may unnecessarily tax the resources of memory-limited clients. Typically, only the client and the server are aware of the video bitrate. As such, the network lacks the video bitrate information necessary to intelligently manage video delivery rates. Though we later discuss a network proxy-based approach to video delivery bandwidth management (see Chapter 4), in this Chapter, we first look at server-side pacing schemes for progressive download clients.

We developed an HTTP streaming server architecture, named Zippy, which uses paced delivery to distribute bandwidth usage over time, and to increase the scalability of the server, compared to a standard HTTP Web server. The Zippy architecture relies on a single pacer thread for managing individual session data transmissions, rather than using an individual process per connection, as with a default Apache prefork mode HTTP server. Figure 2-1 shows the difference between the Apache multi-process architecture and the Zippy single thread architecture. With Zippy, connection fairness between sessions is explicitly enforced by the session pacer, rather than the OS scheduler. Sessions are never preempted by the session pacer; sessions perform a limited amount of processing and then yield to the session

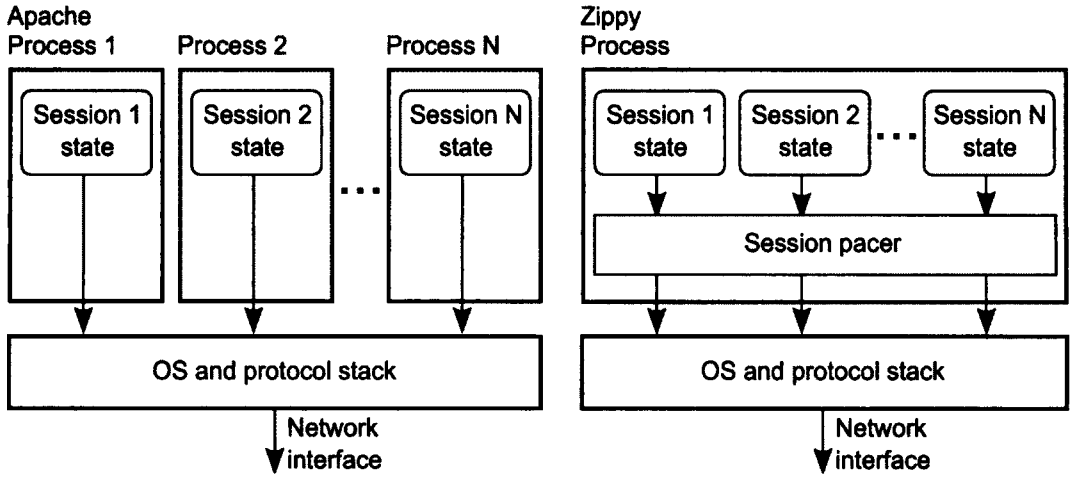


Figure 2-1: Comparison of Apache and Zippy process architectures.

pacer. The non-greedy nature of this scheme allows Zippy to support large numbers of concurrent long-lived connections.

2.1 Pacing Algorithm

Given C client sessions, the session pacer maintains a heap ordered by each session's next absolute send times $\{t_1, .., t_C\}$. Absolute send times t_i are recalculated after each send using the current wall clock time plus the session pacing delay, i.e., $t_i = T_{now} + \sigma_i$. Zippy calculates pacing delay using a fixed chunk size X and an assumed constant bit rate b_i, where $b_i = \frac{F_i}{D_i}$ is derived from the video file size F_i and video duration D_i. The pacing delay also makes adjustments for overhead delays ϵ:

$$\sigma_i = \begin{cases} \frac{X}{r_i} - \epsilon, & \epsilon < \frac{X}{r_i} \\ \sigma_{min}, & \epsilon \geq \frac{X}{r_i} \end{cases} \tag{2.1}$$

The overhead ϵ includes processing latency and *catch-up* delays (described below). If the overhead ϵ exceeds the pacing rate, a minimum delay σ_{min} is used to ensure session fairness. The fixed chunk size X is chosen as a fraction of the TCP window size to prevent overrun. The fixed chunk size is also used to manage session fairness; each session processes

a maximum of X bits of data and then yields to the session pacer.

C	total number of client sessions
X	pacing chunk size (in bits)
H	initial playback buffer threshold (in seconds)
σ_i	session pacing delay for video i (in seconds)
b_i	bitrate of current video i (in kbps)
D_i	duration of video i (in seconds)
F_i	file size of video i (in bits)
ϵ	processing latency plus catch-up delay (in seconds)

Table 2.1: HTTP streaming server pacing variables.

2.2 Intelligent Bursting

The Zippy HTTP streaming server implementation employs two intelligent bursting mechanisms: *initial playback* bursting and *catch-up* bursting. The former is used to decrease playback latency for media files; the latter is used to catch up sessions when network congestion has inhibited chunk sends. Bursting uses only excess system bandwidth, divided equally between all bursting sessions.

To combat jitter and prevent underrun, video file playback typically will not commence until a sufficient amount of video H (measured in seconds) has been buffered by the client. With paced output, the playback buffering latency is equal to the buffer size H. Playback latency negatively affects user experience, but can be avoided by bursting the initial portion of the media file. Assuming a network throughput of τ, the playback latency can be reduced to $H' = \frac{H*b_i}{\tau}$.

During periods of network congestion, full or near-full TCP windows may cause partial sends or send failures. In these situations, the actual number of bits sent $X' < X$, requires a delay $\sigma_i' < \sigma_i$. To prevent player underrun, future pacing delays are shortened to help sessions catch up. Though larger chunk sizes could be used for catch-up instead of shorter delays, if network congestion caused the failure, then the TCP window is likely limiting

how much data can be sent, making larger chunk sizes less effective. The catch-up delay ϵ cumulatively keeps track of missed send deadlines. Deadlines are considered missed when $T_{now} - t_i > \sigma_i'$.

2.3 Experimental Configuration

Zippy and Apache 2.2.6 were installed on a machine with a 2.2 GHz Intel® Core™2 Duo CPU and 2 GB RAM, running Fedora Core 8 linux (FC8). (The Apache version corresponds to the default FC8 httpd installation used.) To ensure that the test client was not a limiting factor, test client software was installed on a machine with dual 2.5 GHz quad core Intel® Xeon® CPUs and 4 GB RAM, running Red Hat Enterprise linux 5.1. The machines were connected through a Gigabit Ethernet LAN.

The tests were performed using 1 MB and 6 MB data files. A constant bit rate of 400 kbps was assumed, which results in file durations of 20 and 120 seconds, respectively. The client buffering requirement was assumed to be 4 seconds (or 200 KB). Client underrun checks were performed against the known constant bit rate, for each packet, after the initial buffer load. A 10 KB file was also used to simulate a typical Web-page object (e.g., an icon or HTML data file).

The test client is a multithreaded application which spawns a thread per connection. The connections were initiated as a flash crowd, with 500 microseconds between each request. Timestamps were recorded relative to the start of the test, as well as relative to the first TCP connection attempt. We examined the performance of 100 and 1000 concurrent connections.

2.4 Experimental Results

In our experiments, we compare the characteristics of our Zippy HTTP streaming server implementation with that of the de facto standard Apache HTTP server. From a client quality of experience (QoE) point of view, we use the initial playback latency metric to quantify the performance of each approach. We also look at the total download time

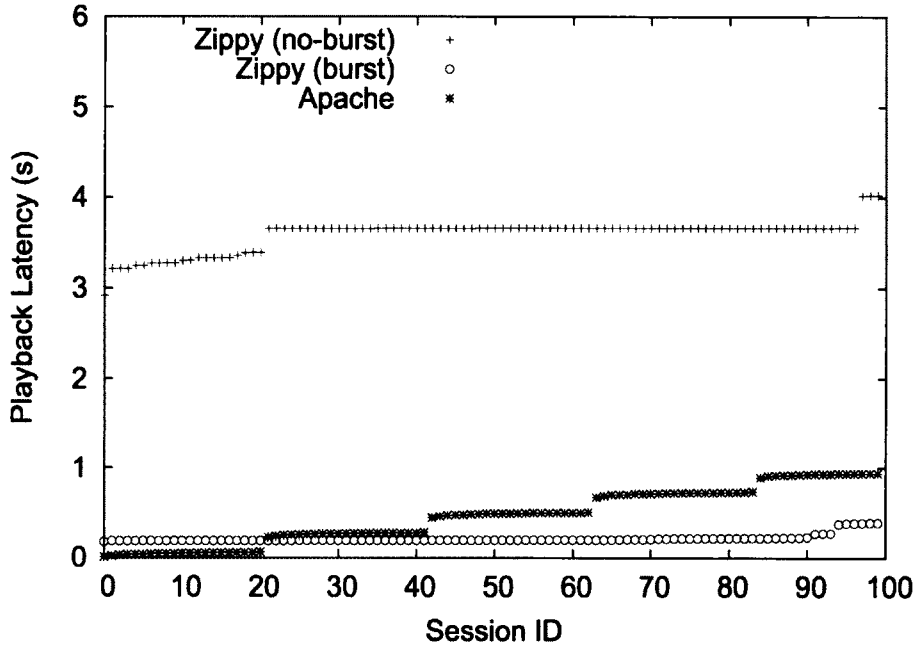


Figure 2-2: Playback latency for 100 concurrent sessions (1 MB File).

and server bandwidth usage, over time, to quantify the scalability of each approach. The following sections detail the results of our experiments.

2.4.1 Initial Playback Latency Comparison

We evaluate initial playback latency as the amount of time required to send enough data to fill the client buffer. Given our assumption of a 4 second buffer, streamed output without bursting should take less than 4 seconds to send the 200 KB. For a single straight download, over Gigabit Ethernet, 200 KB should take about 2 milliseconds plus overhead. Figures 2-2 and 2-3 show the initial playback latencies for each session retrieving the 1 MB file, in the 100 and 1000 session cases, respectively. The latencies are calculated as the offset from the first TCP connection request for each session and are sorted from low to high.

In Figure 2-2 the Zippy no-burst line, as expected, is consistently just below 4 seconds. The Zippy burst line shows a much lower latency, but with similar consistency across all sessions. The first 20 Apache connections are much faster than Zippy (burst or no-burst)

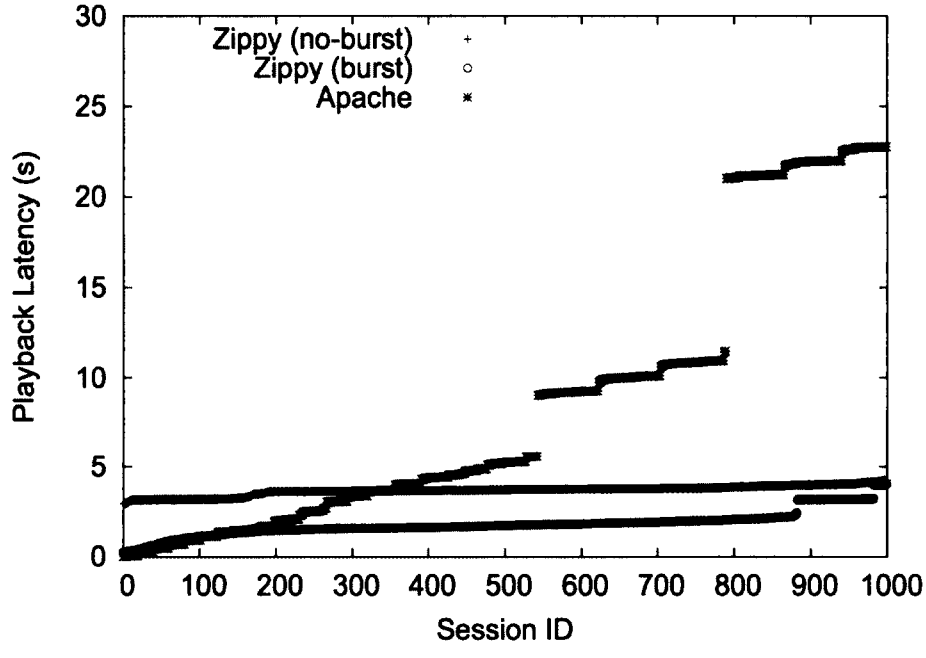


Figure 2-3: Playback latency for 1000 concurrent sessions (1 MB file).

and take about 60 milliseconds or $\sim$ 3 milliseconds per connection (close to the theoretical 2 milliseconds minimum download time, when taking overhead into account). A step function arises every 20 connections in the Apache plot. This correspond to the default maximum number of spare processes and represent the blocking latency of run-to-completion download.

In Figure 2-3 we can see that Apache performance gets progressively worse with 1000 sessions, compared to 100 sessions. The large vertical gaps in the Apache plot correspond to the points where Apache's blocking delays cause TCP timeouts and the TCP exponential backoff causes more significant latency penalties.

With 1000 sessions, the total bandwidth required goes up significantly, inhibiting Zippy's ability to burst. We can see this in Figure 2-3 as the playback latency for Zippy burst and Zippy no-burst converge for a small number of sessions. However, the worst case for both bursting and not bursting is still significantly better than Apache, for more than 60% of sessions.

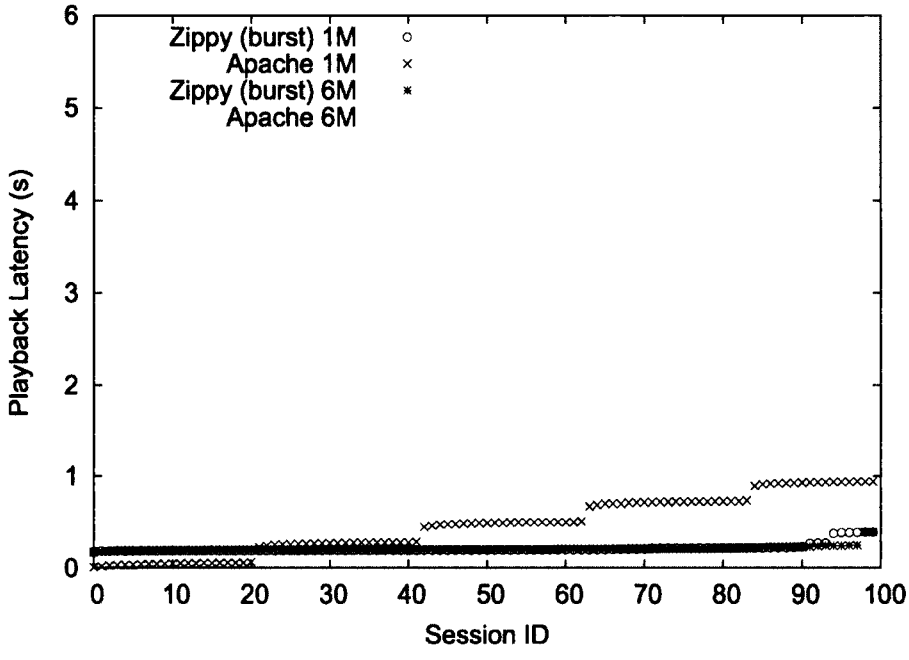


Figure 2-4: Playback latency for 100 concurrent sessions (1 MB vs. 6 MB file).

The 1 MB file is relatively small by modern video file standards. We also tested with a 6 MB file. Though the 6 MB file is also relatively small, our goal was only to compare the relative scalability of the Zippy and Apache approaches as file sizes grow. Even with this relatively modest increase in file size, a dramatic difference can be seen in the Apache performance. Figure Figures 2-4 and 2-5 show the initial playback latencies for each session retrieving the 6 MB file, in the 100 and 1000 session cases, respectively. As with Figures 2-2 and 2-3, the latencies are calculated as the offset from the first TCP connection request for each session and are sorted from low to high. For the Zippy plots, we only consider the case where intelligent bursting is enabled.

In both Figures 2-4 and 2-5, we can see that the Zippy plots for the 1 MB and the 6 MB files are almost identical. The paced Zippy architecture is not impacted by file size. Apache, however, is clearly affected by the larger file size. In Figure 2-4, we can see how the increased download time of the larger 6 MB file inhibits the playback latency of later sessions. In Figure 2-5, we can see how the increased download time also causes TCP timeouts to occur

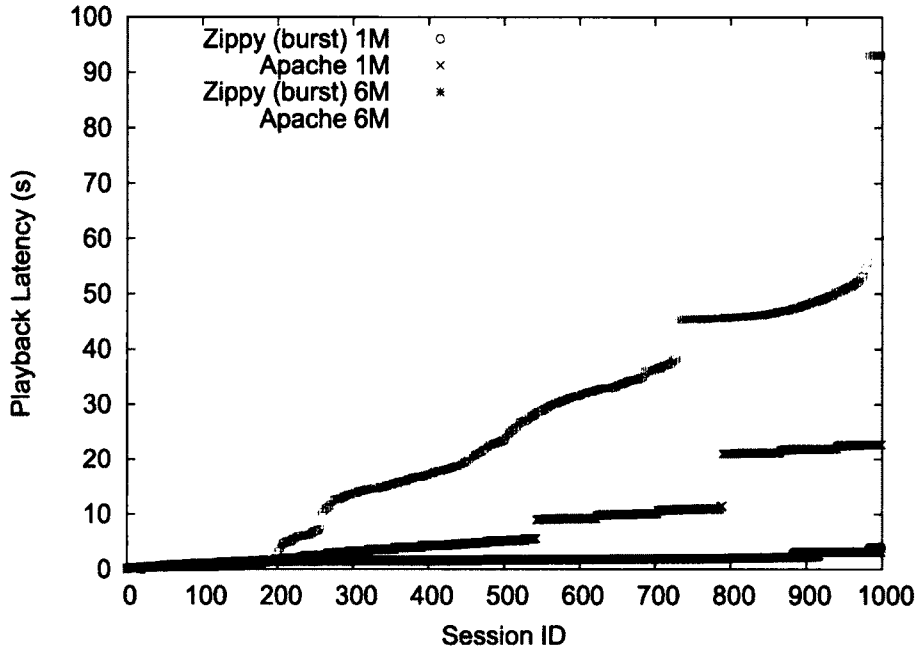


Figure 2-5: Playback latency for 1000 concurrent sessions (1 MB vs. 6 MB file).

sooner in the Apache 6 MB file case, which further impacts overall performance with the larger number of sessions.

2.4.2 Total Download Time Comparison

We evaluate the total download time for the file as the relative time between session initiation and file download completion. Given our assumptions of a 20 second file duration, streamed output without bursting should take less than 20 seconds from the time the HTTP connection is accepted. For a single straight download over Gigabit Ethernet, 1 MB should take about 10 milliseconds, plus overhead. Figures 2-6 and 2-7 show the download start and end times for each session retrieving the 1 MB file, in the 100 and 1000 session cases, respectively. The start times are offset from the start of the test, and are sorted from low to high.

In Figure 2-6 the Zippy no-burst line, as expected, is consistently just below 20 seconds. The Zippy burst line is consistently at about 16 seconds, which corresponds to the 20 second

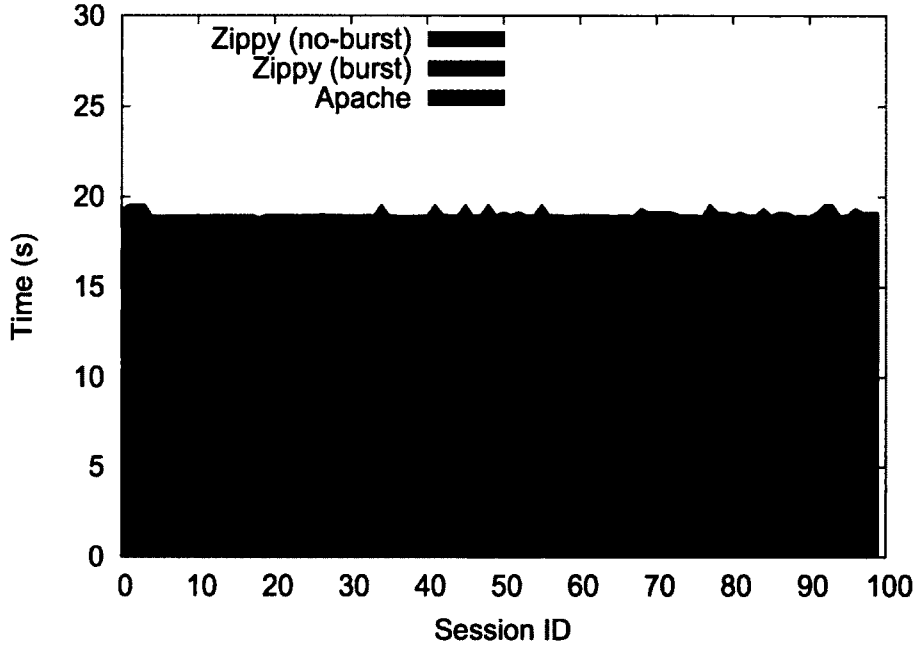


Figure 2-6: Download time for 100 concurrent sessions (1 MB file).

duration reduced by the 4 second burst. The Apache straight download times are dwarfed by the paced download times, and as expected, in the worst case, Apache takes a little more than one second to complete.

In Figure 2-7 we can see again that for 1000 sessions, Zippy performance is about the same, with paced delivery taking just less than 20 seconds with no bursting and around 16 seconds with bursting. The last 100 or so bursted sessions experienced TCP timeouts, causing their sessions to begin late, however, they still completed in around 16 seconds, relative to the start of the actual delivery. Though initially there was not enough excess bandwidth to accept the session, we can see that those sessions still met their playback deadlines and completed in less than 20 seconds from the start of the test. Apache, however, fares noticeably worse as the number of sessions increases. Due to the exponential backoff in TCP, Apache takes significantly longer to download the last 200 or so connections. Even though the total time to actually download is less, the user-perceived playback latency is quite high.

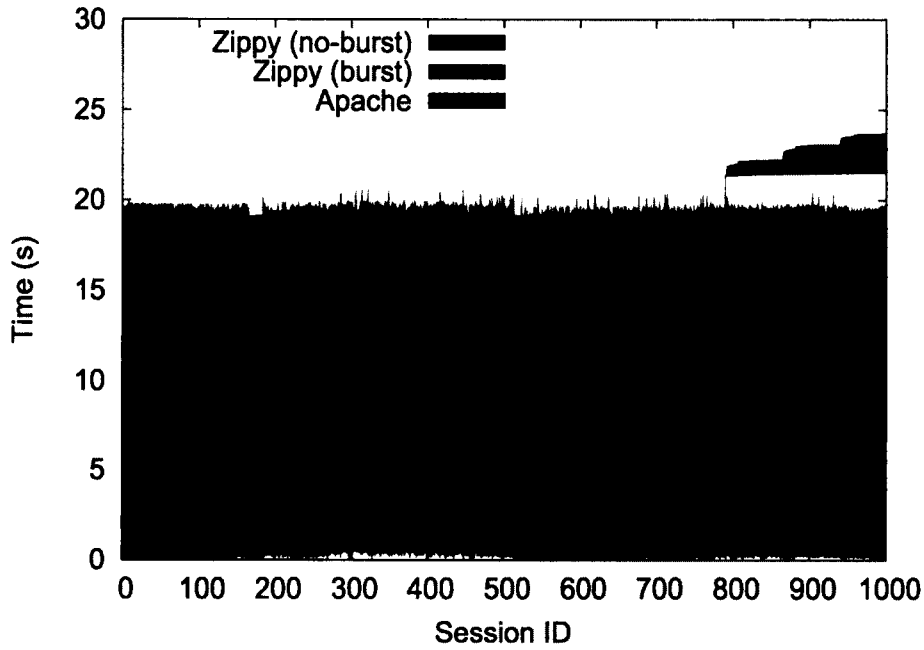


Figure 2-7: Download time for 1000 concurrent sessions (1 MB file).

For larger files, straight download latency gets worse, and more TCP timeouts occur. In Figure 2-8, we can see a comparison of the 1 MB and 6 MB file download times for Zippy and Apache. For the Zippy plots, we only consider the case where intelligent bursting is enabled. As expected, in the 6 MB file Zippy case, the total download time is just under 116 seconds. Looking at the Apache cases, we can see that the 6 MB file clearly takes much longer to download than the 1 MB file. Though the download time is much less than 120 seconds, The playback latency penalties are extremely high. In the worst case, TCP timeouts forced a latency of over 90 seconds before download started. Even though the downloads completed in 1 second, it does not negate the heavy impact to QoE for the viewer. With full length high definition video files on the order of gigabytes, the scalability issue is even more striking. Compounding this is the fact that many memory-limited clients are unable to buffer entire video files which causes TCP back pressure and exacerbates the blocking issues.

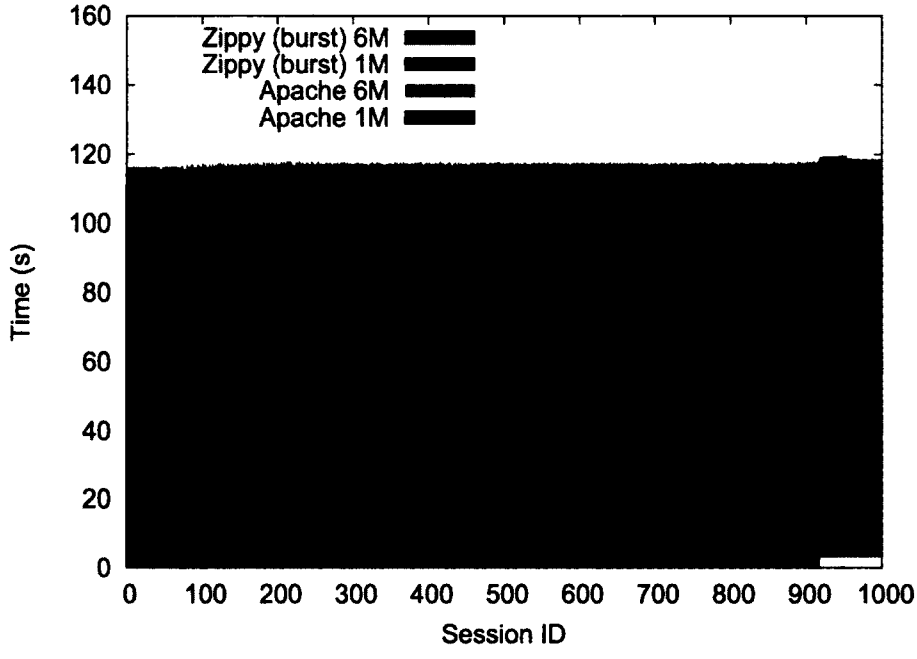


Figure 2-8: Download time for 1000 concurrent sessions (1 MB vs 6 MB file).

2.4.3 Server Bandwidth Usage Comparison

We evaluate aggregate server bandwidth usage using `tcpdump` traces and aggregating byte counts on a per-second basis. Given our assumptions of a 400 kbps video bitrate, streamed output without bursting should require 400 kbps per active connection. Figures 2-9 and 2-10 show the bandwidth consumed in the 100 and 1000 sessions cases, respectively.

In Figure 2-9 the Apache plot is near the practical usable capacity of the Gigabit Ethernet network and the OS protocol stack. The bandwidth usage is clustered within the first second of the test. This corresponds with the low total download times for Apache that were seen in Figure 2-6. The Zippy burst plot also has a marker close to the network limit, at the very beginning, which corresponds to the initial burst to all clients. The Zippy plots, in general have the same sinusoidal shape, however the Zippy burst plot is shifted to the left, in time, due to the burst. The end times for the Zippy burst and no-burst plots are at the expected 16 and 20 seconds, respectively, and the average bandwidth used, over the full 16/20 seconds test duration, is close to the expected 40 Mbps (i.e., 400 kbps $\times$ 100

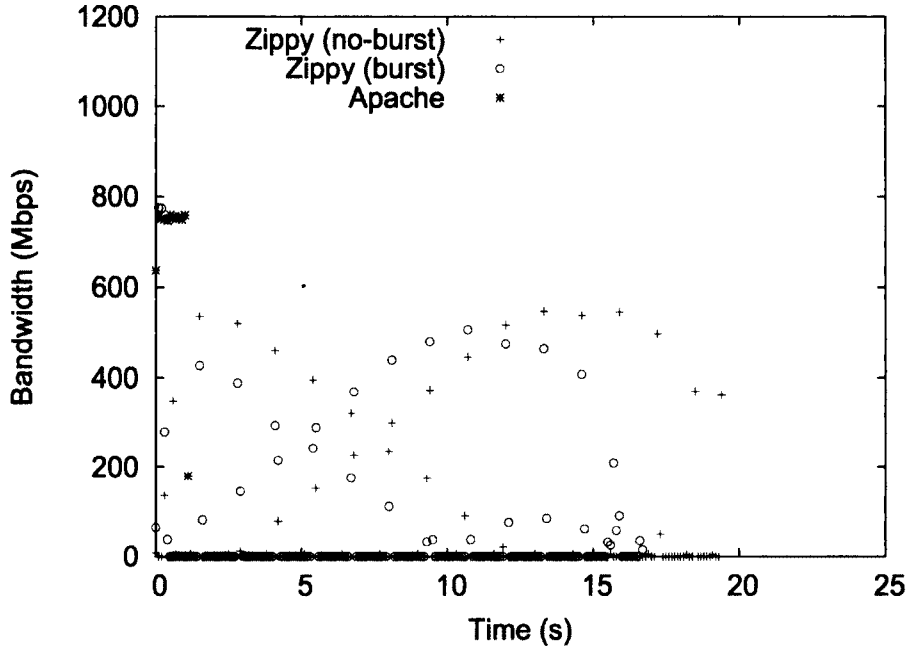


Figure 2-9: Bandwidth usage for 100 concurrent sessions (10 KB file).

sessions).

The cyclic nature of the Zippy plots is an artifact cause by the combination of data send clustering and the offsets between the sampling interval and the pacing interval. Data send clustering occurs when all sessions are initiated at the same time, as with our flash crowd scenario. This session synchronization manifests itself as bursty bandwidth usage. We can see that most of the time, the Zippy plots are registering zero bandwidth usage. The true average bandwidth is much lower than the peaks shown in the graph. The individual bursts are actually hitting the practical limits of the Gigabit Ethernet interface, however, due to the bursts crossing the 1 second sampling interval, the bursts are averaged over two sampling intervals. We can see that a first interval gets m Mbps and the following interval gets $m_{max} - m$ Mbps, where m_{max} is the practical maximum capacity of the Gigabit Ethernet interface. The fixed differential between the sampling interval and the pacing interval results in the "eye" pattern seen in Figure 2-9, which is similar to plotting $\sin(x)$ and $\cos(x)$ on top of each other.

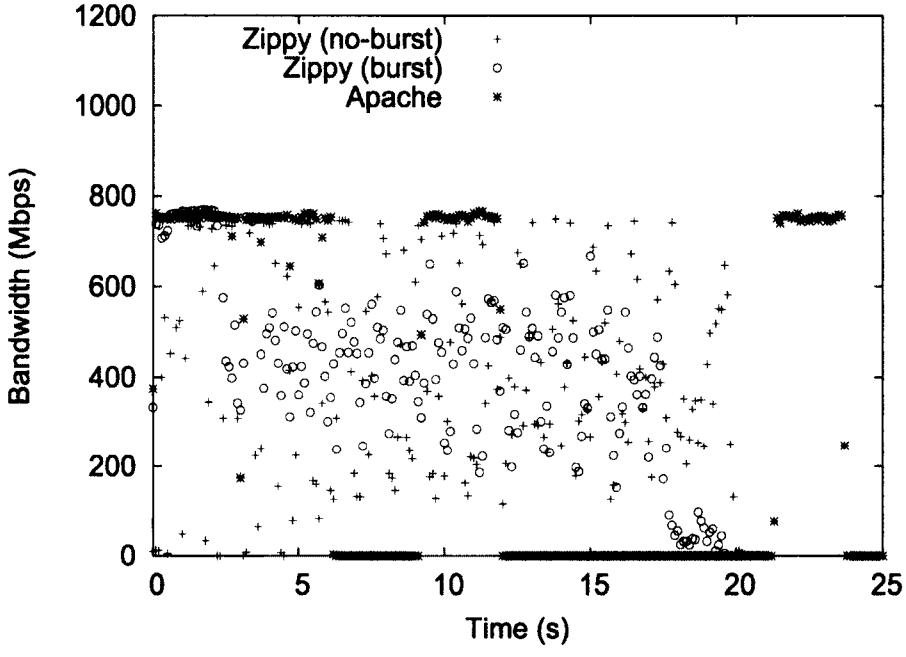


Figure 2-10: Bandwidth usage for 1000 concurrent sessions (10 KB file).

In Figure 2-10 the Apache plot is again always at maximum bandwidth, with gaps that correspond to the TCP backoffs shown in Figure 2-7. The Zippy plots no longer exhibit a sinusoidal shape due to the fact that the larger number of clients prevent any significant down time in transmission; there are very few zero bandwidth usage points. The Zippy burst plot shows a burst at the beginning and tails off at about 16 seconds. In between, the clustering of points is in the 400-500 Mbps range which corresponds to the expected 400 Mbps (i.e., 400 kbps $\times$ 1000 sessions). Though the Zippy no-burst plot is less consistent that the Zippy burst plot, both can be seen to be relatively evenly distributed.

2.4.4 Small File Download Comparison

While our Zippy HTTP streaming server was designed with large file delivery and long lived connections in mind, the Apache HTTP server was primarily intended for small file downloads. To evaluate the versatility of our Zippy approach, we compared the Zippy and Apache small file retrieval performance using a 10 KB file. Figures 2-11 and 2-12 show

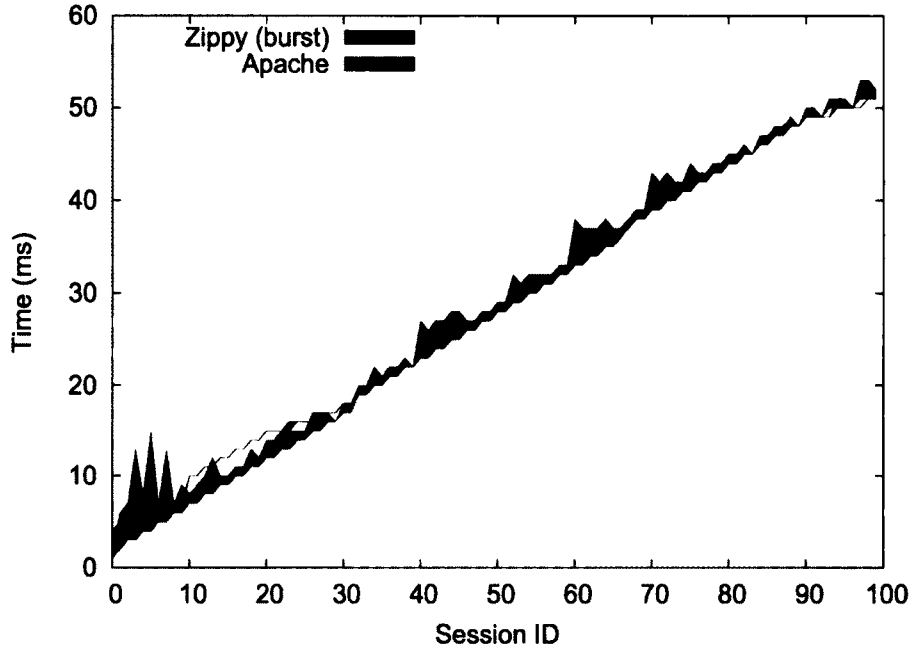


Figure 2-11: Small file download latency for 100 consecutive sessions.

the download times for the 100 and 1000 session cases, respectively. For a single straight download, over Gigabit Ethernet, 10 KB should only take 10 microseconds, plus overhead. This is significantly less than the 500 microsecond inter-request time, and therefore both servers should have ample opportunity to service the requests.

In Figure 2-11, we can see that the total time to download for Zippy is generally higher than that of Apache and exhibits higher variation in download time. This is due to the σ_{min} delay value that is used by Zippy to ensure inter-session fairness. Though none of the sessions should be overlapping in this scenario, given that the inter-request time is an order of magnitude larger than the theoretical download time, the Zippy server must still check to see if any concurrent sessions require servicing. As long as the file is smaller than the fixed chunk size X, the latency penalty should be minimal.

Even with the small additional latency introduced by the concurrency checks, Zippy, in general, is able to keep pace with Apache, for small file downloads. In Figure 2-12, we can see that as we scale the number of session out to 1000, there is no real discernible difference

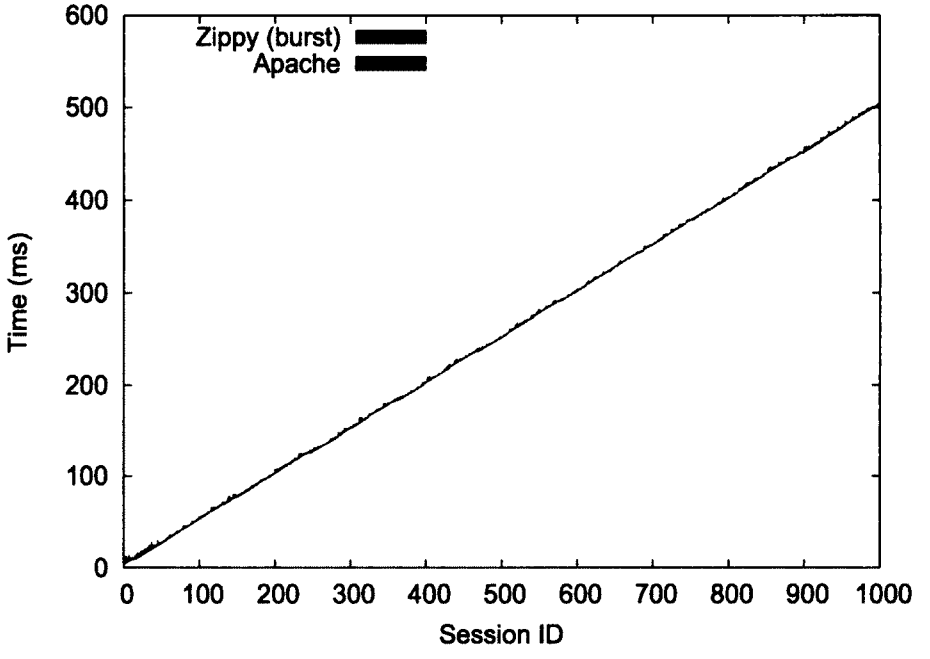


Figure 2-12: Small file download latency for 1000 consecutive sessions.

in performance between Zippy and Apache.

2.5 Summary of Server-side Pacing Contributions

Prior to the advent of HTTP adaptive streaming, we investigated HTTP streaming. In the early days of HTTP-based video delivery, schemes were primarily categorized as either "download and play" or "progressive download", though neither of those phrases adequately described the schemes which they were classifying. "Download and play" typically referred to a two-step process of: (a) download the entire video as fast as possible, then (b) play the video. "Progressive download" most often referred to a parallel process of: (a) download entire video as fast as possible, but (b) start playing as soon as enough of the video has been downloaded to fill the player buffer. Though the term "progressive" could be interpreted to mean "paced", in general it did not. HTTP streaming generally referred to a server-side paced delivery mechanism. The Microsoft Windows Media HTTP Streaming Protocol [94]

was a full featured HTTP-based video streaming protocol which rivaled RTSP, but it was proprietary, requiring the Microsoft Windows Media Player® client. Though HTTP was being successfully used for video delivery, there was still a stigma associated with using HTTP for video delivery. Even though HTTP was designed with smaller files in mind, it has been adapted as the de facto standard transport for many types of content, including video.

The main contribution of our Zippy HTTP streaming server was the creation of an HTTP streaming (i.e., paced delivery) server which could work with any HTTP-based media player. We used the basic Zippy server as a platform to investigate server scalability issues, as well as the behaviors of multiple different HTTP media players. We were also able to use our platform to evaluate media player reactions to different pacing schemes, and from that, we were able to confirm the functionality of our intelligent bursting scheme using the Microsoft Windows Media Player® and Apple QuickTime® player. The second contribution of our Zippy HTTP streaming server was our comparison of the architectural differences between a streaming server, designed for long lived connections, with the standard Apache server architecture, designed for small file delivery. With an understanding of the trade-offs and considerations involved in the design of each architecture, we were able to design an HTTP server that generically addresses other high latency transactions, e.g., transparent proxy connections. The third contribution of our Zippy HTTP streaming server is a more long term understanding of application-level pacing. The Zippy HTTP streaming server has been instrumental in the evolution of the bitrate selection override proxy, described in Chapter 4, and the open questions relating to individual segment pacing in HTTP adaptive streaming delivery, as described in Section 5.8. Beyond the basic paced delivery investigation, our work with the Zippy concepts continues to provide insights and inspiration for future research.

Chapter 3

HTTP Adaptive Bitrate Client

In the previous chapter, we discussed a server-side pacing scheme for progressive download clients. Prior to the release of Apple iOS 3.0, most commercial media players relied on pre-download or progressive download of content and did not support bitrate adaptation. The HLS implementation released with the Apple iPhone® began a movement to develop new client-side rate adaptation technologies. In this Chapter, we investigate methods for implementing rate adaptation in clients which may not natively support an adaptive streaming protocol. Specifically, we discuss the use of stitched media files for pre-download and progressive download media players, as well as, RTP segment files for RTSP and other frame-based media players.

Though many modern devices natively support HTTP adaptive streaming protocols (e.g., HLS or Smooth Streaming), this is a relatively recent development. There are a large number of legacy devices and platforms which do not fully support HTTP adaptive streaming natively (e.g., Android™ 1.x and 2.x, and Blackberry® 4.x and 5.x). Many of those devices do support other network-based video streaming options. For those devices, we developed a client-side proxy architecture for implementing an HTTP adaptive streaming-like approach. Protocol support for the non-HTTP adaptive streaming platforms fall into four general categories:

- HTTP progressive download,
- local file data source,
- frame-based data source, or

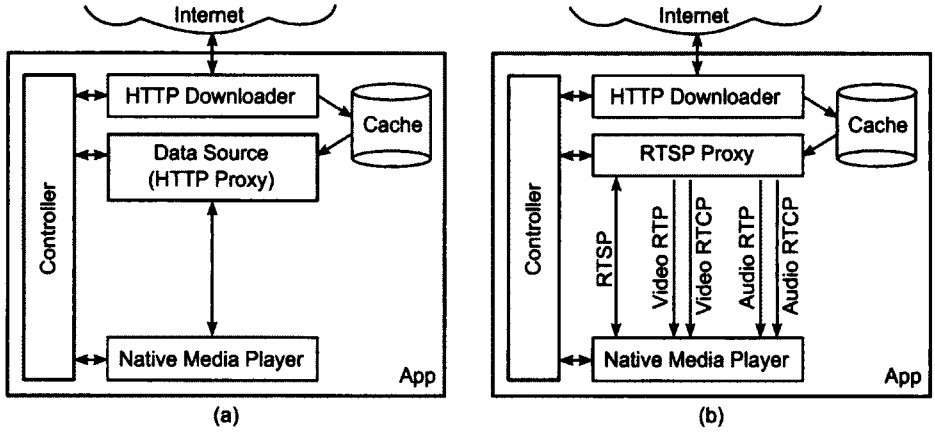


Figure 3-1: Client-side rate adaptation proxy architectures.

- RTSP streaming.

Figures 3-1 (a) and (b) diagram the general architectures for the client-side proxy approaches. Figure 3-1 (a) corresponds to the HTTP progressive download, local file, and frame-based data source cases, while Figure 3-1 (b) corresponds to the RTSP streaming case. Both architectures rely on an HTTP downloader module which retrieves segments and stores them in a local cache. They also both have a controller which coordinates the initiation of HTTP downloads and the starting of the native media player. The controller is also responsible for dynamically selecting the bitrate and instructing the HTTP downloader to retrieve data for the selected bitrate. The controller uses multiple inputs to select the appropriate bitrate, including: available bandwidth estimates, based on the amount of time it takes the HTTP downloader to retrieve data, the current cache occupancy level, representing the data already downloaded but not yet delivered to the native media player, as well as playback control information, e.g., is the player paused or stopped, fast forwarding or rewinding, just beginning or nearing the end, etc. Rate selection is discussed in greater detail in Chapter 5. The primary difference between Figures 3-1 (a) and (b) are contained in the client-side proxy implementations.

In the local file data source case, the native media player accesses the proxy via standard file IO APIs, issuing byte range requests. The HTTP progressive download case works sim-

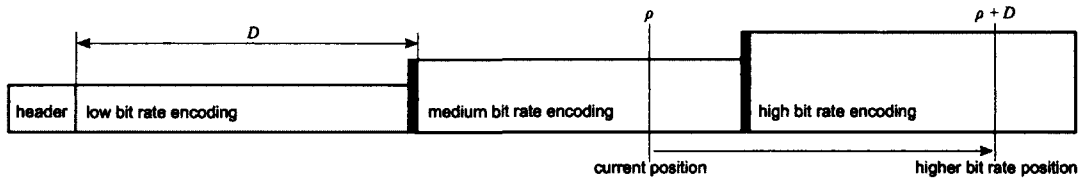


Figure 3-2: Stitched media file (three bitrates): rate adaptation seek.

ilarly, only the proxy is implemented as an HTTP server listening on the localhost address. The HTTP proxy works with Range GETs, however, for straight download requests, the HTTP proxy implements paced delivery, as described in Chapter 2. Data source-based rate adaptation relies on stitched media files, which are described further in Section 3.1.

In the RTSP streaming case, an RTSP proxy server listening on the localhost address accepts connections from the native media player and implements paced frame-based delivery. RTP data is collected and packed into segments which are retrieved by the HTTP downloader. The frame-based representation of RTP segment data can also be used in frame-based data source implementations (e.g., with Microsoft® Silverlight™). RTP segment creation is also described further in Section 3.4.

Because of their reliance on stitched media files, the local file data source and HTTP proxy approaches are only applicable to VoD. The RTSP proxy and frame-based data source approaches can be used for either VoD or live streams. Though some platforms can support either a local file data source/HTTP proxy or RTSP proxy, only one is needed. The RTSP proxy approach, though more versatile, is also more complex and consumes more resources. Older devices which do not require live streaming support often benefit from the simpler local data source or HTTP proxy approach.

3.1 Stitched Media Files

Figure 3-2 shows an example stitched media files. It takes multiple pre-transcoded files of different bitrates and concatenates them together into a single file, with padding inserted between each encoding. The stitched media file is a standard 3gp file. The header con-

tains standard 3gp header elements describing the audio/video frame data contained in the stitched media file and includes information for all encodings. A source video is transcoded into B different bitrate encodings $\{b_0, \ldots, b_{B-1}\}$, each with a padded duration D. The starting position within the stitched media file for any given encoding is $A(b_i) = D \cdot i$. Encodings are stored in a known order, e.g., low-to-high by bitrate, as in Figure 3-2. The total duration of the stitched media file is $B \cdot D$.

B	number of discrete bitrates
D	padded duration of the video (in seconds)
ρ	current playout position (in seconds)
ε	key frame offset variation (in seconds)

Table 3.1: Stitched media file rate switch variables.

Rate switching is achieved by issuing a seek request to the media player with an offset of $\pm D$. For a high-to-low stitched media file, adding D selects the next higher bitrate, while subtracting D selects the next lower bitrate. For a given playout position ρ, Figure 3-2 shows the $\rho \pm D$ offset from the current bitrate (medium) into the next higher or lower bitrates (high and low, respectively). Switching by a single bitrate minimizes viewing discontinuity, but skipping multiple bitrates is also possible with an offset of $\pm(n \cdot D)$. Seeking to a key-frame boundary is also important to minimize distortion. For encodings where key-frame boundaries do not align, the offset should account for error in locating the key-frame nearest the switch point: $\pm(n \cdot D) - \varepsilon$. If the stitched media file data for the target of the seek has not yet been downloaded, a future playout position ρ' should be calculated for when the seek should be issued, taking into account the retrieval time for the data: $\rho' = \rho + RTT + R/T$, where RTT is the network round trip latency, R is the size of the data to be retrieved, and T is the throughput of the network.

The stitched media file approach minimizes rate adaptation latency, but incurs a startup penalty for downloading the extra headers. initial playback latency can often be hidden by the application (e.g., through the use of an advertisement or splash screen), whereas

rate adaptation latency is always noticeable to viewers. For content providers, rate switch continuity is an important aspect of viewing experience. The scheme is not suitable for live video, given the need to prestitch the files, but works well for video on demand (VoD) applications. Some devices with very limited memory may not support processing large header, however, the majority of modern devices support feature length 3gp files.

3.2 Stitched Media Player

We implemented stitched media player libraries for the BlackBerry® and Google Android™ platforms, using the architecture shown in Figure 3-1 (a). It takes advantage of the native media player which retrieves data from our local data proxy. The data proxy pre-fetches and caches portions of the stitched media file (pseudo-segments). Cached data is retrieved by the HTTP downloader using platform specific APIs to issue HTTP Range GETs. Range GETs allow the client to use pseudo-segmentation to pace content retrieval. Though the content is not physically divided into file segments (as with HLS), fixed data ranges effectively define pseudo-segment boundaries, where the pseudo-segment boundaries are measured in bytes, rather than seconds. The HTTP downloader measures download bandwidth and reports to the controller which initiates rate switches.

An index file is used to optimize the nearest key-frame selection process (i.e., resolving ε) and for converting from the time domain into the byte domain for fast file retrieval. The index file is created as part of the pre-processing step when concatenating the individual bitrate files together into the stitched media file. The overhead for stitched media file pre-processing is comparable to that of RTSP hinting and HLS segmentation and manifest generation. The storage space required is also comparable, however, segment-based schemes use many more files. Large numbers of files can often lead to performance penalties in storage systems with directory size limits. The HTTP downloader may either download the index file when playback begins, or access it dynamically when a rate switch is ordered by the controller. The HTTP downloader uses the index information to generate its HTTP Range requests.

When the controller wishes to change rates, it instructs the HTTP downloader to begin pre-fetching the new bitrate data and checks how much data is available in the data proxy. Given P seconds worth of data in the data proxy, a timer is set for $P - u$ seconds, where u is an underrun prevention factor (on the order of seconds). The playout of the cached data provides time for pre-fetching the new bitrate data. When switching to a lower rate, playing out the cached data maximizes higher quality rendering. When switching to a higher rate, u should be optimized such that the rate switch occurs as soon as possible while minimizing low quality rendering. Upon timer expiration, the controller checks the current position ρ and calculates the time offset $\rho \pm (n \cdot D) - \varepsilon$ for the nearest key-frame in the new encoding, and issues a seek command to the native media player. Our implementation employs an index file mapping ρ to byte and time offsets which take ε into account.

3.2.1 Rate Adaptation Latency

The advantage of the stitched media file approach in rate adaptation latency comes from the fact that seek operations are faster than reinitializing the media player. For media players that do not natively support segment-based parsing and rendering continuity, the player reinitialization required for each segment, regardless of segment size, is significantly higher than a seek operation, as shown in Figure 3-3. As mentioned previously, however, our stitched media file scheme does incur an initial playback latency penalty, as shown in Figure 3-4, given the larger header size.

For both sets of test, we used a BlackBerry® Bold 9000 on AT&T and a Motorola Droid™on Verizon. Both platforms provide sample applications for playing videos which we modified to measure player restart latency and player seek latency. We transcoded an 80 minute source video into two bitrates: 126 kbps and 286 kbps (H.264, 24 fps), both with 64 kbps (AAC-LC, mono, 22050 Hz) audio. We extracted 10, 60, 300, 1800, and 3600 second clips for each bitrate to simulate a variety of different segment sizes for the rate adaptation latency tests in Figure 3-3. We also stitched 126 kbps and 286 kbps clips together for each duration, to simulate a variety of different stitched media file header sizes for the initial

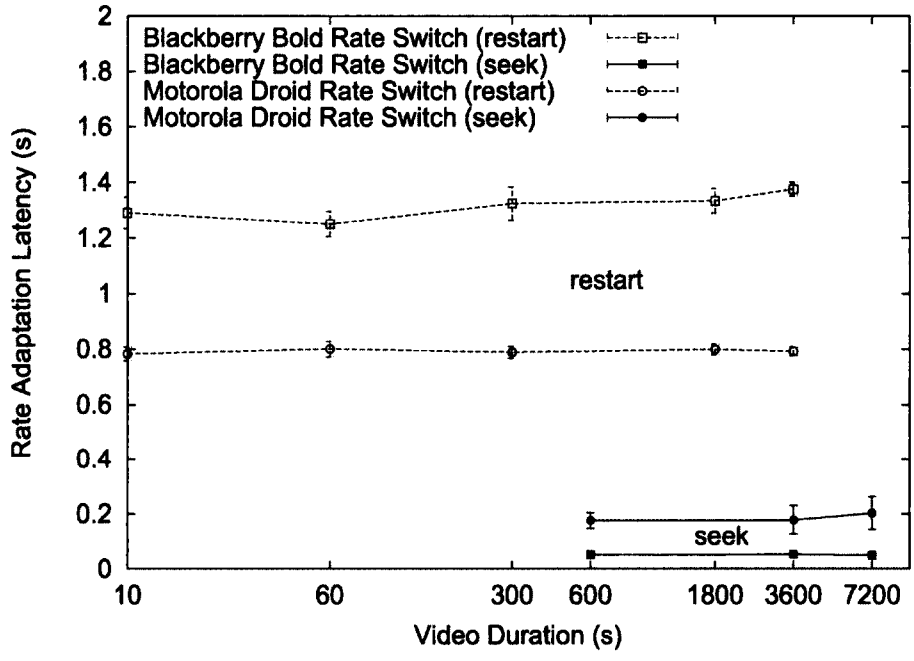


Figure 3-3: Stitched media file rate adaptation latency.

playback latency tests in Figure 3-4.

We can see from Figure 3-3 the impact of flushing hardware buffers and reloading header information, when switching between files, even for small 10 second files as are used in segment-based rate adaptation schemes. Seeking within a stitched media file with two bitrates is significantly faster than restarting the player, even for large files. We only provide seek values for longer clips, as the need for rate adaptation in short clips is negligible. Seek times are shown for the total duration of the 2 bitrate stitched media file (e.g., 7200 seconds for a two bitrate 3600 second clip). Player restart times were measured by playing the 60 second, 126 kbps file, then destroying and creating a new player to play the 256 kbps clip at the specified duration. Player seek times were measured by playing the 1 hour stitched media file at 126 kbps for 60 seconds, then seeking to the 286 kbps offset. For this test, all clips were played from a local micro SD card to remove network variability. Each plot is an average of 8 runs with standard deviations shown. Figure 3-3 clearly illustrates two main points: clip duration is not a significant factor in player restart latency and seek latency is

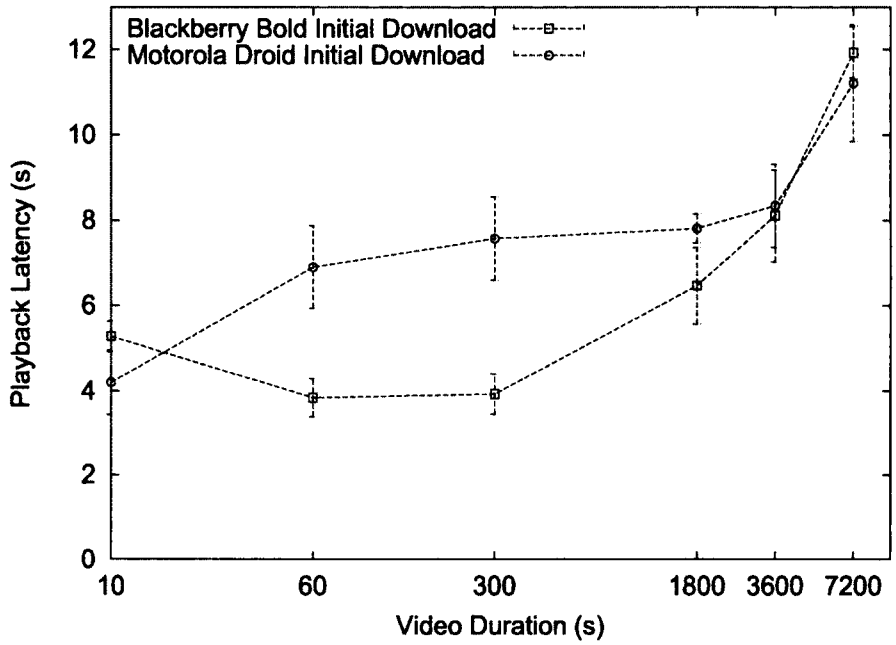


Figure 3-4: Stitched media file initial playback latency.

significantly lower than restart latency.

In Figure 3-4, we show the initial playback latency for each of the 286 kbps non-stitched clips, as well as the 1 hour stitched media file (shown as a 7200 second total duration file). For this test, all clips were played from the Akamai^{TM} CDN. Each plot is an average of 8 runs with standard deviations shown. As expected, longer duration files with more header data have higher IPL, except in the case of the BlackBerry® Bold 10 second clip, where download completion triggers garbage collection which inhibits playback. Cellular bandwidth varies greatly, even in off-peak hours when our tests were run (between 2 and 5 AM), but for chapterized playback with ads inserted between chapters, the initial playback latency is typically maskable.

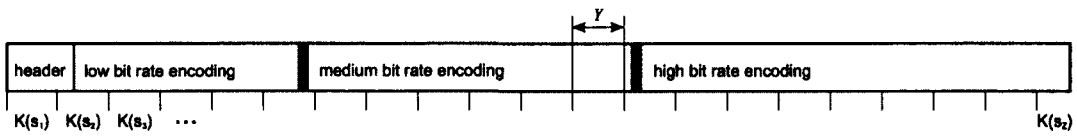


Figure 3-5: Stitched media file (three bitrates): cipher block overlay.

3.3 Stitched Media File Encryption

Our stitched media file DRM encryption scheme uses a stream cipher-based approach, i.e., a pseudo-random stream of bits is generated and used to encrypt the video data using an XOR operation. We refer to the pseudo-random stream of bits as a *key*. A key $K(s_i)$ is generated using a cipher initialized with a seed s_i. Though a single key could be used to encrypt an entire stitched media file, the need to support seek operations necessitates a more flexible approach. A single key must be generated sequentially. Random accesses like seek operations require generating the entire key sequence, up to the seek target. This can result in the needless generation of a large amount of key data. To limit this unnecessary key generation, we use a block-based approach, enforcing a key length limit Y for each stitched media file.

Z	number of encryption blocks
Y	encryption block size (in bits)
F	total stitched media file size (in bits)
s_i	seed value for the i^{th} block
$K(s_i)$	encryption key for the i^{th} block

Table 3.2: Stitched media file encryption variables.

Figure 3-5 shows a stitched media file overlaid with Z key blocks $\{K(s_1), \ldots, K(s_Z)\}$, each of size Y bytes. The number of blocks $Z = \lceil F/Y \rceil$, where F is the total size of the stitched media file. Assuming seek locations are randomly distributed, on average $Y/2$ bytes of key must be discarded in order to begin decrypting at the seek location. When varying

stream cipher key length, there is a trade-off between initialization overhead for the cipher and unnecessary key generation.

When using this block-based approach, managing multiple seeds must be considered. The root seed s_0 is typically combined with an initialization vector (IV) to generate the per block seeds. The IV usually takes into account the block number i, similar to the way the SRTP sequence number is used. In the case of Figure 3-5, $s_i = s'(s_0, i) : 0 < i \leq Z$, where the function $s'(seed, index)$ may be a simple hash or another stream cipher; the acquisition of the root seed s_0 is the primary security concern. We use a standard scheme involving: a client registration process through which device specific shared keys are securely registered with the DRM server using shared key encryption, and root seeds being encrypted with the device specific shared key prior to being transmitted to the device. For the purposes of this discussion, we assume that the time to generate the seeds is negligible and that standard key exchange practices are secure.

3.3.1 Pseudo-segment Download and Decryption

The primary concern when performing rate adaptation must be for rendering continuity. To make sure there are no playback interruptions, the data for the new bitrate must be downloaded while the current bitrate continues to download and render. In a multi-threaded environment, the timing interactions between downloading the video, decrypting the video, rendering the video, and other ancillary background tasks can be complex to model. This is compounded during a rate switch, when both the current and the new bitrates must be downloaded and decrypted to allow for a seamless rate transition.

To simplify our discussion, we discretize the download, decrypt, and render operations so that we can highlight the timing interactions and the effects of network throughput and cipher overhead. Figure 3-6 shows a hypothetical, discretized timing diagram for a video playing a medium bitrate encoding at time t and switching to a high bitrate encoding at time $t + 5$. Within each discrete time quanta, we group the download and decryption activities for each bitrate to show the network and CPU margins. The arrows track the

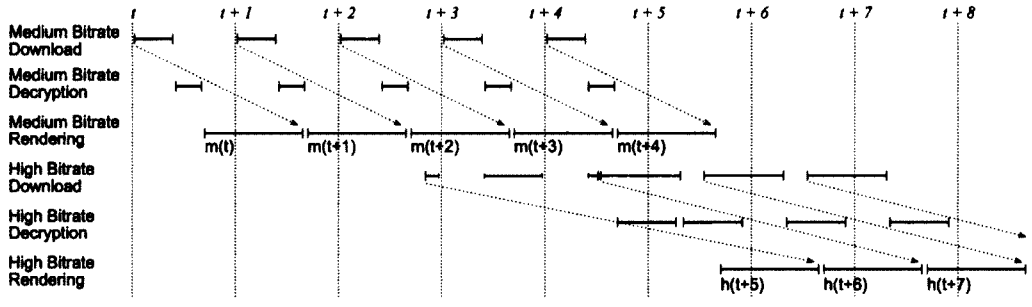


Figure 3-6: Discretized timing diagram for downloading, decrypting, and rendering.

total processing for a single time quanta's worth of video.

In Figure 3-6, each time quantum is a discretized playout duration. In this example, we assume that the network throughput is fairly high, allowing for a rate switch to a higher bitrate. (We can see that the download time is less than half the time quantum.) Once the data has been downloaded, it must be decrypted before it may be rendered. The decryption time must also be less than the time quantum. Though decryption may be pipelined with download (i.e., the current decryption may overlap with the next download), decryption is CPU intensive whereas download is not. CPU is also consumed for software-based video decoding and rendering. Decryption time must be kept low, to prevent interfering with playback continuity, especially as video quality continues to increase.

Given the excess bandwidth in this example, a rate switch to a higher bitrate is initiated between time $t + 2$ and $t + 3$. When high bitrate download begins it must be interleaved with continuing medium bitrate download. The download margin, i.e., the amount of excess network capacity, is the primary component in rate adaptation latency. Download of the new bitrate must only use excess bandwidth, to prevent playback interruption. Similarly, decryption must only use excess CPU time. Calculation of CPU margin is complicated, however, by the different device capabilities, different media player software, and variety of background tasks both pre-installed by the device manufacturer and downloaded by the user. It must also be considered that higher bitrate data will take more time to download and more time to decrypt, than the lower bitrate data. And because the data may span

multiple key blocks, the decryption time for a single time quantum's worth of data may be disproportionately larger for higher bitrates, due to additional stream cipher initializations.

We can see in Figure 3-6 that even though the rate switch is initiated during $\rho = t + 2$, two additional full time quanta are consumed to download the higher bitrate content, while continuing to download the medium bitrate content. The rate switch does not actually occur until $\rho' = t + 5$. In an actual implementation, the last three blocks of medium bitrate data might be downloaded in a single burst and then the high bitrate data would be downloaded, rather than interleaving the two downloads, but the total time to download would still be the same. The example shown does not contain any decryption contention, only download contention. In reality, both may impact the $\rho' - \rho$ rate adaptation latency.

3.3.2 Cipher Performance

We considered three popular stream ciphers: AES128 CTR, HC128, and RC4. Our primary concern was not the relative security of these protocols, but rather the relative performance. With limited CPU resources on mobile devices, the trade-off between security and performance becomes more important. Each of the stream ciphers has a unique performance profile. AES128 CTR is computationally expensive for generating keys. It uses the AES128 block cipher algorithm to generate 128 bits of key data at a time, based on the seed, IV, and counter. The counter is incremented and the process repeats until the entire key length is achieved. HC128, on the other hand, has a lower computational cost, but high initialization overhead for setting up the lookup tables used by the key generation algorithm. The key is generated 32 bits at a time by looking up a value in one of the tables and then modifying that table entry for future use. This is repeated until the entire key length is satisfied. Both AES128 CTR and HC128 take a 128 bit seed value and a 128 bit IV as inputs. Though RC4 does not have the inherent concept of an IV, we implemented a hardened version which combines a 128 bit seed with a 128 bit IV to initialize the key generator. RC4 keys are generated one byte at a time by selecting a byte from a bag and then mixing the bag before selecting the next byte. Our hardened version of RC4 also discards the first 3072 bytes of

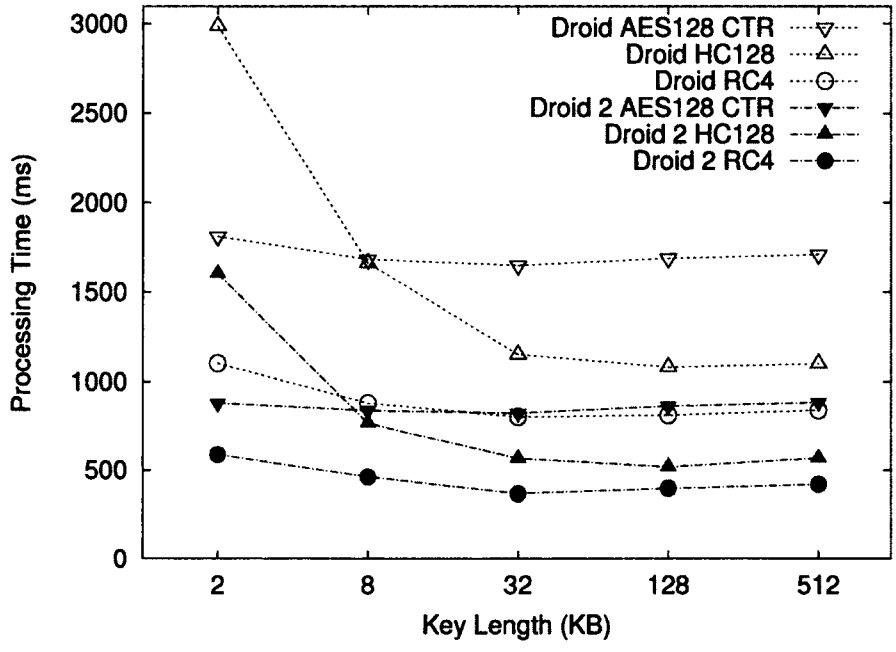


Figure 3-7: Stream cipher performance comparison.

key generated to mitigate well-known bit correlation issues in the algorithm. This adds to the key generation overhead.

Figure 3-7 shows a comparison of the three stream ciphers, implemented in Java for the Android™ platform, and run on two different devices: the Motorola Droid™ and the Motorola Droid™ 2. The results show the time required to decrypt a 1 MB file, using a given cipher and key length. The 1 MB file size roughly corresponds to 10 seconds of 800 kbps video. Though 800 kbps is at the high end for smart phones, newer tablet devices often use twice that bitrate. When we consider the decryption CPU margin, a 10 second time quantum is fairly typical. (10 seconds is the recommended value for HLS segment duration.) Rendering and other background operating system tasks often consume significant CPU resources which must take priority over download and decryptions functions. We can see from Figure 3-7 that some of the cipher configurations can take on the order of seconds to decrypt 10 seconds worth of data. This leaves little margin for error in preventing the mobile CPU from being overwhelmed.

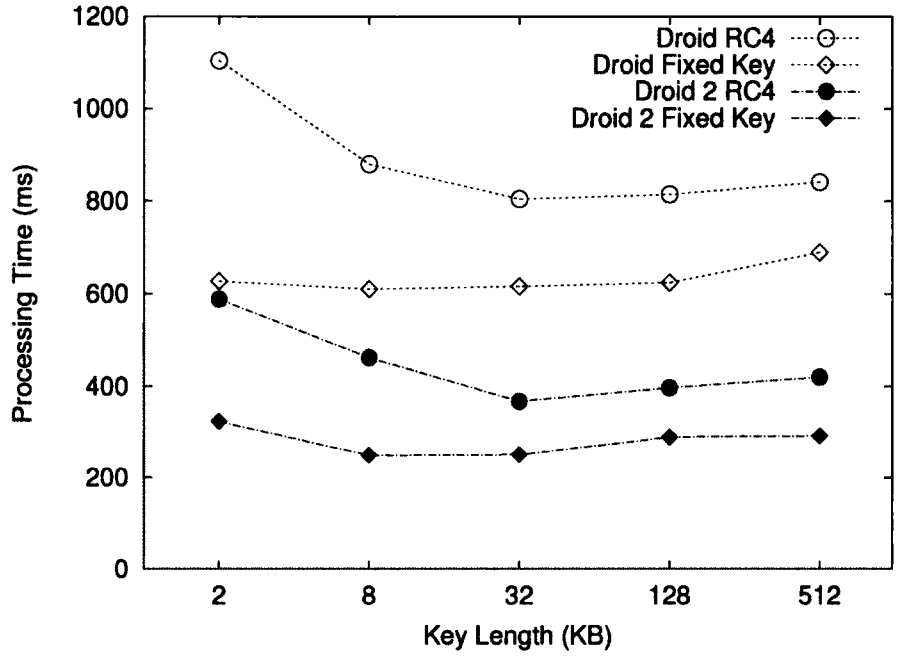


Figure 3-8: Fixed key cipher performance comparison.

We tested exponentially increasing key lengths and plotted them on a log scale. Each result is the average of eight runs. The initialization overhead of HC128 is clearly a penalty at very small key lengths. AES128 CTR and RC4 however, with their fixed algorithms are more consistent across key lengths. Also very apparent is the performance advantage of RC4, which does not use cryptographic transforms, over the computationally intensive AES128 CTR algorithm. Though key length does not have significant impact on processing time for longer keys, longer keys are not necessarily better. Longer keys increase unnecessary key generation in seek operations.

In addition to the three stream ciphers, we also tested a static, pre-generated key. With a static key, the same fixed sequence of bytes is simply repeated, rather than generating random bytes. Figure 3-8 shows a comparison between static keys and RC4 (the best performing cipher from Figure 3-7). We again tested a Java implementation on the Motorola Droid™ and the Motorola Droid™ 2 Android™ platforms, decrypting a 1 MB file. As expected, the static key performs better. There is low overhead at small key lengths, though

the overhead increases as the key length increases. This is due to the initialization time required to read the key into memory. Larger static keys also require more memory to store them, and can consume significant bandwidth for remote key retrieval. The disadvantage of static keys, however, is their insecurity. The repeating value is much more susceptible to brute force attacks than a pseudo-random byte stream. In general, the static key has minimal security value, however it does provides a lower bound for encryption performance.

The other key observation from Figures 3-7 and 3-8 is the performance disparity between the first generation Droid and the Droid 2. Mobile device hardware capabilities are constantly and rapidly improving, but so too is the demand for higher quality video and more secure DRM. In choosing a stream cipher, we have limited the scope of the problem by removing download time from the equation. For CBC algorithms which require data feedback, the download time can add significant latency to the decryption process, especially after a seek. With no download requirement, keys may be pre-generated as long as sufficient memory exists. Selection of a sufficiently large key size, however, is important for encryption/decryption efficiency. For frame-based schemes, e.g., SRTP, frame sizes are typically small. If a new key is generated for each frame, the overhead may be significant; a low overhead algorithm is important.

3.3.3 Rate Adaptation Timing

Figure 3-6 shows a theoretical timing diagram for download and decryption. Figures 3-9 and 3-10 show traces from our production stitched media file player implementation, running on the Motorola Droid™ 2. We show runs for the HC128 and RC4 ciphers using 32768 byte key lengths. We produced stitched media files with two encodings. The first encoding contains 126 kbps video (H.264) with 64 kbps audio (AAC-LC). The second encoding contains 286 kbps video (H.264) with 64 kbps audio (AAC-LC). The two encodings were stitched together and the stitched media files were then encrypted using the two ciphers at the specified key length. The tests were performed on the Verizon Wireless™ 3G network.

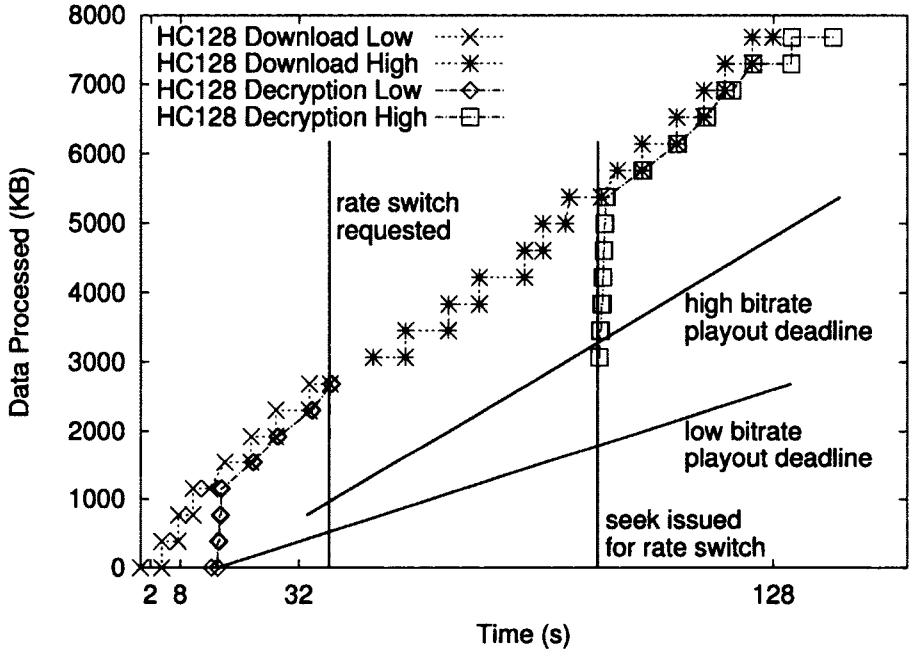


Figure 3-9: HC128 stitched media file playback rate switch trace.

With respect to the architecture shown in Figure 3-1 (a), the data source performs the on-device data decryption, prior to forwarding the content to the native media player. For the Android™ platform, the data source is implemented as an HTTP proxy. The encrypted data is retrieved by a separate HTTP download thread and cached on disk until the data source needs it. The encrypted data is downloaded in fixed sized (384 KB) chunks, to simplify the download calculations.

In Figures 3-9 and 3-10, each vertical level corresponds to a 384 KB chunk of data. Our implementation always downloads four chunks of data before starting playback. We can see that after eight low bitrate chunks have been downloaded, a rate switch is determined viable and download of high bitrate chunks is initiated. We choose to download high bitrate data that overlaps with the low bitrate data being playing out, to provide maximum flexibility in when precisely to execute the rate switch. The rate switch time is set well in the future to take full advantage of the low bitrate data already downloaded and sent to the player. Though the download and switch times could be optimized to increase quality by

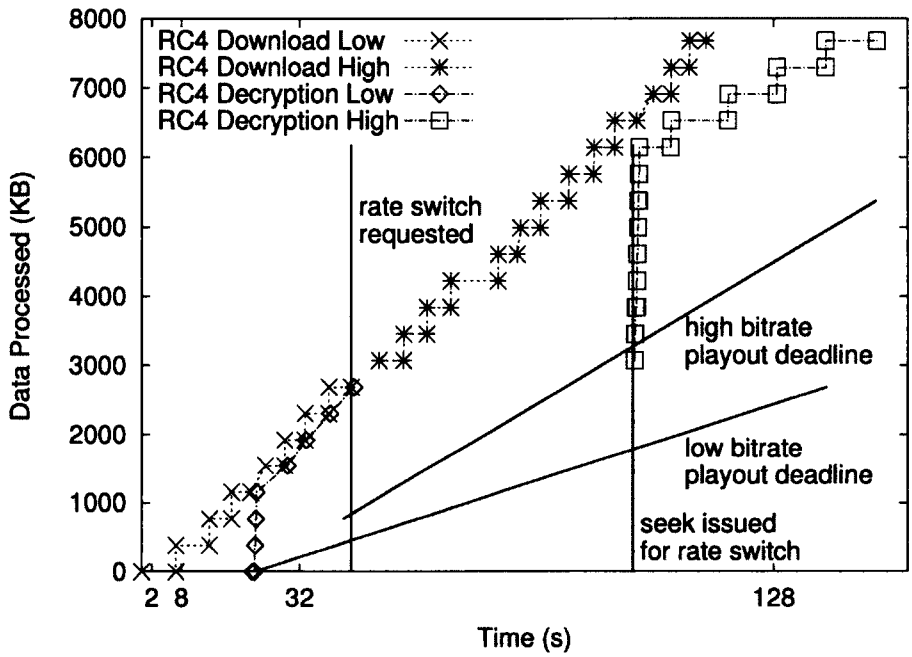


Figure 3-10: RC4 stitched media file playback rate switch trace.

switching to the high bitrate sooner, this particular implementation is designed to minimize the probability of playback interruption.

Though we can see in Figures 3-9 and 3-10 that the downloads occur well in advance of the rendering time, it is more important that the data be available when the player requests it, to prevent underrun. Paced delivery from the HTTP proxy (data source) is ideal for bandwidth management, but some less robust media players are more prone to prematurely declaring underruns. Media players which take a download-and-play approach often prefer that all the data be immediately available, and the data source must adapt accordingly.

3.4 RTP Segment Files

We implemented RTSP proxy libraries for the Google Android^{TM} platform, using the architecture shown in Figure 3-1 (b). It takes advantage of the native media player which connects to the RTSP proxy. The RTSP proxy responds with the presentation descrip-

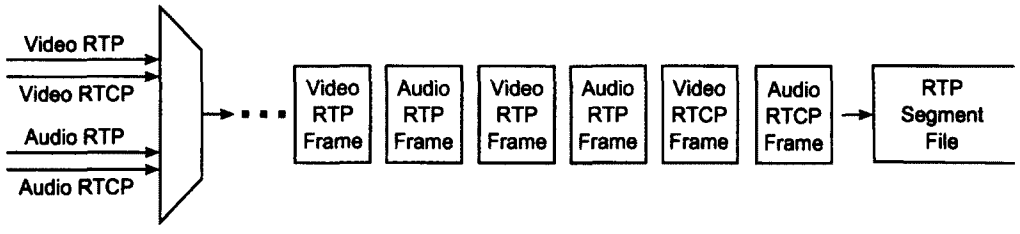


Figure 3-11: RTP time-based serialization for segment creation.

tion in Session Description Protocol (SDP) [95] format, and configures the audio and video RTP and RTCP channels. We also implemented a frame-based data source for the Microsoft® Silverlight™ platform (as an alternative to Smooth Streaming), using the architecture shown in Figure 3-1 (a). It takes advantage of the asynchronous frame-based `MediaStreamSource` APIs through which the native media player requests data. The data source parses the RTP headers and RTCP packets to properly order the frames for delivery to the media player.

Figure 3-11 diagrams the process of creating RTP segment files from separate video RTP/RTCP and audio RTP/RTCP streams. Individual RTP and RTCP packets are multiplexed into a single stream using the RTP timestamp in each packet. A strict time-based ordering simplifies parsing for the RTSP proxy. The RTP packets are prepended with a small header containing track information and packet size. The RTSP proxy parses the segment file sequentially, and paces RTP/RTCP packet delivery based on the relative RTP timestamps. The frame-based data source parses the segment file, queueing audio/video frames and providing them to the native media player on request.

Rate adaptation is performed transparently in the RTSP proxy or frame-based data source. As long as video frame rate and resolution are maintained, stream switching is a viable method for rate adaptation. In this case, the stream switching is done in the client-side proxy, rather than at the server-side. From the controller and HTTP downloader perspective, rate adaptation procedures are identical to any other HTTP adaptive streaming protocol; only the last-inch delivery to the native media player differs. Rate adaptation

algorithms are discussed in further detail in Chapter 5.

3.5 Server-side Pacing Transparency

Though most of our experiments are performed using local Apache HTTP servers or commercial CDNs, we also investigated the interoperability of our client rate adaptation architecture with the Zippy HTTP streaming server described in Chapter 2. The use of server-side in addition to client-side pacing can be counterproductive, if the client rate adaptation algorithm uses download rate in its rate selection logic (as most do). Though this could be exploited by the server to influence client rate selection (see Section 5.8), in general it is undesirable. The impact of a pacing server is really dependent upon the burst sizes employed by the server. If the burst size is larger than the size of the segment being requested by the client, there is likely to be no impact on the client. If the server uses a large initial burst to, like the one described in Section 2.2, this further reduces the probability of data pacing occurring, though it is not a guarantee. HTTP streaming servers need to be aware of the content being requested and the context in which it is being requested. HTTP Range requests with range sizes that correspond to video durations of less than 10 seconds are typically indicative of client-side pacing schemes, as with our stitched media file approach. The retrieval of content whose duration is less than 10 seconds is another possible indication of client-side pacing, as with our RTP segment approach. A more explicit proprietary HTTP header is another option, though the header would require standardization. Applying some additional intelligence to the HTTP streaming server allows it to optimize delivery for legacy download and play clients, without hindering newer adaptive bitrate clients.

3.6 Summary of Client Rate Adaptation Contributions

With the release of HLS protocol in the Apple iPhone®, HTTP adaptive streaming gained a great deal of momentum. Though much of the popularity can be attributed to the

marketing of the iPhone itself, a combination of the simplicity of the scheme, the fact that it just worked, and its mandated use in all applications that streamed video over the AT&T network made it wildly successful. Though the Microsoft® SilverLight™ Smooth Streaming protocol [96] was released prior to HLS, its proprietary nature inhibited its initial growth. Though the HLS specification was released as an open standard, it mandated the use of the MPEG2-TS transport stream format, which, up until that time had primarily only been used in broadcast television delivery. The majority of consumer devices lacked native support for MPEG2-TS content, leaving most devices still with no means of supporting HTTP adaptive streaming.

The primary contribution of our client-side data proxy research was the definition of an architecture that was flexible enough to be ported to a wide variety of client devices and able to interface with a wide variety of native media player interfaces. Using the client rate adaptation architecture we were able to significantly extend the reach of HTTP adaptive streaming technology. We were able to use our architecture to implement HTTP adaptive streaming on a number of different platforms including: Android™, Blackberry®, and Windows®, using different data proxy interfaces to communicate with the native media players, but while still maintaining a common rate selection and content download architecture. The common architecture provides the ability to easily test and evaluate the performance of different rate adaptation algorithms on multiple devices and operating system (OS) platforms. Our rate adaptation client was instrumental in the testing and refinement of our bitrate selection override proxy described in Chapter 4. Though the implementation details for each OS platform provide interesting data points that enhance our understanding of the versatility of HTTP adaptive streaming, the common rate adaptation client architecture also provides a testbed for our continued investigation of rate adaptation algorithms, like the one discussed in Chapter 5.

Chapter 4

HTTP Adaptive Streaming Bitrate Selection Override Proxy

In working with the HTTP adaptive streaming clients described in Chapter 3, it became evident that the greedy nature of the individual clients was not entirely conducive to fair bandwidth allocation between multiple clients. Though greedy clients may eventually reach a steady state bandwidth distribution, competition for bandwidth can cause bitrate thrashing and throughput squelching which affect QoE in individual clients. We began to look at methods for implementing session tracking and fair bandwidth distribution between sessions. In a degenerate case, this could be implemented as part of a server, similar to the methods described in Chapter 2, but most large scale deployments have many servers in a server farm, which require coordination, and individual clients may experience different network conditions downstream from the servers. For this reason, we adopted a network proxy-based approach to managing HTTP adaptive streaming session. In this chapter, we discuss two paradigms for performing bandwidth management using rate selection overrides on a per-segment basis.

Using HLS as the model for HTTP adaptive streaming, we examined the use of a network-based proxy for bandwidth distribution and CoS differentiation. We apply the network-based proxy approach to both standard HLS clients as well as our frame-based proxy client described in Chapter 3. The network proxy can be deployed at various network depths. Depending on the network depth, one of two modes may be used:

- full override mode: where the network proxy performs bitrate selection for the client,

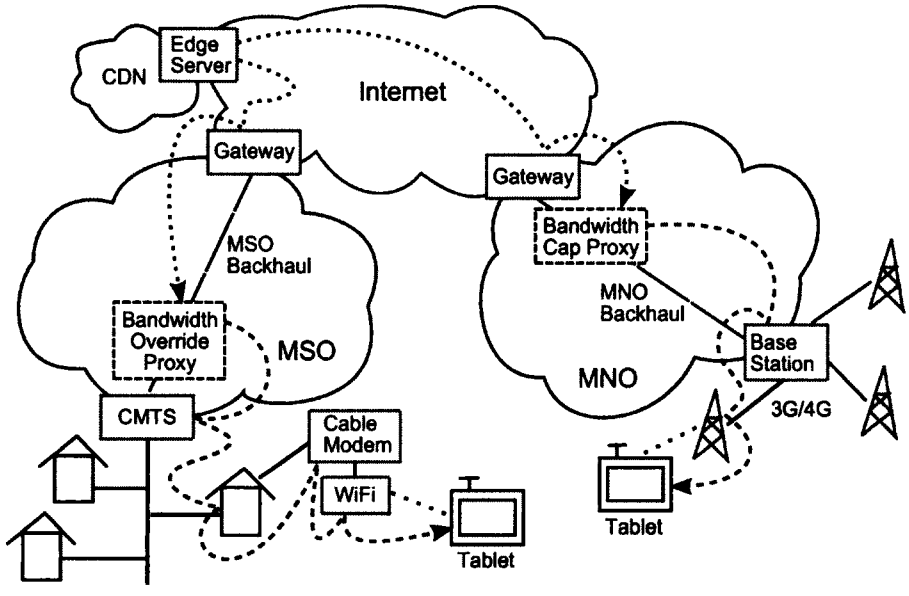


Figure 4-1: Rate selection override network proxy deployment diagram.

completely removing rate adaptation functionality from the client, or

- bandwidth cap mode: where the network proxy allows the client to perform rate adaptation but may cap the client's bitrate, overriding the client's selection if it exceeds the bandwidth cap.

For deployments where the network proxy is located deep in the network, close to the client, the network proxy can take complete responsibility for rate selection, using full override mode. In Figure 4-1, this is represented by the bandwidth override proxy in the MSO network. The content coming from the external CDN crosses the MSO backhaul, down to the CMTS, where the network proxy is situated, close to the clients. If the client does not support rate adaptation natively, then the network proxy can provide that functionality. More likely, though, if a client rate adaptation algorithm is overly aggressive and causing congestion, or not aggressive enough and providing a poor QoE to the viewer, the network proxy can circumvent those deficiencies and apply a provider or operator defined rate selection criteria.

For deployments where the network proxy is located further away from the client, it

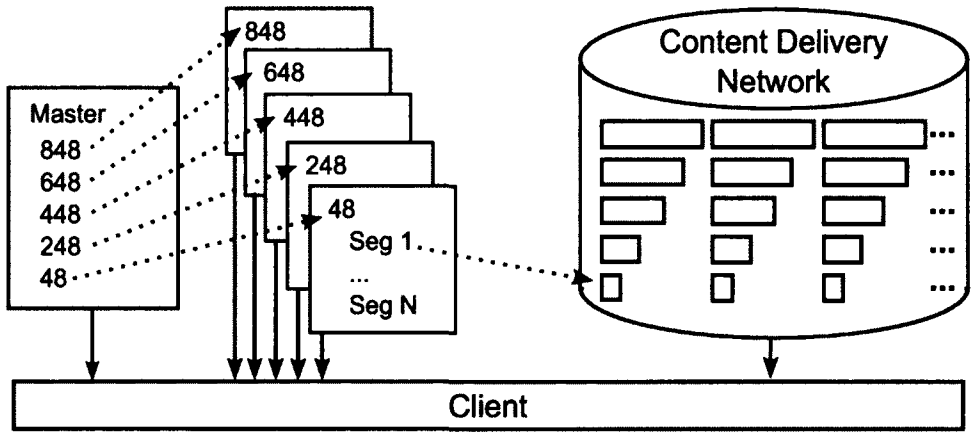


Figure 4-2: Standard HTTP Live Streaming delivery.

often makes sense to use the bandwidth cap mode, rather than the full override mode, which allows the client to continue selecting bitrates based on its localized network conditions. In Figure 4-1, this is represented by the bandwidth cap proxy in the MNO network. The content coming from the external CDN hits the network proxy, prior to crossing the MNO backhaul down to the radio access network (RAN). The bandwidth cap proxy aggregates client requests from throughout the MNO network and caps them based on the limits of the MNOs Internet connection. If the client's local bandwidth is high, but congestion occurs further upstream, where the network proxy resides, the network proxy can cap the bitrate delivered to the client. If the client's local bandwidth falls, however, that condition is undetectable by the network proxy, because the client is downstream from the network proxy. In that case, the client can still react on its own, by selecting a lower bitrate.

Figure 4-2 shows a standard HLS configuration, with a master m3u8 manifest containing URLs pointing to individual bitrate manifests. Each individual bitrate manifest contains URLs pointing to video segment files which are distributed through a CDN. The m3u8 manifests may be stored in the CDN with the segment files, or that may alternately be distributed through a separate application server.

Figure 4-3 shows the standard m3u8 and segment requests flow. The client begins by retrieving the master m3u8 manifest. The client begins playback with the first bitrate listed

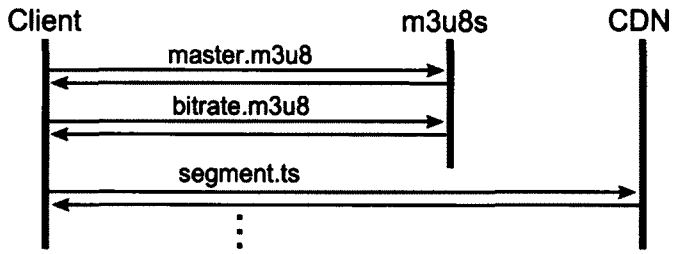


Figure 4-3: Standard HTTP Live Streaming delivery request flow.

in the master m3u8 manifest; it is expected that the first bitrate is the suggested bitrate of the content publisher. The client then retrieves the individual bitrate m3u8 manifest and begins downloading segments.

4.1 Network Proxy Controlled Rate Adaptation

In the full override mode, the network proxy takes full control over rate adaptation. Available bitrate information is hidden from the client by the network proxy. The client has no expectation of rate adaptation, and the network proxy performs rate selection and replacement in a transparent fashion.

Figure 4-4 shows an augmented HLS configuration with a bandwidth manager inserted between the client and the m3u8 manifests. The standard m3u8 manifests and segments are used; no modification to the master m3u8 manifest, the individual bitrate m3u8 manifests, or the video segment files themselves is required. The bandwidth manager uses the existing m3u8 manifests to determine which bitrates are available and where the segments are located.

Figure 4-5 shows the augmented m3u8 and segment requests flow for the full override mode. When a client requests the master m3u8 manifest, the bandwidth manager responds to the client's master m3u8 manifest request with a generic single bitrate m3u8 manifest. The generic m3u8 manifest contains segment URLs pointing to the bandwidth manager. Having no knowledge of what actual bitrates are available, the client will request only those

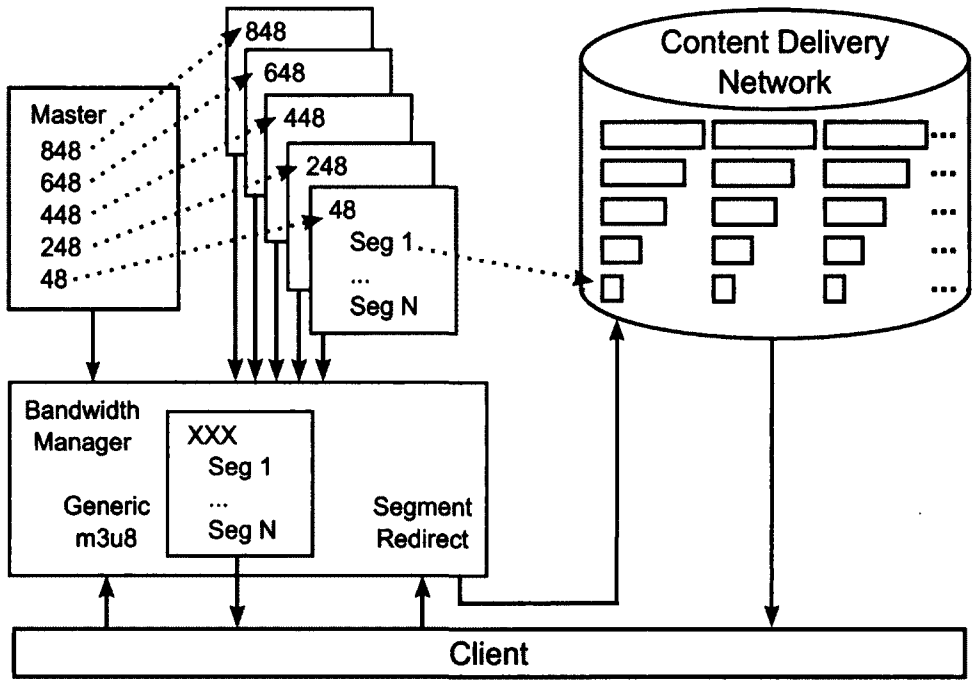


Figure 4-4: Network proxy controlled HTTP Live Streaming delivery.

segments specified in the generic m3u8 manifest. The bandwidth manager accepts the segment requests from the client, selects a suitable bitrate, based on the current bandwidth estimate, and redirects the client to the actual segment (or, in the case that insufficient bandwidth exists to deliver any segment, aborts the connection).

4.1.1 Optimized Bandwidth Usage Testbed

To better illustrate the basic functionality of our network controlled segment selection scheme, we use a scaled down configuration with two clients and limit the available bandwidth such that insufficient bandwidth exists for both clients to stream at the highest bitrate simultaneously. Though the scheme scales well beyond two devices, limiting the number of clients allows us to more easily isolate the actions of each individual client, and allows us to use actual iOS devices, where large scale tests with actual iOS devices would be cost prohibitive.

Figure 4-6 shows the network setup for our bitrate override test. The server contains

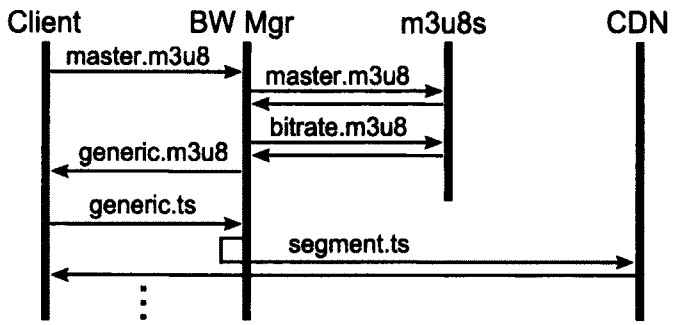


Figure 4-5: Network proxy controlled HTTP Live Streaming delivery request flow.

an Apache 2.2 server for serving m3u8 playlist files and MPEG-TS segment files. For deployment simplicity, we have also integrated the bandwidth management components (i.e., playlist and segment proxying) into the same server. In this configuration, the server responds directly to the client requests using segments stored on the local disk. For the bandwidth cap tests, we use an alternate configuration with an external CDN and the bandwidth manager issues HTTP 302 redirects to the client (or responds with a HTTP 403 Forbidden status code, in the case that the null bitrate is applied). The server is equipped with dual quad-core 2.27 GHz Xeon processors and 8 GB of RAM running a 2.6 rPath® Linux kernel.

The server is connected to a Linux server that acts as a router, routing between the WiFi access point and the server. The Linux router uses a `tc` hierarchical token bucket to implement bandwidth limits on the interface between itself and the WiFi access point. Results were captured using `tcpdump` on the interface between the Linux router and the WiFi access point. The Linux router is also equipped with dual quad-core 2.27 GHz Xeon processors and 8 GB of RAM, but running a 2.6 Red Hat® Linux kernel.

The two mobile devices A and B that were used were an iPhone® 3GS and an iPod touch® 2G, both running iOS 3.1.3. Videos were played using the native media player. The player was launched via the Safari® browser by providing a URL to the m3u8 playlist files on the server.

A 10 minute video ("Big Buck Bunny") was ingested and transcoded into five bitrates.

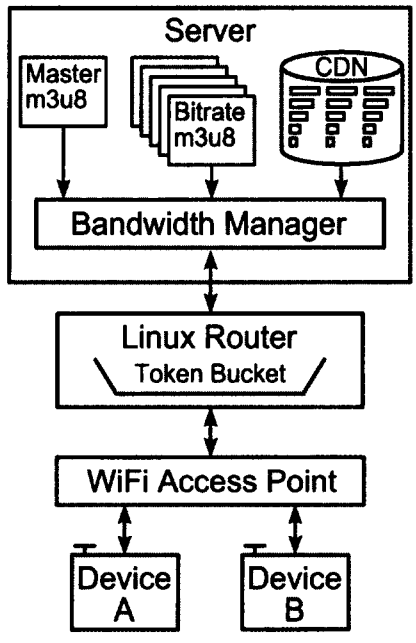


Figure 4-6: Bitrate override test network configuration.

The audio bitrate was 48 kbps for all outputs; the video was encoded at 800 kbps, 600 kbps, 400 kbps, 200 kbps, and 0 kbps (audio only) resulting in total bitrates of 848, 648, 448, 248, and 48 kbps, respectively. Each encoding was identically segmented into 10 second segments, and each segment was encrypted with AES-128.

It is important for dynamic segment replacement that the segments are synchronized, to prevent any rendering discontinuity. To prevent rendering distortions when switching rates, it is also important that every segment begin with a I-frame, that B-frames do not cross segment boundaries, and depending on the quality of the decoder, that frame rates and resolutions are maintained across bitrates. I-frames may be ensured by setting a fixed GOP duration which evenly divides into the segment duration. In our tests we used 30 fps video with a fixed GOP size of 75 frames producing an I-frame every 2.5 seconds, which divides evenly into a 10 second segment duration. We also disabled the insertion of B-frames for our tests and used consistent frame rates and resolutions for all encodings.

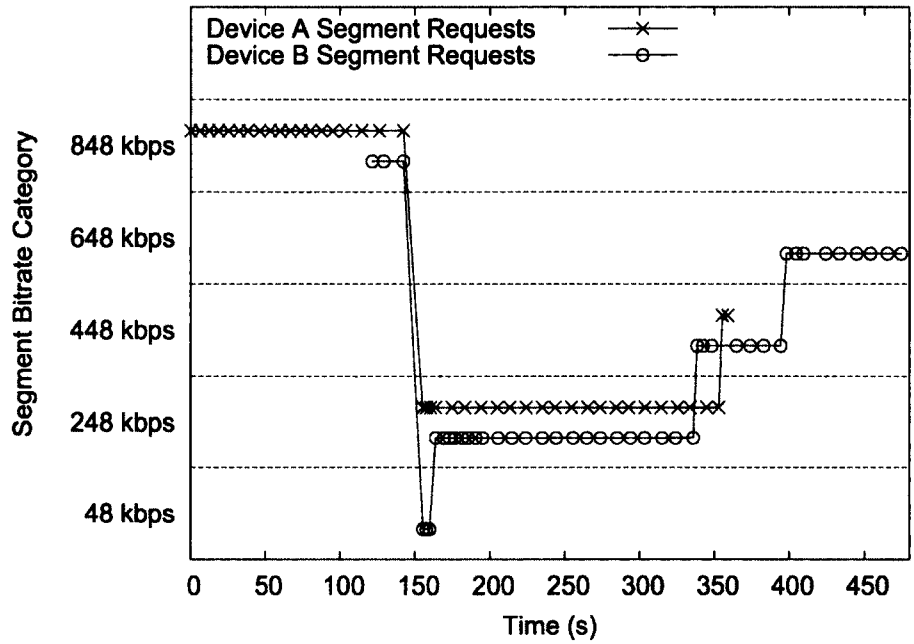


Figure 4-7: Native rate adaptation segment requests.

4.1.2 Optimized Bandwidth Usage Results

Figures 4-7 and 4-8 show the results of two experiments. Both were performed using an eight minute test run playing six minutes of video on each phone, with a two minute staggered start, i.e., device A started at time t_0, device B was started at time $t_0 + 120$, device A was stopped at time $t_0 + 360$, and device B was stopped at time $t_0 + 480$. Starting device A first allowed device A to establish a baseline before starting device B to observe the effect of contention. Device A was then stopped before device B to evaluate the recovery. The token bucket rate ceiling was set at 1600 kbps for both tests.

The first set of tests were performed using a standard m3u8 configuration, i.e., a master playlist pointing at 5 individual bitrate playlists, allowing the native media player to perform rate selection. The master playlist contains the individual bitrate playlists in descending order of bitrate, therefore the highest bitrate will always be selected as the initial bitrate. Figure 4-7 shows the segment requests for the standard m3u8 configuration. The y-axis shows which of the five bitrates was requested for each segment. The plots for device A and

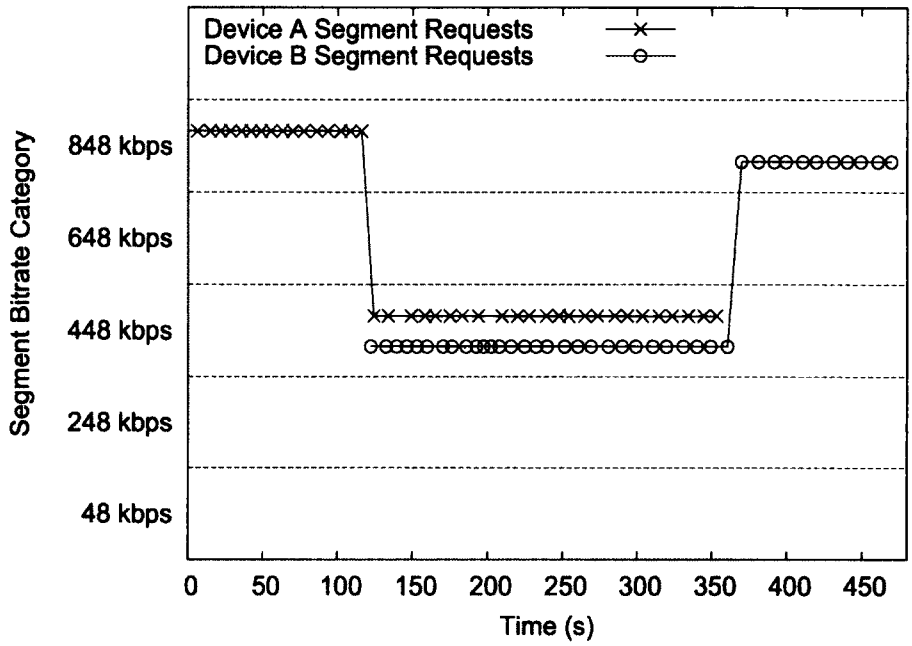


Figure 4-8: Network controlled rate adaptation segment selections.

device B have been staggered vertically to make them easier to read.

We can see the staggered start of device B at 120 seconds and the immediate back-off of both device A and device B. This is expected given that both devices are requesting 848 kbps, but a bandwidth cap is set at 1600 kbps. Device A backs off to 248 kbps, while device B goes to audio-only (48 kbps). Device B quickly recovers to 248 kbps, but the disruptive experience caused by switching to audio-only has already been felt by the user. Both devices eventually recover to 448 kbps, and when device A stops playing, device B recovers further to 648 kbps, but does not achieve 848 kbps before the end of the test run. The native media player attempts to provide high quality video initially, but then overshoots while backing off, when congestion occurs. The slow recovery prevents additional bitrate thrashing, but the user pays the penalty in slow recovery time. The time to get back to 448 kbps was around 200 seconds for both devices. Three minutes of unnecessarily low bitrate video could have been avoided with better knowledge of network conditions.

The second set of tests employed the bandwidth manager which supplied the two devices

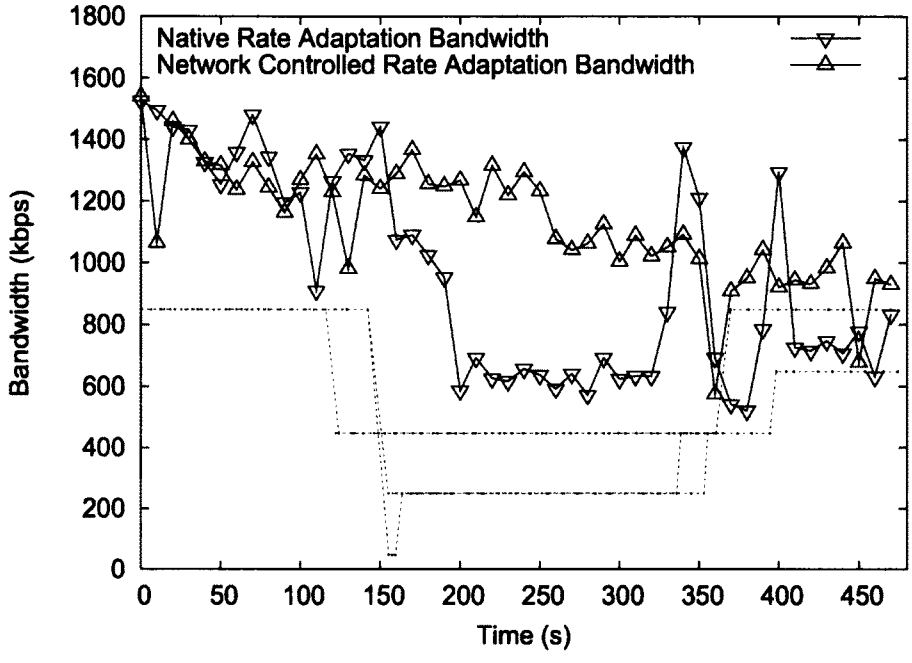


Figure 4-9: Native vs. network controlled rate adaptation bandwidth consumption.

with a single generic bitrate playlist and made bitrate selection decisions for the clients. Figure 4-8 shows the segment bitrates selected for the network proxy controlled approach. As with Figure 4-7, the plots, the y-axis shows which of the five bitrates was requested for each segment and the plots for the two devices have been staggered vertically to make them easier to read. We can again see the staggered start of device B at 120 seconds and the immediate back-off of both device A and device B. In the network proxy controlled case, however, we assume that the bandwidth manager is aware of the maximum network capacity of 1600 kbps. Using this information, the bandwidth manager is able to select suitable bitrates (i.e., 648 kbps) for both devices. The bandwidth is evenly distributed and does not exhibit the overshoot in bitrate back-off that was seen in the native case. This provides better continuity and a higher quality viewing experience. At 360 seconds, when device A stops playing, the full 848 kbps video is immediately made available to device B.

Figure 4-9 shows the total bandwidth used by devices A and B in both the native and network proxy controlled test cases. The bandwidth values are aggregated on a 10 second

basis. Bitrate information from Figures 4-7 and 4-8 are overlaid onto the plot to correlate bandwidth spikes with rate adaptation events. We can see in the native rate adaptation case, the overshoot in bitrate reduction results in a lower total bandwidth consumption through the middle portion of the graph. We can also see that there are bandwidth spikes at every rate switch. This is because the native player always requests three segments: the current segment, the previous segment, and the next segment, in a burst, when switching bitrates. In the proxied rate adaptation case, the total bandwidth usage is more linear and makes more optimal use of the available bandwidth. The slight downward trend is due to the fact that the player will initially request data at a higher rate, to fill its playback buffer. This eventually evens out to a steady state near the segment bitrate, as we see it doing near the end of the graph.

4.2 Network Proxy Capped Rate Adaptation

In the bandwidth cap mode, the network proxy measures total bandwidth usage and only overrides client bitrate requests when they would cause the overall bandwidth usage to exceed the cap. Available bitrate information is processed by the proxy so that it is aware of the available bitrates, but the information is passed through to the client unmodified. The client is unaware of the network proxy's existence and performs rate adaptation as it normally would. The network proxy performs bitrate capping and segment replacement in a transparent fashion.

Figure 4-10 shows an augmented HLS configuration with a bandwidth manager inserted between the client and the m3u8 manifests. Just as in the full override mode, the standard m3u8 manifests and segments are used; no modification to the master m3u8 manifest, the individual bitrate m3u8 manifests, or the video segment files themselves is required. The bandwidth manager also continues to use the existing m3u8 manifests to determine what bitrates are available, and where segments are located, however, it does not hide that information from the client.

Figure 4-11 shows the augmented m3u8 and segment requests flow for the bandwidth

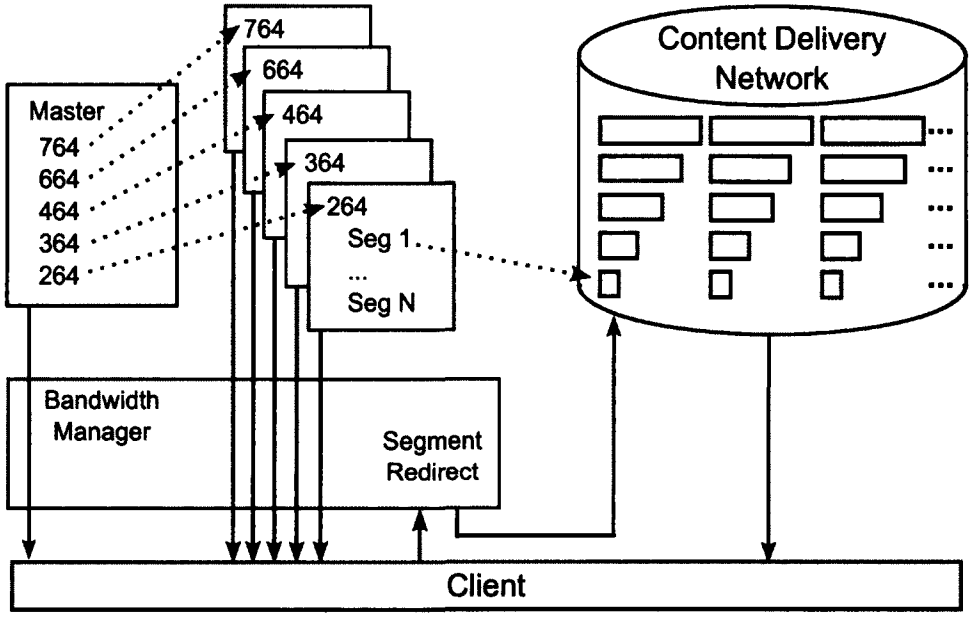


Figure 4-10: Network proxy bandwidth capped HTTP Live Streaming delivery.

cap mode. When a client requests the master m3u8 manifest, the bandwidth manager gleans bitrate information from it and passes it through to the client. In some cases, the network proxy may alter the URLs contained in the manifest to simplify interception of future requests, but the available bitrate information remains unchanged. The client begins playback, as it normally would, with the first bitrate listed in the master m3u8 manifest. It retrieves the individual bitrate m3u8 manifest and begins downloading segments. The bandwidth manager inspects each segment request to verify that it will not cause the bandwidth cap to be exceeded. If servicing the request would violate the cap, the bandwidth manager selects a more suitable bitrate and redirects the client to the alternate segment (or, in the case that insufficient bandwidth exists to deliver any segment, aborts the connection).

4.2.1 Dynamic Rate Capping Testbed

In the bandwidth capping scenario, as with the full override case, we use a scaled down configuration to simplify the discussion. In this case we use four clients and assume that the downstream bandwidth is more than sufficient to support all clients streaming the

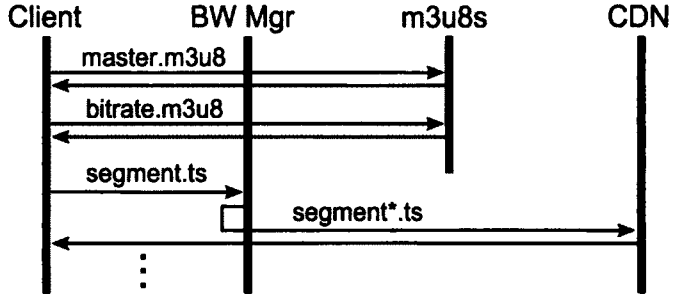


Figure 4-11: Network proxy bandwidth capped HTTP Live Streaming delivery request flow.

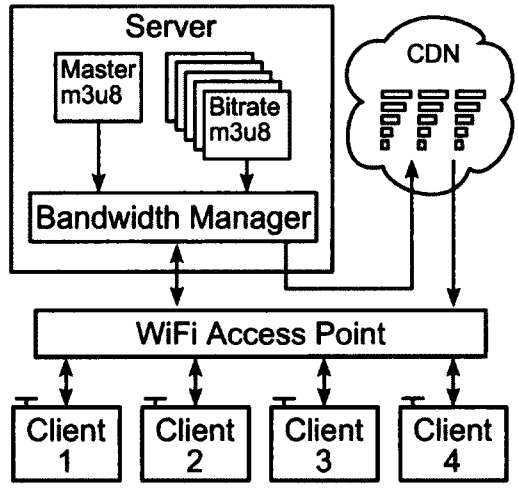


Figure 4-12: Bandwidth cap test network configuration.

highest bitrate. A bandwidth cap is assumed to apply to the upstream link. We employed Microsoft® Silverlight™ clients using a frame-based data source and RTP segments (as described in Section 3.4) running on a Windows® 7 laptop. Though the scheme scales well beyond four clients, limiting the number of clients allows us to more easily isolate the actions of each individual client.

Figure 4-12 shows the network setup for our bandwidth cap test. The server contains the bandwidth management components (i.e., playlist and segment proxying), but the segment files are in an external CDN, in this case Akamai. In this configuration, the bandwidth manager issues HTTP 302 redirects in response to client requests, redirecting them to the

CDN (or responds with HTTP 403 Forbidden status code, in the case that the null bitrate is applied). The server is equipped with dual quad-core 2.27 GHz Xeon processors and 8 GB of RAM running a 2.6 rPath® Linux kernel. The server is connected through WiFi to the clients running on the a laptop. Results were captured by the bandwidth manager.

A 10 minute video ("Big Buck Bunny") was ingested and transcoded into five bitrates. The audio bitrate was 64 kbps for all outputs; the video was encoded at 700 kbps, 600 kbps, 400 kbps, 300 kbps, and 200 kbps resulting in total bitrates of 764, 664, 464, 364, and 264 kbps, respectively. Each encoding was identically segmented into 10 second segments, and each segment was encrypted with HC-128. As with the full override case, segments were synchronized, to prevent any rendering discontinuity, segment began with a I-frame, B-frames were disabled, and frame rates and resolutions were maintained across bitrates. I-frames were ensured using a fixed GOP size of 24 frames with a frame rate of 24 fps producing an I-frame every second.

4.2.2 Dynamic Rate Capping Results

Figure 4-13 shows the results for the bandwidth capping test. The test was performed using a ten minute test run, with client 1 playing video throughout the entire test, starting at time t_0, client 2 starting at time $t_0 + 90$ and stopping at time $t_0 + 270$, client 3 starting at time $t_0 + 150$ and stopping at time $t_0 + 360$, and client 4 starting at time $t_0 + 190$ and stopping at time $t_0 + 430$. Staggering the start of each client makes it easier to see the effect of adding each additional client. Likewise, staggering the stop of each client makes it easier to see the effect of removing each client in turn. The upstream bandwidth cap was set at 1500 kbps.

The tests were performed using a standard m3u8 configuration, i.e., a master playlist pointing at 5 individual bitrate playlists, allowing the native media player to perform rate selection. The master playlist contains the individual bitrate playlists in descending order of bitrate, therefore the highest bitrate will always be selected as the initial bitrate. Because the actual bandwidth available to the clients is plentiful, the clients always request the

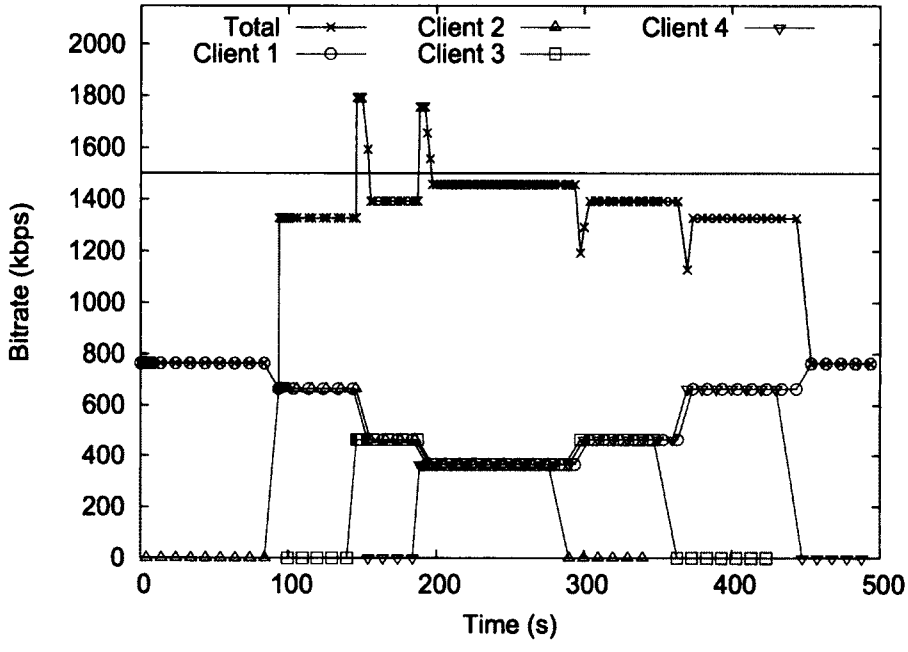


Figure 4-13: Network proxy capped segment requests.

highest bitrate. The bandwidth manager, however, must enforce the 1500 kbps bandwidth cap. Figure 4-13 shows the bitrate supplied to each client, in response to their segment requests. Figure 4-13 also plots the sum total of all the segment bitrates being accessed.

We can see that at the start, client 1 requests 764 kbps and receives 764 kbps. The total bandwidth used at that point is 764 kbps, well below the 1500 kbps cap. When client 2 starts after 90 seconds, it also requests 764 kbps, however, the combined total of clients 1 and 2 each retrieving 764 kbps would exceed the cap. The bandwidth manager detects this and caps both clients at 664 kbps. When client 3 starts after another 60 seconds, it also requests 764 kbps, however, the combined total of clients 1 and 2 retrieving 664 kbps and client 3 retrieving 764 kbps would again exceed the cap. The bandwidth manager again detects the cap violation and this time caps all three clients at 464 kbps. Similarly, when client 4 starts playing, again the total bitrate would exceed the cap, and the bandwidth manager caps all four clients at 364 kbps. We can see that there are momentary spikes in the total bandwidth, due to the initial bursts of requests from the clients, as they initially

fill their playback buffers. The bandwidth manager detects these but allows them, given the understanding that initial playback buffer fills are one time transient bursts.

When the clients complete their playback, the bandwidth manager detects the end of the session and recalculates the distribution of bandwidth between the active clients. Session detection is discussed in further detail in the follow section. We can see in Figure 4-13 that 20 seconds after client 2 stops playing, the bandwidth cap on clients 1, 3, and 4 is raised back to 464 kbps. Similarly, 20 seconds after client 3 stops playing, the bandwidth cap on clients 1 and 4 is raised to 664 kbps, and 20 seconds after client 4 stops playing, the bandwidth cap is lifted and client 1 returns to playing 764 kbps.

4.3 Session Detection

We have assumed that the network proxy is aware of the networks bandwidth limitations. The bandwidth manager in Figures 4-4 and 4-10 compares the bandwidth limits with the estimated bandwidth usage. To account for the bursty nature of segment download, the bandwidth manager estimates bandwidth usage based on the bitrates of segment requests. It then amortizes those bitrates over the segment duration to account for request interleaving.

The bandwidth manager also estimates the number of active sessions based on request frequency. Given a segment duration L, it is expected that each client must request a segment every $L \pm L$ seconds. Sessions are therefore considered active if a segment request has been received in the past $2 \cdot L$ seconds. For the Apple recommended segment duration of $L = 10$, a session detection latency of 20 seconds may be used. For the purposes of CoS differentiation, it is assumed that the CoS for a given client is identifiable in the request (e.g., as a query string parameter or via a well-known HTTP header field).

Though the bandwidth manager only requires limited statistics for tracking client sessions, the network proxy has access to a significant amount of information about individual client content viewing habits. Content access patterns can be used to enhance caching algorithms, as well as to predict future bandwidth requirements. Further investigation of segment request data mining in the network proxy is an interesting topic for future research.

4.4 Proxy-based CoS Enforcement

Using the proxy architectures described above, we are able to not only override individual client requests, but also implement a CoS enforcement scheme, which uses client request overrides to prevent clients in a given CoS from exceeding their rightful bandwidth allocations. We assume a set of C clients $\{0, \ldots, C-1\}$ and a maximum network capacity N. Each client belongs to one and only one of W classes of service $\{1, \ldots, W\}$, where w_c represents the CoS for a given client c, and a larger w_c value corresponds to a higher CoS. We also assume a discrete set of B bitrates $\{b_0, \ldots, b_{B-1}\}$, which may be delivered to a client, where b_i^c denotes the bitrate i currently being streamed to a given client c. For simplicity we assume that $\forall i \in 1 \ldots B-1 : b_i > b_{i-1}$ and that $b_B = \infty$ and that $b_{-1} = 0$. We propose that every client within a CoS should receive the same bitrate and that clients with a higher CoS should receive an equal or higher bitrate than clients with a lower CoS, i.e., $\forall i, j \in 0 \ldots C-1 : (w_i = w_j \Rightarrow b_x^i = b_x^j) \wedge (w_i > w_j \Rightarrow b_x^i \geq b_x^j)$.

N	total network capacity (in kbps)
E	excess network capacity (in kbps)
C	total number of clients
C_w	number of clients in CoS w
W	number of classes of service
B	number of discrete bitrates
L	segment duration (in seconds)
b_i^c	current bitrate being downloaded by client c (in kbps)
$b_i^{(w)}$	current bitrate assigned to CoS w (in kbps)
w_c	CoS of client c ($\in \{1, \ldots, W\}$)

Table 4.1: Rate selection algorithm variables.

This strict assignment of a single bitrate to all clients within a class of service may result in unused network resources, if the excess network capacity E is not large enough to accommodate all clients, i.e., $E < (b_{x+1} - b_x) \cdot C_w$, where C_w is the number of clients in class w. For simplicity, we maintain this restriction in the current discussion. We relax this constraint in the discussion of rate adaptation algorithms in Chapter 5. In the full

override mode, the bitrate selected is the bitrate presented to the client. In the bandwidth cap mode, the bitrate selected is the maximum bitrate the client is allowed to have. The client may request a bitrate lower than the maximum determined by the algorithm. In such a case, the excess capacity E may be higher than estimated.

Algorithm 4.1 Strict depth first bandwidth distribution algorithm.

```
E ← N
for all w ∈ 1...W do
  b_i^(w) ← b_{-1}
end for
for all x ∈ 1...B do
  for all w ∈ W...1 do
    if E < (b_x − b_{x−1}) · C_w then
      return
    else
      b_i^(w) ← b_x
      E ← E − (b_x − b_{x−1}) · C_w
    end if
  end for
end for
```

Algorithm 4.1 shows an example of a basic breadth-first bandwidth allocation scheme. Each CoS is initially assigned the null bitrate b_{-1}. We then iterate through each CoS from high to low and increase the bitrate of each CoS by one level. This process continues until increasing the bandwidth for the next CoS would exceed he available bandwidth. We denote the bitrate b_i currently assigned to a given CoS w by $b_x^{(w)}$. Stopping as soon as any CoS is unable to upgrade ensures that priority inversion does not occur in favor of any lower CoS. It also, however, prevents any higher CoS from receiving extra bandwidth. Assuming every CoS has the same number of clients, i.e., $\forall i, j \in 1 \ldots W : C_i = C_j$, the latter is usually not an issue, as bitrate distributions are typically exhibit super-linear increases, i.e., $\forall i \in 1 \ldots B - 2 : b_{i+1} - b_i \geq b_i - b_{i-1}$. The de facto standard Apple suggested bitrates: $\{150, 240, 440, 640, 1240, 2540, 4540\}$ [97], e.g., contain inter-bitrate gaps of: $\{90, 200, 200, 600, 1300, 2000\}$.

Given that the number of bitrates B and the number of classes of service W are typ-

ically small, the cost of rerunning Algorithm 4.1 is relatively small. It is assumed that the maximum network capacity N does not change very often, however, when it does, the algorithm must be rerun. The incremental addition or removal of individual sessions which do not impact the bitrate allocations, however, do not require the algorithm to be rerun. Given the previously calculated excess capacity E, we can quickly verify if the session that was added or removed will impact the current bitrate allocations. When a new session is added to a given CoS group w, as long as $E - b_x^{(w)} > 0$, then no reallocation is required. Similarly, whenever a session is removed from a given CoS group w, as long as $\forall i \in 1 \ldots W : E + b_x^{(w)} < b_x^{(i)} \cdot C_i$, then no reallocation is required. If either of those conditions is violated, a reallocation should be initiated.

In highly dynamic networks, where sessions come and go frequently, the elasticity of a larger excess capacity E may be beneficial. Less dynamic networks, however, may prefer to allocate excess capacity to higher CoS clients. In cases where higher classes of service may have fewer clients, we can modify Algorithm 4.1 to allow it to attempt to allocate more of the excess capacity, while still maintaining the CoS enforcement.

Algorithm 4.2 Loose depth first bandwidth distribution algorithm.

```
minW ← 1
E ← N
for all w ∈ 1...W do
  b_i^(w) ← b_-1
end for
for all x ∈ 1...B do
  for all w ∈ W...minW do
    if E < (b_x − b_x−1) · C_w then
      minW ← w + 1
    else
      b_i^(w) ← b_x
      E ← E − (b_x − b_x−1) · C_w
    end if
  end for
end for
```

In Algorithm 4.2, we can see that instead of stopping as soon as any CoS is unable to

upgrade, as in Algorithm 4.1, it instead sets a CoS floor preventing that CoS or any lower CoS from being considered for further bitrate upgrades. The augmented algorithm now allows higher classes of service to claim higher bitrates if sufficient excess capacity exists. Depending on the distribution of clients across classes of service, and specific policies that network operators may wish to enforce, alternate algorithms may provide more optimal bandwidth allocations. The application of rate selection override to less general scenarios is an interesting area for future research.

4.4.1 CoS Rate Capping Results

Using the same configuration as the one described in Section 4.2.1, we tested the CoS differentiating bandwidth capping algorithm described in Algorithm 4.2. We used the same four Windows® clients, but this time placed each client into its own CoS and lowered the bandwidth cap to 1450 kbps. The clients are order by their CoS, i.e., client 1 is in the highest CoS, while client 4 is in the lowest CoS. Figure 4-14 shows the results of the ten minute test run, with client 1 playing video throughout the entire test, starting at time t_0, client 2 starting at time $t_0 + 80$ and stopping at time $t_0 + 330$, client 3 starting at time $t_0 + 150$ and stopping at time $t_0 + 400$, and client 4 starting at time $t_0 + 230$ and stopping at time $t_0 + 480$. We again staggered the start and stop of each client to make it easier to see the effects of adding and removing each client.

We can see that at the start, client 1 requests 764 kbps and receives 764 kbps. The total bandwidth used at that point is 764 kbps, well below the 1500 kbps cap. When client 2 starts playing, it also requests 764 kbps, however, the combined total of clients 1 and 2 each retrieving 764 kbps would exceed the cap. The bandwidth manager detects this and caps the lower CoS client, i.e., client 2, at 664 kbps. When client 3 starts playing, it also requests 764 kbps, though that is immediately reduced to 664 kbps so as not to exceed the bitrate currently assigned to client 2. Even at 664 kbps, the combined total of client 1 retrieving 764 kbps and clients 2 and 3 retrieving 664 kbps would again exceed the cap. The bandwidth manager again detects the cap violation and this time caps all three

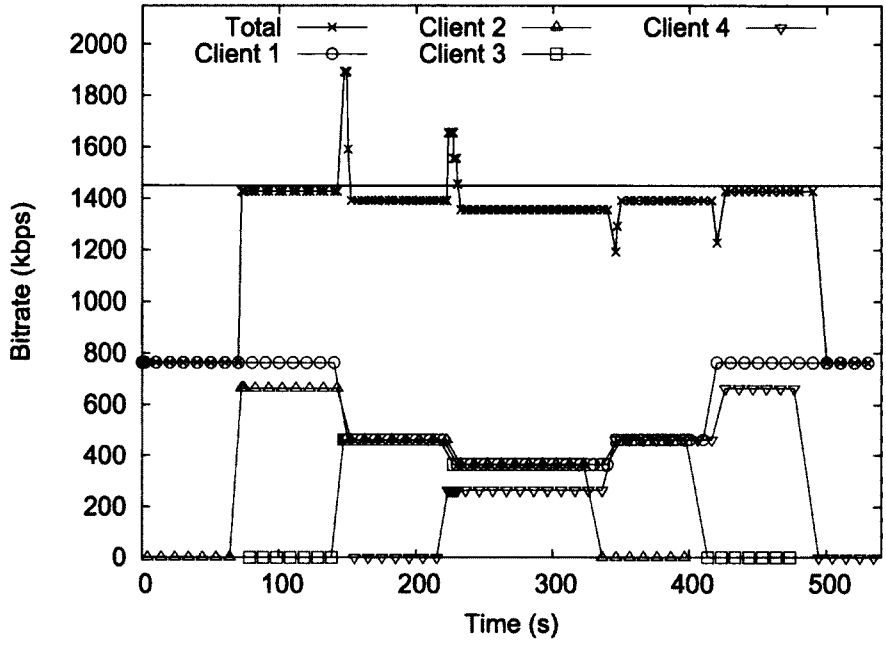


Figure 4-14: Network proxy capped segment requests (multiple CoS).

clients at 464 kbps. When client 4 eventually starts playing, it also requests 764 kbps, but is immediately lowered to 464 kbps so as not to exceed the bitrate currently assigned to client 3. Assigning all four clients 464 kbps, however, would exceed the cap. The bandwidth manager detects the cap violation and lowers client 4 to 364 kbps. As with Figure 4-13, that there are momentary spikes in the total bandwidth, due to the initial bursts of requests from the clients, as they initially fill their playback buffers. These are allowed by the bandwidth manager with the understanding that initial playback buffer fills are one time transient bursts.

When the clients stop their playback, the bandwidth manager detects the end of each session and recalculates the distribution of bandwidth between the active clients. We can see in Figure 4-14 that 20 seconds after client 2 stops playing, the bandwidth cap on clients 1, 3, and 4 is raised back to 464 kbps. Similarly, 20 seconds after client 3 stops playing, the bandwidth cap on clients 1 and 4 is raised back to 764 kbps and 664 kbps, respectively, and client 1 continues with 764 kbps until it completes its playback.

4.4.2 Rate Selection Transparency

In both the full override mode and the bandwidth cap mode, the network proxy works transparently with respect to both clients and servers, simultaneously. We have verified this functionality in conjunction with the server pacing scheme described in Chapter 2 and multiple HTTP adaptive streaming approaches, including the RTP segmentation scheme described in Chapter 3. By making rate selection decisions at the segment request level, using only the encoded bitrate and maximum network capacity (rather than instantaneous throughput measurements), the network proxy-based approach is able to be agnostic to both client and server-side pacing schemes. This abstraction allows the proxy to use HTTP 302 Redirects (or HTTP 403 Forbidden responses, in the case that the null bitrate is applied), instead of terminating and transparently proxying TCP sessions, which improves the scalability of the network proxy and simplifies the calculations in Algorithms 4.1 and 4.2.

4.5 Summary of Bitrate Selection Override Proxy Contributions

As we developed the client rate adaptation architecture described in Chapter 3, the inherently greedy nature of rate selection algorithms was very apparent. An overly aggressive implementation affects not only the network, but also the local device CPU. The server pacing scheme discussed in Chapter 2 represented the first generation solution for slowing down the naturally greedy nature of clients without impacting playback. Considering the net effect of pacing, i.e., reduction in network throughput while honoring playback boundaries, rate selection override produces the same result for HTTP adaptive streaming clients. Taking advantage of the general interchangeability of different bitrate segments on which HTTP adaptive streaming is predicated, we developed our network proxy architecture for performing rate selection override.

The primary contribution of our network proxy research was to demonstrate the ability to create a scalable platform from which to monitor and manage HTTP adaptive streaming

sessions. We used our network proxy to demonstrate two different rate selection override modes: full override mode and bandwidth capping mode. Though these two modes may not cover all possible policy enforcement scenarios, they do represent a significant step forward in the evolution of managed OTT video delivery. Networks have long employed traffic monitoring and management schemes (e.g., DiffServ) to regulate network access by an over-subscription of greedy clients. Traditional network management schemes also support CoS differentiation for service level agreement (SLA) enforcement. Though packet-level traffic management schemes may be applied to video delivery streams, we believe that more intelligent content-aware traffic management schemes produce better QoE for the viewer. The second contribution of the network proxy rate selection override research was our ability to clearly demonstrate improved QoE through segment-level network resource utilization management. The ability to manage network resources while maintaining high QoE is imperative for content providers and network operators who wish to enforce and monetize video delivery SLAs.

The network proxy is a natural extension of the CDN request router which is responsible for directing content requests to a suitable content asset location and logging the content access. Though this typically just involves redirecting a client to a surrogate cache containing the requested content, our research essentially investigates the ability to use the CDN request router to make more intelligent content selection decisions (i.e., by performing rate selection override). We also investigated the ability to perform session tracking, based on the temporal locality of client request patterns. The amount of client and session information which can be gleaned from content requests is overwhelming, but the CDN request router has access to all of it. With the addition of a small amount of content metadata, the CDN request router is perfectly positioned to enforce policies and make rate selection override decisions.

Traditionally, there has been no reason for a third party CDN to accept responsibility for enforcing content provider policies. As MSOs and MNOs continue to migrate toward OTT delivery paradigms and are increasingly building out their own internal CDNs, these

service providers must take responsibility for their own internal CDN request routing. The rate selection override proxy, as part of the CDN request router, provides operators with an approach for implementing centralized OTT video delivery management platform. The third major contribution of our network proxy architecture is that it provides a foundation for continued research into transparent management techniques for unmanaged OTT clients. Our CoS policy enforcement approaches have proven successful and continued research into policy enforcement shows great promise.

Chapter 5

HTTP Adaptive Streaming Rate Adaptation Algorithm

In Chapter 3 we presented an architecture for implementing data proxies in client devices to perform rate adaptation. In Chapter 4 we presented an architecture for implementing network proxies to perform rate adaptation and rate selection override. In this chapter, we discuss a rate adaptation algorithm, which, in its base form, provides a typical greedy approach to rate selection in HTTP adaptive streaming clients. We also detail configurable parameters which enable the rate adaptation algorithm to support CoS differentiation. The algorithm is suitable for use with any client-side architecture, including the one described in Chapter 3. It also works with the transparent bandwidth capping capabilities described in Chapter 4. Though we focus our discussion on the CoS differentiation properties of the rate adaptation algorithm, our baseline comparison is with the degenerate single CoS case of our rate adaptation implementation.

5.1 Segment-based Queueing Model

Using the same notation as we did for the discussion of network proxy-based CoS enforcement in Section 4.4, we start with a set of C clients $\{0, \ldots, C-1\}$ and a maximum network capacity N. Each client belongs to one of W classes of service $\{1, \ldots, W\}$. We use w_c to denote the CoS to which a given client c belongs. Larger w_c values correspond to a higher CoS. We also assume a set of B bitrates $\{b_0, \ldots, b_{B-1}\}$, where b_i^c denotes the bitrate i currently

being downloaded by client c. For simplicity we assume that $\forall i \in 1 \ldots B-1 : b_i > b_{i-1}$ and that $b_B = \infty$ and $b_{-1} = 0$. We then add a simple queueing scheme, where each client has a queue of length Q, measured in number of segments. We use q_c to denote the current queue occupancy for client c.

N	total network capacity (in kbps)
E	excess network capacity (in kbps)
C	total number of clients
C'	total number of active clients
C_w	number of clients in CoS w
W	number of classes of service
B	number of discrete bitrates
Q	client queue capacity (in segments)
L	segment duration (in seconds)
b_i^c	current bitrate being downloaded by client c (in kbps)
q_c	current queue occupancy of client c (in segments)
w_c	CoS of client c ($\in \{1, \ldots, W\}$)
α_c	abort timeout for client c (in seconds)
β_c	abort backoff timeout for client c (in seconds)
β_c^+	success backoff timeout for client c (in seconds)
γ_c	random up-switch initiation credit for client c (in seconds)
δ_c	current segment download time for client c (in seconds)
f_c	network fill counter for client c (in bytes)
d_c	playback buffer drain counter for client c (in seconds)

Table 5.1: Rate adaptation algorithm variables.

It is assumed that all segments have the same fixed playout duration of L seconds, and that new segments are always available for download. Segment download is only inhibited by bandwidth limitations and queue fullness. A fixed playout duration results in a constant queue drain rate of one segment every L seconds, i.e., $R_{drain} = \frac{1}{L}$, while the queue fill rate is dependent on the available network bandwidth. Given a fixed network capacity N, the equal-share distributed (ESD) bandwidth per client is $\frac{N}{C}$, however, not all clients are necessarily actively downloading, e.g., if their queues are already full. Therefore, the ESD available bandwidth per active client, and thus the queue fill rate, is actually $R_{fill} = \frac{N}{C'}$, where $C' \leq C$ represents the number of clients actively downloading segments. Because C'

changes over time, the R_{fill} is a discontinuous function which depends on the current state of all the clients. This is discussed further in the network simulation section.

5.2 Rate Adaptation Algorithm

In our basic rate adaptation model, rate switches occur whenever the queue becomes full or the queue empties. We use these conditions as indication that there is either excess bandwidth, or insufficient bandwidth, respectively, to support the current bitrate. When the queue fills, i.e., $q_c = Q$, an up-switch to the next higher bitrate occurs (assuming the client is not already at the highest bitrate). When the queue empties, i.e., $q_c = 0$, a down-switch to the next lower bitrate occurs (assuming the client is not already at the lowest bitrate). Though, adjusting the queue thresholds for up-switch and down-switch and skipping bitrates on up-switch or down-switch can alter the short-term responsiveness of the rate adaptation algorithm, they do not materially affect the steady state bitrate to which clients eventually converge. We are primarily concerned with the steady state results, so for simplicity, we only consider rate switches to adjacent bitrates when the queue is either full or empty, or when a download is aborted, as discussed below.

For up-switches, we consider the suitability of the next higher bitrate, given the most recent segment download time δ_c; for down-switches, we rely on download aborts as a pre-emptive indication of an imminent empty queue condition. For segments which are taking too long, a download abort triggers a proactive down-switch. We denote the rate adaptation abort timeout for a given client c as α_c. Equations 5.1 and 5.2 show the down-switch and up-switch criteria, respectively, as defined above, where δ_c is the amount of time client c_c has spent downloading the current segment.

$$r_{down} : q_c = 0 \vee \delta_c > \alpha_c \tag{5.1}$$

$$r_{up} : q_c = Q \wedge \frac{b_{i+1}^c}{b_i^c} \cdot \delta_c < \alpha_c \tag{5.2}$$

A typical value for α_c would be: $\alpha_c = L$. In this case, up-switches occur when the queue

is full and there is sufficient bandwidth to retrieve the next higher bitrate, while down-switches occur if the queue empties or if a segment takes longer than its playout duration to download. By increasing or decreasing the value of α_c, we can alter the rate adaptation scheme to be more or less aggressive, i.e., larger α_c values imply that it is acceptable for segment downloads to take longer, while shorter α_c values require that segment downloads take less time.

5.3 Segment Download Timeouts

Proper selection of the abort timeout value α_c and a corresponding backoff timer value β_c are critical to the rate adaptation functionality. Selecting an α_c value proportional to the CoS provides CoS differentiation in independent adaptive bitrate clients. When selecting an α_c value, we maintain that $\forall i, j \in 0 \ldots C-1 : w_i > w_j \Rightarrow \alpha_i > \alpha_j$. For the purposes of this discussion, we have selected a linear mapping of CoS to α_c, however, alternate mapping functions may be used to adjust the distribution of bandwidth between classes:

$$F[w_c \rightarrow \alpha_c] : L \cdot \frac{w_c}{W} \tag{5.3}$$

Using the mapping shown in Equation 5.3, the highest CoS receives the typical value $\alpha_c = L$, while all lower classes of service receive a value of $\alpha_c < L$. In the degenerate case of a single CoS, all clients receive the typical mapping of $\alpha_c = L$. In general, the α_c value effectively acts as a coarse bandwidth distribution mechanism. For this to hold, however, in the case of a download abort, the client must wait until its next download interval begins. With respect to Equation 5.3, this amounts to a backoff delay of $L - \alpha_c$.

$$F[w_c \rightarrow \beta_c] : L - \alpha_c \tag{5.4}$$

For strict bandwidth distribution enforcement, a delay should also be inserted after a successful download, such that the client waits the full segment duration period L before attempting its next download, i.e., a delay of $L - \delta_c$. Ignoring the delay results in more

aggressive downloads, but should eventually result in a queue full condition, at which point the downloads will be naturally delayed by $L - \delta_c$. Waiting for queue to fill before rate limiting, however, has multiple side effects. First, by delaying the rate limiting, it allows lower CoS clients to gain access to a larger proportion of excess bandwidth, relative to their current bitrate. The use of abort timeouts essentially partitions bandwidth; the same partitioning is required for excess bandwidth. But, because lower CoS clients typically are retrieving a lower bitrate than higher CoS clients, lower CoS client request rates (if not limited) will be larger than higher CoS clients. Equal access to excess capacity gives lower CoS clients a larger percentage increase to their current abort timeout-based bandwidth allocation. A second side effect of the congestion caused by excessive lower CoS client requests is that it may prevent higher CoS clients from detecting and rightfully claiming excess bandwidth. The third side effect is that, allowing clients to achieve queue fullness induces an immediate up-switch in bitrate, and increases the probability of additional future up-switches. The two up-switch criteria from Equation 5.2 are queue fullness and sufficient estimated bandwidth. Allowing lower CoS clients to maintain queue fullness through aggressive downloading circumvents the initial up-switch constraint. Given the random nature of networks, there is always a non-zero probability that a burst of packets may cause a bandwidth over-estimation, resulting in an unwarranted up-switch.

To address the over aggressive successive segment download concern, we include a partial backoff after successful download which we refer to as the success backoff b_c^+. For the success backoff, we did not want to use the strict $L - \delta_c$, as that would remove flexibility and force under-utilization of the network. A logical alternative would be to use a delay of $\alpha_c - \delta_c$, forcing the client to wait at least until the end of its abort time window. This, however, can still result in rather aggressive request rates. It seems prudent to at least prevent a given client from impinging on the next higher CoS, so we added an additional $\frac{L}{W}$ factor, to maintain the inter-CoS timeout interval defined in Equation 5.3. One final constraint, however was that it does not make sense to allow the success backoff to cause requests to move out beyond the segment duration L, i.e., the highest CoS should not be delaying, so

for practical reasons, we added the following constraints to the success backoff mapping function:

$$F[w_c \rightarrow \beta_c^+] : \begin{cases} 0, & L \leq \alpha_c \\ \alpha_c - \delta_c, & L \leq \alpha_c + \frac{L}{W} \\ \alpha_c - \delta_c + \frac{L}{W}, & L > \alpha_c + \frac{L}{W} \end{cases} \tag{5.5}$$

To prevent client synchronization, we include two randomization features into our rate adaptation scheme: random up-switch initiation with latching to smooth out access to excess capacity and a progressively increasing random backoff to prevent congestion when a client experiences successive empty queue conditions.

5.3.1 Random Up-switch Initiation

Because of the discrete intervals between bitrates, fixed α_c values can cause under-utilization of the network, if the intervals between the bitrates are large. For an up-switch to take place, we can see that there must be excess bandwidth of $(b_{i+1} - b_i) \cdot C_w$, where C_w is the number of clients in class w, before the entire class can initiate an up-switch. To allow clients within a CoS to incrementally take advantage of excess capacity, we add random jitter to the α_c value to more evenly distribute up-switch requests. Equation 5.7 augments the calculation of α_c (from Equation 5.3) to include the random jitter. We select a uniformly distributed random value γ_c in the range $[0, \frac{L}{W}]$, where $\frac{L}{W}$ corresponds to the inter-CoS interval used by α_c in Equation 5.3. Maintaining the inter-CoS boundary when selecting γ prevents class priority inversion.

$$\gamma_c = RAND\left(\frac{L}{W}\right) \tag{5.6}$$

$$F[w_c \rightarrow \alpha_c] : L \cdot \frac{w_c}{W} + \gamma_c \tag{5.7}$$

The purpose of the random up-switch initiation credit γ_c is to address the condition where the excess network capacity E is greater than the bandwidth needed for one client to

up-switch, but less than the bandwidth needed for all clients to up-switch, i.e., $(b_{i+1} - b_i) < E < (b_{i+1} - b_i) \cdot C_w$. The random value gives each client within the CoS an equal probability of acquiring some of the excess bandwidth. An issue arises, however, when the γ_c value is not consistent and causes the client to oscillate between bitrates. Bitrate thrashing results in an increase in badput which leads to a decrease in goodput. To address this issue, we introduced a latching function, whereby the client uses the same γ_c value until an abort occurs.

5.3.2 Random Backoff

When an empty queue condition occurs and the client is at the lowest bitrate, continued download failures imply a serious network issue. In such cases, continued segment download attempts will only compound the network issue and reduce goodput. To address this, we apply a CoS-weighted progressively increasing backoff scheme. Equation 5.8 augments the calculation of β_c (from Equation 5.4) to include the progressively increasing backoff. We select a uniformly distributed random value in the range $[0, \mu_c \cdot (W - w_c + 1)]$, where μ_c is the consecutive empty queue condition counter and $(W - w_c + 1)$ is the CoS weight.

$$F[w_c \rightarrow \beta_c] : (L - \alpha_c) + L \cdot RAND(\mu_c \cdot (W - w_c + 1)) \tag{5.8}$$

At this point, we also need to take into consideration the random up-switch initiation value γ_c, described in the previous section. Because the base β_c mapping component for aborts: $(L - \alpha_c)$, contains α_c, and α_c contains the random up-switch initiation value γ_c, β_c needs to now compensate for any badput caused by the γ_c overage. We rectify this by adding in an additional $2 \cdot \gamma_c$ into Equation 5.9:

$$F[w_c \rightarrow \beta_c] : (L - \alpha_c) + 2 \cdot \gamma_c + L \cdot RAND(\mu_c \cdot (W - w_c + 1)) \tag{5.9}$$

One of the γ_c values added in is to offset the γ_c component of α_c. The other γ_c value that is added in is a penalty for consuming excess bandwidth prior to the abort. The abort will

have occurred after waiting and downloading for an additional γ_c seconds beyond the CoS limit. We allow the other clients to reclaim that bandwidth by enforcing the penalty.

5.4 Rate Adaptation Callback Procedures

Using the down-switch and up-switch rules defined above, we can specify the basic rate adaptation algorithm procedures, as shown in Procedures 5.1-5.5. Procedure 5.1 is a simple callback function which sets a download abort timer at the start of a new segment download and selects a new random up-switch initiation credit.

Procedure 5.1 *SegmentDownloadStart*()

inputs
  $c \in \{0 \ldots C-1\}$
do
  $abortTimer \leftarrow \alpha_c$
  if $\gamma_c = 0$ **then**
    $\gamma_c \leftarrow RAND\left(\frac{L}{W}\right)$
  end if

Procedure 5.2 *SegmentDownloadComplete*()

inputs
  $c \in \{0 \ldots C-1\}$
do
  $q_c \leftarrow q_c + 1$
  $abortTimer \leftarrow \emptyset$
  if $q_c = Q$ **then**
    if $\frac{b_{i+1}^c}{b_i^c} \cdot \delta_c < \alpha_c$ **then**
      $b_i^c \leftarrow b_{i+1}$
    end if
  else
    $backoffTimer \leftarrow \beta_c^+$
  end if

Procedure 5.2 shows the segment download completion callback. When a segment download completes successfully, the queue occupancy is incremented and the rate adaptation

algorithm checks to see if an up-switch is warranted, i.e., it checks to see if the new segment causes the queue to fill and if the current bandwidth could support the next higher bitrate. If an up-switch is warranted, the client bitrate is updated for the next download. Otherwise, if the queue is not yet full, the algorithm sets the success backoff timer before beginning download of the next segment.

Procedure 5.3 *AbortTimeoutExpired*()

inputs
  $c \in \{0 \ldots C-1\}$
do
  AbortCurrentDownload(c)
  $\gamma_c \leftarrow 0$
  if $b_{i-1}^c > 0$ **then**
    $b_i^c \leftarrow b_{i-1}^c$
  end if
  $backoffTimer \leftarrow \beta_c$

Procedure 5.3 shows the abort timeout expiration callback. If the segment download abort timer set in Procedure 5.1 expires, implying that $\delta_c > \alpha_c$, then the current segment download is aborted, the random up-switch initiation credit is reset, a bitrate down-switch is immediately initiated, and a backoff timer is set, to pace the start of the next segment download. In the case where the client is not already at the lowest bitrate, the backoff timer simply waits for a per-segment backoff period, in the hope that lowering the bitrate will have a sufficient impact. However, in the case where a lower bitrate is not available, the backoff timer uses a progressively increasing backoff scheme.

Procedure 5.4 shows the segment playout completion callback. When a segment completes playout, the queue is decremented making room for a new segment. Unlike other rate adaptation evaluations which assume infinite player buffer sizes, we track the segment-based queue length so that we can properly model inter-client distribution of burst throughput capacity, which is highlighted in our results. After draining the oldest segment from the queue, a queue empty check is performed. If the segment drained was the last segment and a download is currently active, an abort timeout is immediately triggered, forcing a bitrate

Procedure 5.4 *SegmentPlayoutComplete*()

inputs
  $c \in \{0 \ldots C-1\}$
do
  $q_c \leftarrow q_c - 1$
  if $q_c = 0$ **then**
    if $abortTimer \neq \emptyset$ **then**
      $AbortTimeoutExpired(c)$
    end if
  else if $q_c = Q - 1$ **then**
    $StartNextDownload(c)$
  end if

down-switch and backoff as described in Procedure 5.3. Otherwise, if the segment drained causes the queue to no longer be full, the algorithm immediately begins download of the next segment.

Procedure 5.5 *BackoffTimerExpired*()

inputs
  $c \in \{0 \ldots C-1\}$
do
  $backoffTimer \leftarrow \emptyset$
  $StartNextDownload(c)$

Finally, Procedure 5.5 shows the backoff timer expiration callback. Whenever a segment abort or empty queue condition occurs, the backoff timer is set to help reduce congestion in the network. When the backoff timer expires, the algorithm immediately begins download of the next segment.

5.5 Rate Distribution Numerical Estimation

To model the steady-state distribution of bandwidth between classes of service, we developed the function $G(x)$ to determine the discrete target bitrate for a given CoS. The function $G(x)$, shown in Equation 5.10, applies the bitrate floor function to the average bandwidth per client function $g(x)$. The bitrate floor function selects the highest bitrate b' that does

not exceed the calculated average bandwidth, i.e., $\exists i \in 0 \ldots B-1 : \forall j \in 0 \ldots B-1 : b_i < g(x) \wedge b_j > b_i \Rightarrow b_j > g(x)$.

$$G(x) = \lfloor g(x) \rfloor_b \tag{5.10}$$

The average bandwidth per client function $g(x)$, shown in Equation 5.12, starts with the weighted average bandwidth per client $\frac{x}{W} \cdot \frac{N}{C}$, where $\frac{x}{W}$ corresponds to the α_c weighting defined in Equation 5.3, and adds to that, the unused bandwidth of all lower classes of service. The bandwidth reclamation function $g'(x)$, shown in Equation 5.11, normalizes the unused average bandwidth $(1 - \frac{x}{W}) \cdot \frac{N}{C}$ by $\frac{C_i}{\sum_{j=i+1}^{W-1} C_j}$ to distribute it across only those clients in higher classes of service.

$$g'(x) = \sum_{i=0}^{x-1} \frac{(1 - \frac{x}{W}) \cdot \frac{N}{C} \cdot C_i}{\sum_{j=i+1}^{W-1} C_j} \tag{5.11}$$

$$g(x) = \begin{cases} \frac{x}{W} \cdot \left(\frac{N}{C} + g'(x)\right), & x > 0 \\ \frac{x}{W} \cdot \frac{N}{C}, & x = 0 \end{cases} \tag{5.12}$$

Because the bandwidth reclamation function $g'(x)$ also applies the weighting condition $(1 - \frac{x}{W})$, excess unused bandwidth will still exist in the network. Applying the random jitter to initiate rate up-switches helps to utilize this excess capacity, however, the random nature of its distribution makes it hard to quantify numerically. Our simulation results verify the numerical results with deterministic up-switches and then show the throughput benefits of applying random up-switches.

5.6 Multi-Client Rate Adaptation Simulation Environment

In order to accurately model the network interactions of HTTP adaptive streaming clients, we use a network model where the network capacity N is divided into packet chunks of size P, distributed randomly between actively downloading clients in each one second time quanta. Though our rate adaptation algorithm works at the segment level, the ability of

the network model to accurately reflect segment completion is a critical component. If at any point there are no actively downloading clients, because each client is either backing off or has a full queue, then the remaining unallocated bandwidth for that time quanta is marked as unused.

Procedure 5.6 describes the simulation methodology, where given a simulation duration T, a fixed number of clients C, a fixed network capacity N, and a fixed network packet size P, the simulator loops through each time quanta and performs per-client upkeep operations, as well as distributes download bandwidth on a per-packet basis. Each client is modeled with a segment drain counter d_c measured in seconds, and a segment fill counter f_c measured in bits. For each time quanta, drain counters are updated and segment playout completion callbacks are triggered as necessary. For each network packet, a fill counter is updated and segment download completion callbacks are triggered as necessary. The abort timeout is also checked in the network packet loop. This allows δ_c to achieve a sub-second resolution, for more precise modeling. The simulation keeps track of network usage statistics for unused bandwidth, goodput, and badput (bandwidth wasted on aborted segment downloads).

5.6.1 Simulation Configuration

For the results presented in this section, we use a fixed number of clients $C = 30$, the standard HLS target segment duration of $L = 10$ seconds, and the Apple suggested standard definition bitrates of: $\{150, 240, 440, 640, 1240\}$ [97]. We use the Apple recommended bitrates given their status as the de facto standard. As mentioned previously, the Apple recommended bitrates are not evenly distributed, nor are they distributed in any mathematically uniform manner. Rate switch inflection points are directly affected by the distribution of bitrate intervals, as the bitrate interval directly impacts the up-switch bandwidth requirement $(b_{i+1} - b_i) \cdot C_w$.

For each test case, we ran simulations, varying the network capacity over the range [500 : 40000] kbps, using increments of 500 kbps. We chose the range [500 : 40000] so that we could evaluate both the high and low edge conditions, i.e., where $\frac{N}{C} < b_0$ and

Procedure 5.6 Network Simulation Model

```
do
  unused ← 0
  goodput ← 0
  badput ← 0
  t ← 0
  while t < T do
    for all c ∈ 0...C − 1 do
      if q_c ≠ 0 then
        d_c ← d_c + 1
        if d_c mod L = 0 then
          SegmentPlayoutComplete(c)
        end if
      end if
      if backoffTimer ≠ ∅ then
        backoffTimer ← backoffTimer − 1
        if backoffTimer = 0 then
          BackoffTimerExpired(c)
        end if
      end if
    end for
    n ← 0
    while n < N do
      if ∀c ∈ 0...C − 1 : q_c = Q ∨ backoffTimer ≠ ∅ then
        unused ← unused + (N − n)
        n ← N
      else
        c ← RAND(C')
        f_c ← f_c + P
        if f_c ≥ b_i^c · L then
          goodput ← goodput + f_c
          f_c ← 0
          SegmentDownloadComplete(c)
        else if δ_c > α_c then
          badput ← badput + f_c
          f_c ← 0
          AbortTimeoutExpired(c)
        end if
        n ← n + P
      end if
    end while
    t ← t + 1
  end while
```

where $\frac{N}{C} > b_{B-1}$. Each simulation was run for $T = 100000$ seconds. The simulation period of 100000 seconds ($\approx$ 27.78 hours) is more than sufficient for achieving a steady state condition from which to evaluate the rate adaptation algorithm's performance. In all cases, we start each client with a full queue, i.e., $q_c = Q$, to simulate a "good" steady state starting condition. In most cases, we also start each client at the highest bitrate, i.e., $b_i^c = b_{B-1}$. The exceptions are the numerical estimation case, where we start each client at the estimated bitrate, i.e., $b_i^c = G(w_c)$, and the bounded manifest file case where we start each client at the maximum bitrate for that client, i.e., $b_i^c = b_{B-1-W+w_c}$.

5.6.2 Simulation Evaluation Metrics

We use three primary metrics to evaluate network performance:

- Throughput: comparing goodput vs. badput vs. unused bandwidth at different network capacities.
- Average client bitrate: comparing the average bitrate of clients within a given CoS, at different network capacities.
- Average number of rate switches: comparing the average number of rate switches per client, within a given CoS, at different network capacities.

Using these metrics, we can evaluate the effectiveness of our rate adaptation algorithm to implement CoS differentiation and compare it to other methods for implementing CoS differentiation.

5.7 Multi-Client Rate Adaptation Simulation Results

Using the methodology described above, we ran various simulations with different numbers of classes of service, $W \in \{1, 3, 5\}$, to verify the basic functionality of our scheme. We also performed a series of experiments to show the incremental effects of the different features that were added to the base scheme. We then compared the simulation results for our

rate adaptation algorithm implementation, with our numerical estimates and an alternate manifest-based CoS enforcement scheme. The following sections provide a view of the efficiency of the existing algorithm, as well as opportunities for customization and further tuning.

5.7.1 Multi-Client Rate Adaptation CoS Differentiation

Our first test was to confirm that CoS differentiation could really be achieved through the intelligent abort timeout mechanism defined in our rate adaptation algorithm. For this comparison, we performed test runs with $W = 1$, which corresponds to a typical group of non-CoS-aware clients, $W = 3$, which corresponds to a non-atypical even distribution of clients across three classes of service ("Gold", "Silver", and "Bronze"), and $W = 5$, which represents an even distribution of clients across five classes of service ("Gold", "Silver", "Bronze", "Stone", and "Wood"). In all three test runs, random backoff and random up-switch initiation with latching were enabled.

Figures 5-1 (a)-(c) show the average bitrate played out by clients within a given CoS, for the $W = 1$, $W = 3$, and $W = 5$ cases, respectively. The encoded video bitrates are shown as dotted horizontal lines, for reference. The equal-share distributed (ESD) bitrate of $\frac{N}{C}$ is also plotted, as a reference point for unconditional fair access. In Figure 5-1 (a), the single CoS clients compete for bandwidth, each with equal priority. As expected, the average bitrates asymptotically approach the ESD line as it crosses each encoded bitrate. We refer to these points as the encoded bitrate inflection points, i.e., where $\frac{N}{C} = b_i$.

In Figures 5-1 (b) and (c), we can see that the average bitrates for the different classes of service are clearly differentiated. The gold level clients are able to acquire more than their equal share, as can be seen by the gold plot being consistently above the ESD line. In the 5 CoS case, silver level clients also tend to get more than their equal share, though, in general, less than gold level subscribers. To compensate, the other classes of service get less than an equal share. Bronze, stone, and wood level clients never reach the ESD line, and are not able to attain bitrates higher than the next higher class of service. Silver level

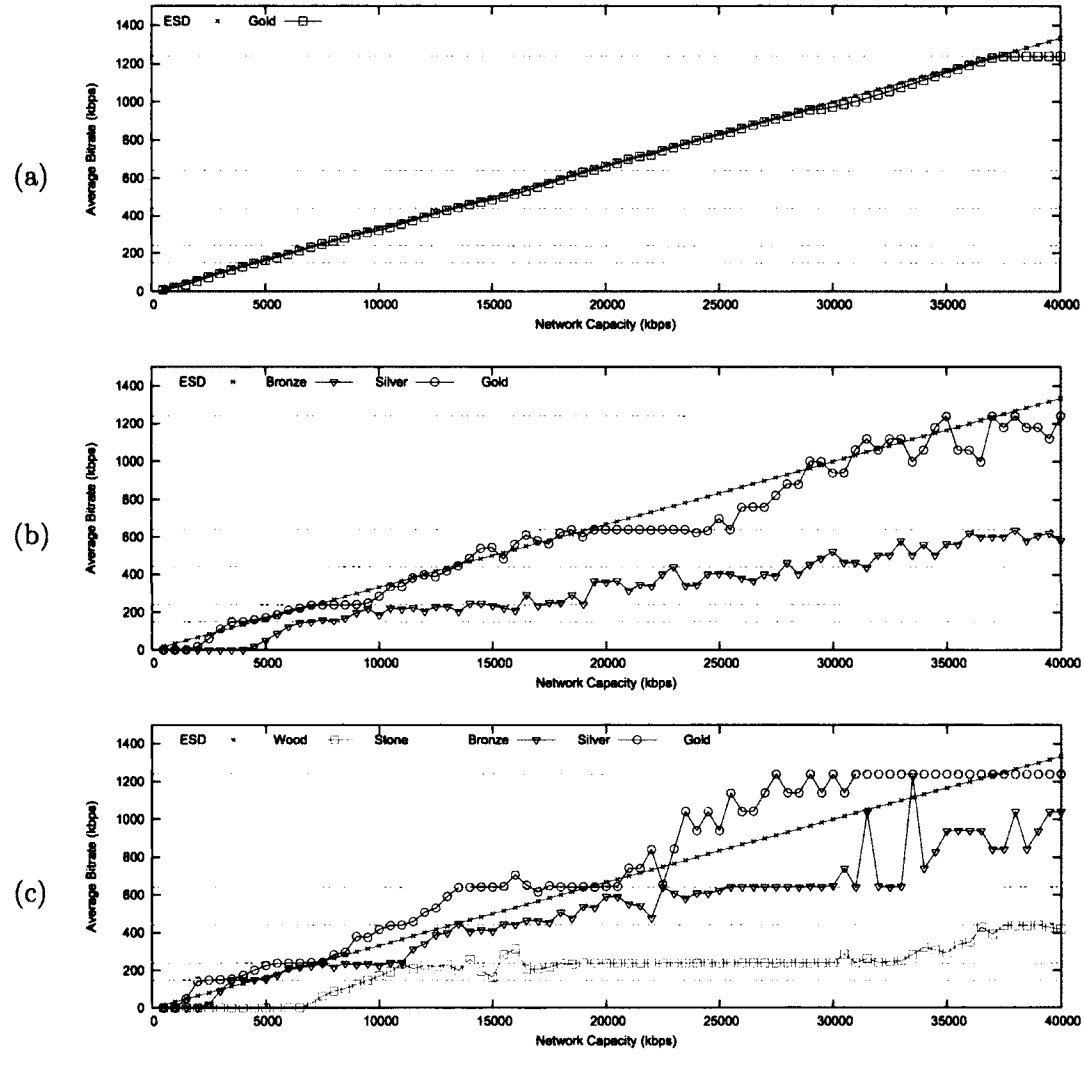


Figure 5-1: Average client bitrate for: (a) single Cos ($W = 1$), (b) multiple CoS ($W = 3$), and (c) multiple CoS ($W = 5$).

clients, in general get less bandwidth than gold level clients. Bronze level clients, in general get less bandwidth than silver level clients. Stone level clients, in general get less bandwidth than bronze level clients. Wood level clients, in general get less bandwidth than stone level clients. Lower CoS clients are still able to attain higher bitrates, they just require higher levels of excess bandwidth availability before they may do so.

As we have discussed, our download abort scheme does result in some amount of excess capacity being unused, which we will see in Figures 5-2 (a)-(c). As such, any client, of any CoS, may gain access to that excess capacity. The strict interval timeouts and backoff schemes are designed to prevent priority inversion. However, there is always a non-zero probability that a lower CoS client can find gaps in the segment downloads through which it can successfully download higher bitrate content. We can see such points in Figure 5-1 (c). Continued investigation into enforcement of excess bandwidth allocations provides an interesting topic for future research.

Figures 5-2 (a)-(c) show the network throughput for the $W = 1$, $W = 3$, and $W = 5$ cases, respectively. In the single CoS case, shown in Figure 5-2 (a), we can see that goodput is very high, though some badput does exist between bitrate inflection points. Increased competition for network resources results in segment download aborts when the ESD bitrate does not match an encoded video bitrate. In the multiple CoS cases, shown in Figures 5-2 (b) and (c), we can see fairly good utilization, though, under-utilization does occur when lower classes of service are yet unable to switch to the next higher bitrate. This is exacerbated by the super-linear increases in the intervals between bitrates, as larger amounts of excess capacity are required before up-switches can occur.

Though we show both the $W = 3$ and $W = 5$ cases to demonstrate the effectiveness of abort timeouts, it should be noted that large W values are not necessarily practical. Existing network protocols, e.g., Ethernet, MPLS, and TCP, use three bit fields for signaling priority. With respect to segment download aborts and our linear mapping of abort timeouts to CoS (see Equation 5.3), a large W value limits the elasticity of the time range, especially with smaller segment durations L. Use of alternate abort timeout mapping schemes, as well as

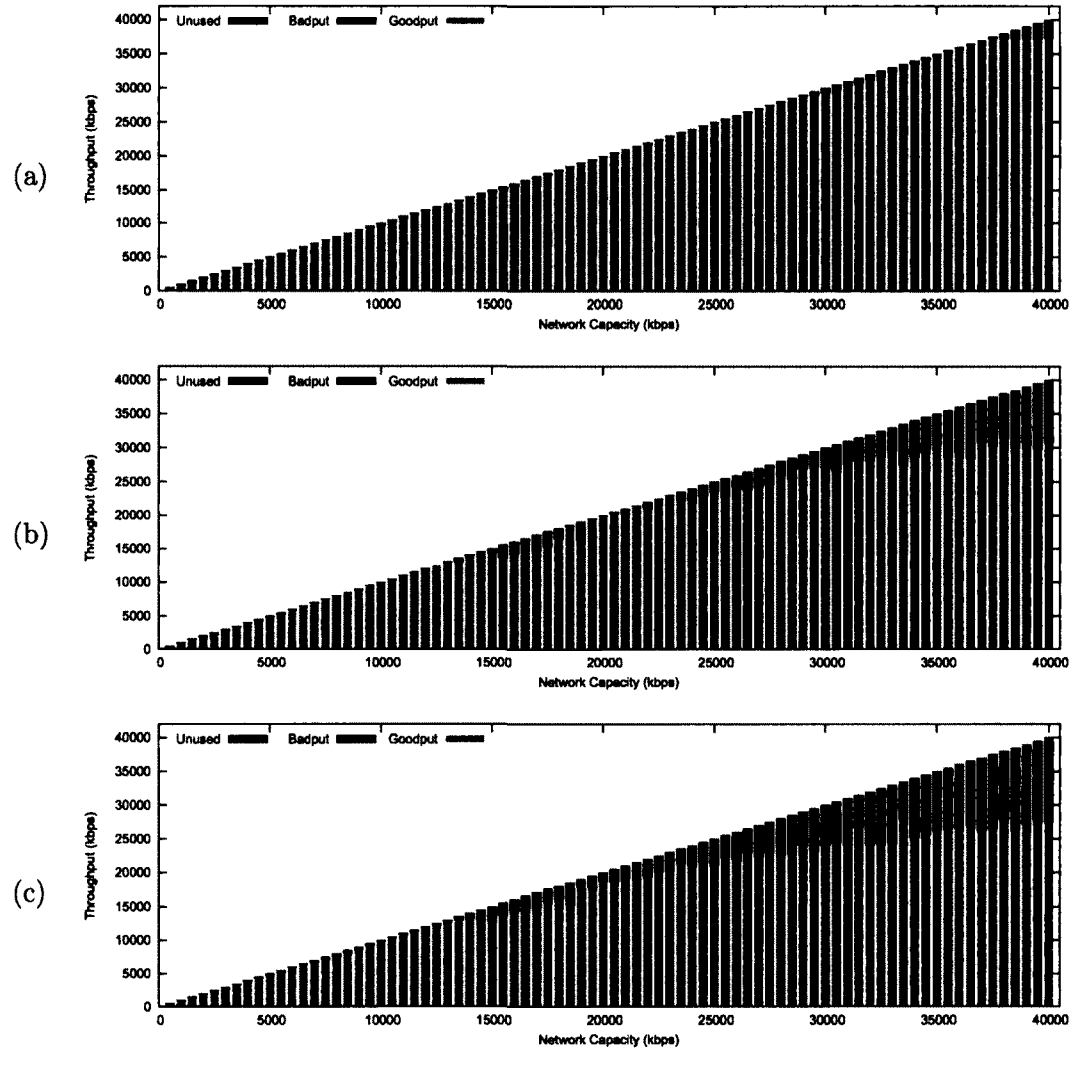


Figure 5-2: Network throughput for: (a) single CoS, (b) multiple CoS $W = 3$, and (c) multiple CoS $W = 5$.

overlapping abort timeout ranges are another interesting topic for future research.

5.7.2 Random Backoff and Up-switch Initiation Performance

In our rate adaptation algorithm, we discussed two randomization features: random backoff and random up-switch initiation, with and without latching. The random backoff was designed to help reduce congestion in low network capacity conditions where empty queue conditions are prevalent. The random up-switch initiation was designed to help increase utilization in high network capacity conditions where large inter-bitrate gaps require a lot of excess capacity before an entire CoS can increase their bitrate. Latching was also incorporated with random up-switch initiation to prevent bitrate thrashing. To show the impact of each of these features, we performed multiple simulations, progressively adding on features. We started with no random backoff and no random up-switch initiation, then added random backoff, then added random up-switch with no latching, and finally added random up-switch with latching. Figures 5-3 (a)-(d) show the overall network utilization as a percentage, for the progression of the four scenarios, respectively, in the $W = 3$ case. Figures 5-4 (a)-(d) show the overall network utilization as a percentage, for same four scenario progression, in the $W = 5$ case.

In Figures 5-3 (a) and 5-4 (a), we can see that at the low end of the network capacity spectrum there is 100% badput as all clients compete for bandwidth, but none are able to complete any downloads. At the high end of the network capacity spectrum in Figures 5-3 (a) and 5-4 (a), there is a lot of unused bandwidth, as there is not enough excess bandwidth for an entire CoS to up-switch to the next bitrate. With Figures 5-3 (b) and 5-4 (b), the addition of random backoff clearly helps the low end utilization by reducing congestion. There are no longer any cases of 100% badput, and goodput quickly rises to ~80%.

In Tables 5.2 and 5.3 we can see comparisons of the overall throughput across all the simulated network capacities. The first row of each table shows the average goodput, badput, and unused bandwidth percentages which correspond to the data show in Figures 5-3 (a) and 5-4 (a). The subsequent rows show the average goodput, badput, and unused

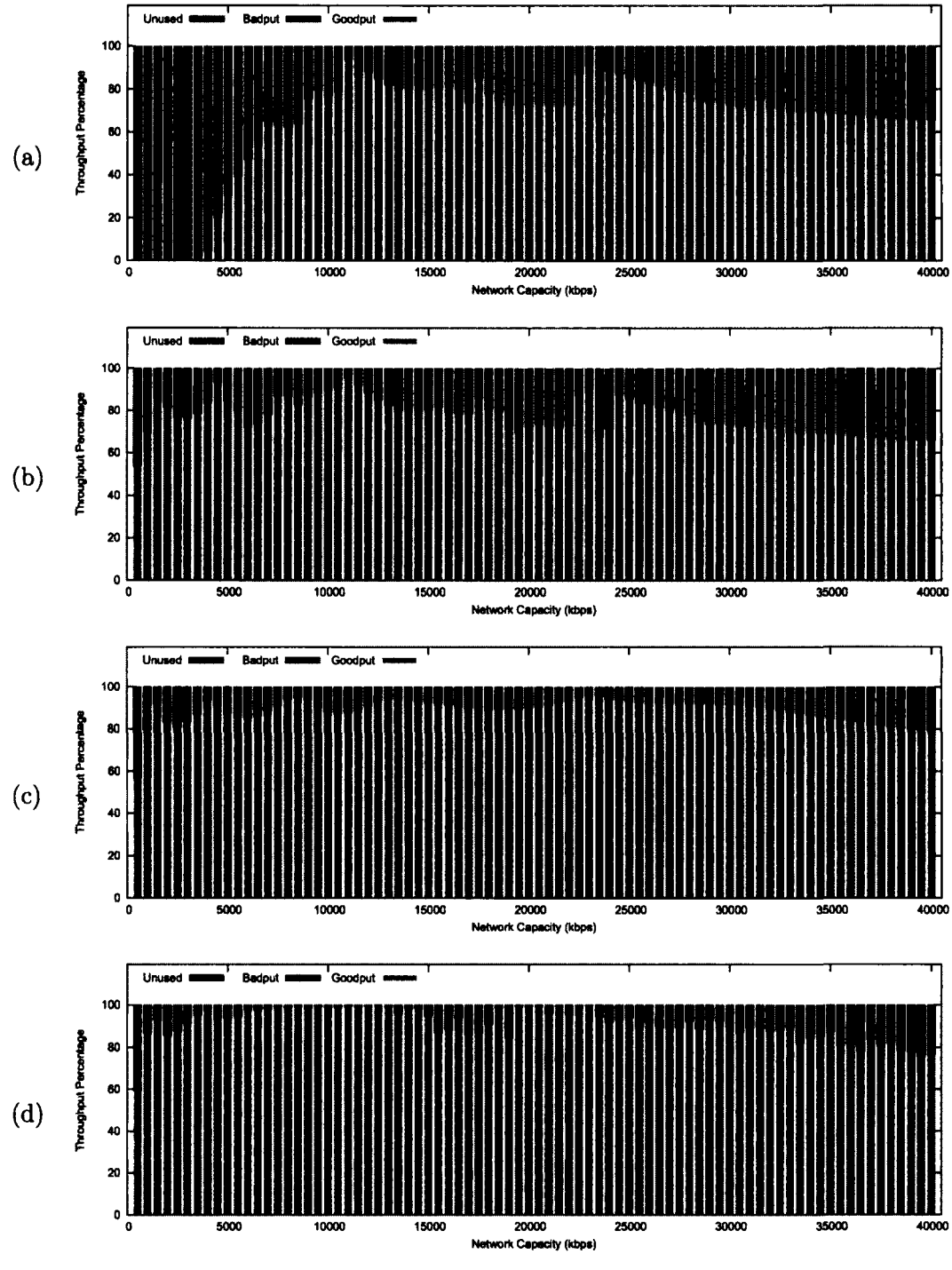


Figure 5-3: Network throughput percentage for: (a) no random backoff, random up-switch initiation, or latching, (b) with random backoff, but no random up-switch initiation or latching, (c) with random backoff and random up-switch initiation, but no latching, and (d) with random backoff, random up-switch initiation, and latching, $W = 3$.

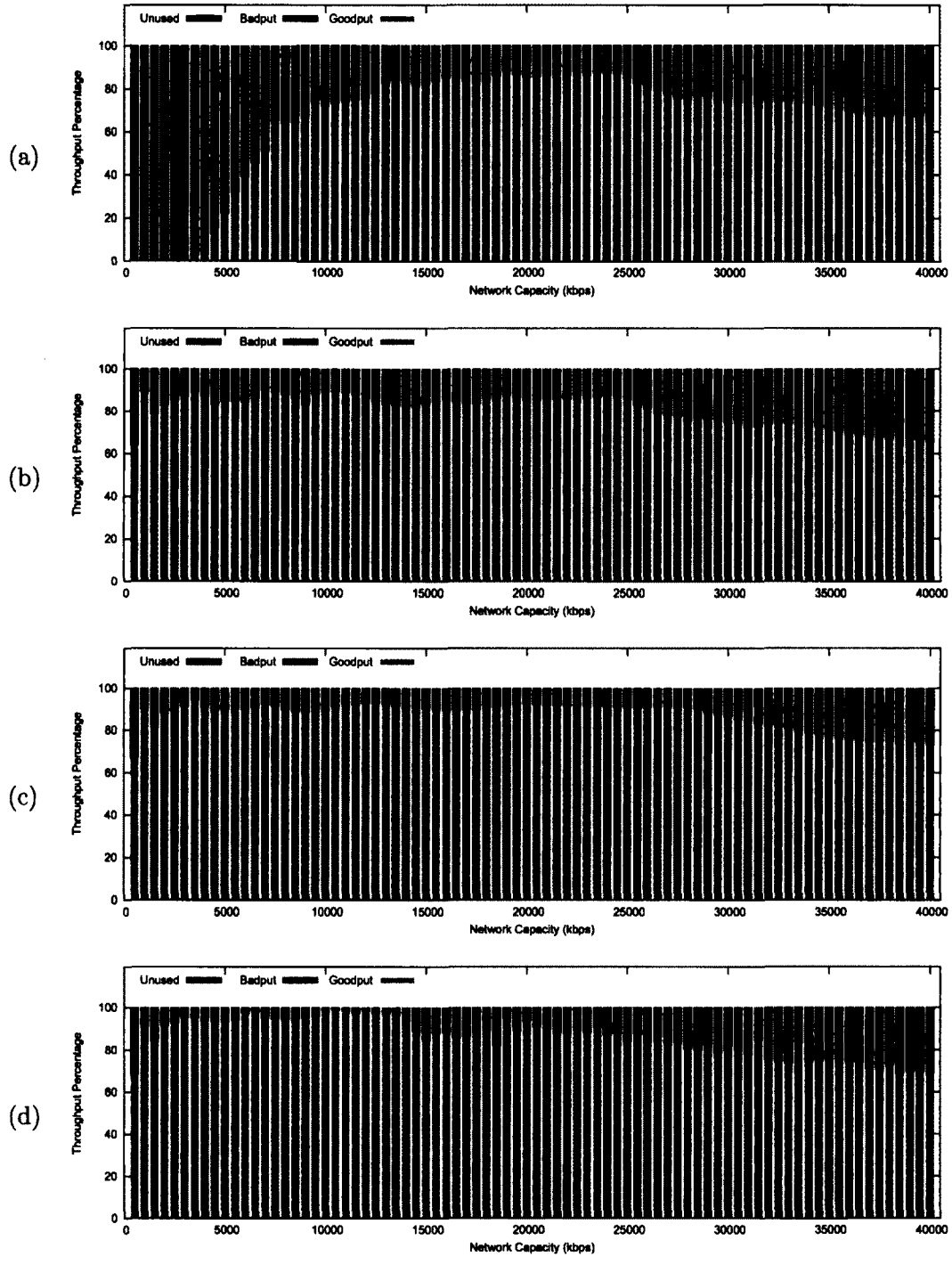


Figure 5-4: Network throughput percentage for: (a) no random backoff, random up-switch initiation, or latching, (b) with random backoff, but no random up-switch initiation or latching, (c) with random backoff and random up-switch initiation, but no latching, and (d) with random backoff, random up-switch initiation, and latching, $W = 5$.

	Goodput	*Badput*	*Unused*
no random backoff, no random up-switch initiation, and no latching	65.14	22.64	12.22
with random backoff, but no random up-switch initiation, and no latching	76.88 (+18.03%)	10.23 (-54.81%)	12.89 (+5.48%)
with random backoff, with random up-switch initiation, but no latching	88.42 (+35.75%)	8.65 (-61.81%)	2.93 (-76.01%)
with random backoff, with random up-switch initiation, and with latching	91.02 (+39.74%)	1.89 (-91.65%)	7.09 (-41.99%)

Table 5.2: Throughput percentage comparison for random backoff, random up-switch initiation, and latching, $W = 3$.

bandwidth percentages, as well as the percentage differences compared to the first row, i.e., the comparison of Figures 5-3 (b)-(d) and 5-4 (b)-(d) to Figures 5-3 (a) and 5-4 (a). We can see that the addition of random backoff improves goodput percentage by 18% and 22%, while reducing badput percentages by almost 55% and 60%, for the $W = 3$ and $W = 5$ cases, respectively.

In Figures 5-3 (c) and 5-4 (c), with the further addition of random up-switch initiation without latching, we can see a significant improvement in the mid-range goodput, as well as better utilization at the high end of the network capacity spectrum. We can see from Tables 5.2 and 5.3 that with random up-switch initiation goodput percentage increases an additional 17% and 9% beyond the gains observed from random backoff, in the $W = 3$ and $W = 5$ cases, respectively. In some cases, however, goodput comes at the cost of higher badput. Though, overall, the average badput percentage is reduced with random up-switch initiation, it is accompanied by a significant reduction in unused bandwidth which reduces the elasticity of the network. The selection of new random γ_c values for each segment introduces the possibility of bitrate thrash, where a given γ_c value is sufficient for up-switching, but a subsequent γ_c value is no longer sufficient for sustaining the higher bitrate.

Latching provides a solution to the thrashing issue, as can be seen in Figures 5-3 (d) and 5-

	Goodput	*Badput*	*Unused*
no random backoff, no random up-switch initiation, and no latching	65.79	23.87	10.34
with random backoff, but no random up-switch initiation, and no latching	80.29 (+22.03%)	9.58 (-59.89%)	10.14 (-1.93%)
with random backoff, with random up-switch initiation, but no latching	86.67 (+31.73%)	8.21 (-65.61%)	5.12 (-50.43%)
with random backoff, with random up-switch initiation, and with latching	85.83 (+30.46%)	2.48 (-89.59%)	11.68 (+13.02%)

Table 5.3: Throughput percentage comparison for random backoff, random up-switch initiation, and latching, $W = 5$.

4 (d). Latching reduces the number of aborts, decreasing badput percentage and improving goodput percentage over the entire network capacity spectrum. The last rows of Tables 5.2 and 5.3 show a 90% reduction in badput percentage with the use of latching with random up-switch initiation. We can also see that the unused bandwidth percentage more than doubles when adding latching to random up-switch initiation. In the $W = 5$ case, the goodput percentage actually decreases slightly due to the latching of sub-optimal random γ_c values.

Figures 5-3 (c) and (d), as well as Figures 5-4 (c) and (d), show the trade of between aggressive up-switching and the use of latching to prevent bitrate thrashing. For the purposes of this discussion, we have chosen a very simple random up-switch formula, i.e., $RAND\left(\frac{L}{W}\right)$, which maintains the CoS boundary, and a very simple latching model, i.e., hold until abort, to demonstrate the effect of these rate adaptation algorithm features. Further investigation into the impact of random γ_c value distributions is an interesting area for future research.

5.7.3 Numerical Estimation Comparison

In order to better simulate real-world network uncertainty, we introduced randomization into the bandwidth distribution, in our simulations. Furthermore, to prevent client syn-

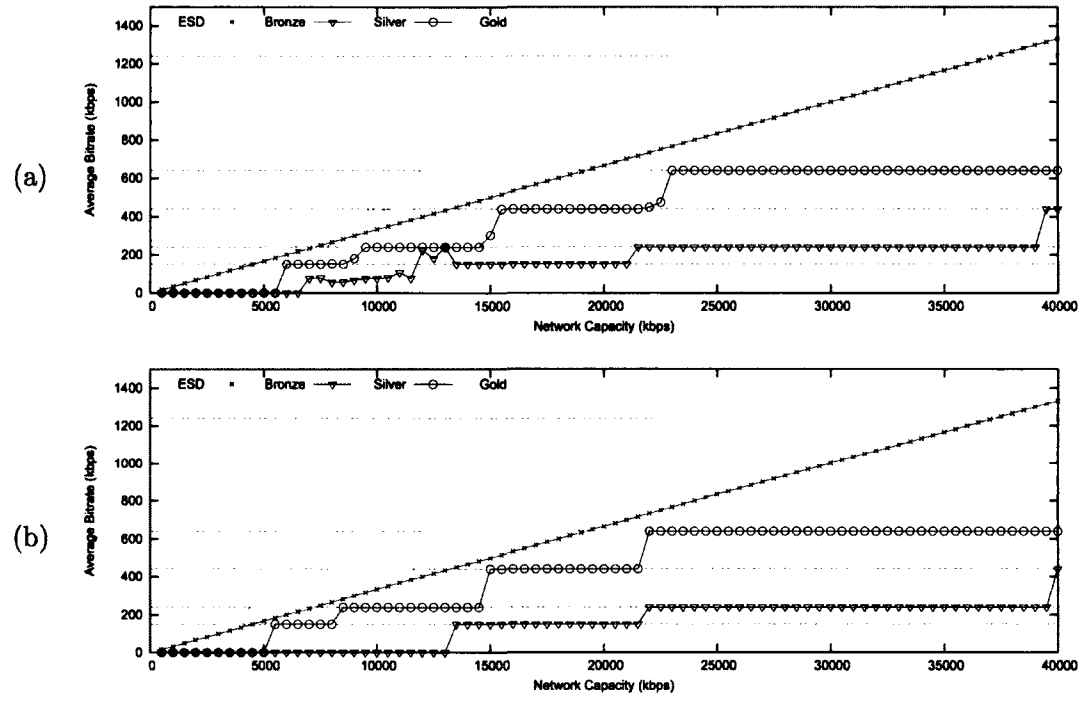


Figure 5-5: Average client bitrate for: (a) round robin bandwidth distribution vs. (b) $G(x)$ numerical estimate, $W = 3$.

chronization, we introduced random jitter to our rate adaptation algorithm. To confirm that our numerical estimation closely approximates our rate adaptation algorithm implementation, we added a deterministic bandwidth distribution mode to our simulator which allocates bandwidth uniformly between active clients. We compared the $W = 3$ deterministic simulation results with the numerical estimate. In the deterministic simulation case, both random backoff and random up-switch initiation were disabled.

Figures 5-5 (a) and (b) show the average bitrate played out by clients for the deterministically simulated and numerically estimated $W = 3$ cases, respectively. We can see from Figure 5-5 (b), that the numerical estimate $G(x)$ generates a uniform step function, as expected. The up-switch points in the deterministic simulation results, shown in Figure 5-5 (a), correspond fairly well to the numerical estimate up-switch points shown in Figure 5-5 (b). The primary exception is in the low network capacity cases, where competition for bandwidth causes aborts and high badput prevents clients from playing. The

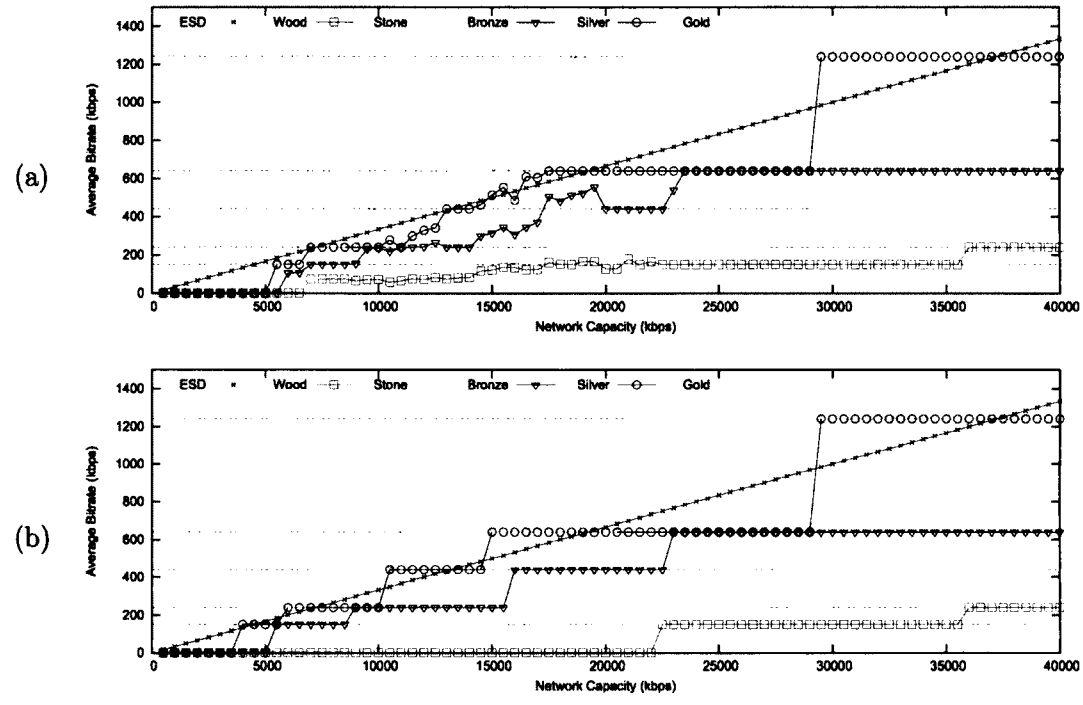


Figure 5-6: Average client bitrate for: (a) round robin bandwidth distribution vs. (b) $G(x)$ numerical estimate, $W = 5$.

bronze level clients have no clear idea of when sufficient bandwidth exists, so they attempt to retrieve segments even when only partial playback is possible. In this situation, a bandwidth capping scheme, as described in Chapter 4 would be a good solution, to be used in combination with the CoS-aware rate adaptation scheme, to provide access control and prevent lower CoS download attempts.

Figures 5-6 (a) and (b) show the average bitrate played out by clients for the deterministically simulated and numerically estimated $W = 5$ cases, respectively. Just as with Figure 5-5 (b), we can see the expected step function in Figure 5-6 (b), generated by the numerical estimate $G(x)$. Again, the up-switch points in the deterministic simulation results, shown in Figure 5-6 (a), correspond fairly well to the numerical estimate up-switch points shown in Figure 5-6 (b). Similar to the $W = 3$ case, the low network capacity cases do not correlate as well, due to the competition for bandwidth and the lack of access control on the lowest CoS, in this case wood.

5.7.4 Manifest-based CoS Differentiation Comparison

In evaluating abort timeout-based CoS differentiation, we compare our scheme to an existing methods for implementing CoS differentiation. The only client-side method of performing CoS differentiation, that we are aware of, is to limit the bitrates advertised in the manifest file. The obvious disadvantage to such an approach is the inability to react dynamically to changes in network conditions, specifically, increases in network capacity. We start by simulating unbounded manifest files with no CoS differentiation, then show the effect of using a bounded manifest file, and finally compare it with our download abort-based CoS enforcement scheme. Figures 5-7 (a)-(c) show the distribution of goodput between classes of service for each of the three scenarios, respectively, in the $W = 3$ case. Figures 5-8 (a)-(c) show the show distribution of goodput between classes of service for the same three scenarios, but for the $W = 5$ case. In all six test runs, random backoff and random up-switch initiation with latching were enabled.

Figure 5-7 (a) shows a simulation with three groups of clients: A, B, and C, all with the same CoS. As expected, the distribution of bandwidth between clients, regardless of CoS, is fairly even. In Figure 5-7 (b), the three groups of clients are assigned different classes of service and special manifest files are generated for clients of each CoS. The different manifest files advertise a different set of bitrates, where the maximum bitrate advertised in the manifest file $b_{max} = b_{B-1-W+w_c}$ is successively lower for each CoS. We can see that as each CoS hits its maximum bitrate, the goodput levels off and under-utilization of the network occurs. Figure 5-7 (c), shows the bandwidth distribution for our intelligent download abort scheme. We can clearly see the differentiation in bandwidth distribution between the three classes of service, however, there is no hard cap on the total network throughput, as in the bounded manifest case. Our scheme, shown in Figure 5-7 (c), is able to provide CoS differentiation, unlike the single CoS case, shown in Figure 5-7 (a), and it provides better network utilization than the bounded manifest file case, shown in Figure 5-7 b. Table 5.4 shows the aggregate utilization percentages across all network capacities corresponding to Figures 5-7 (a)-(c). We can see that the single CoS case clearly has

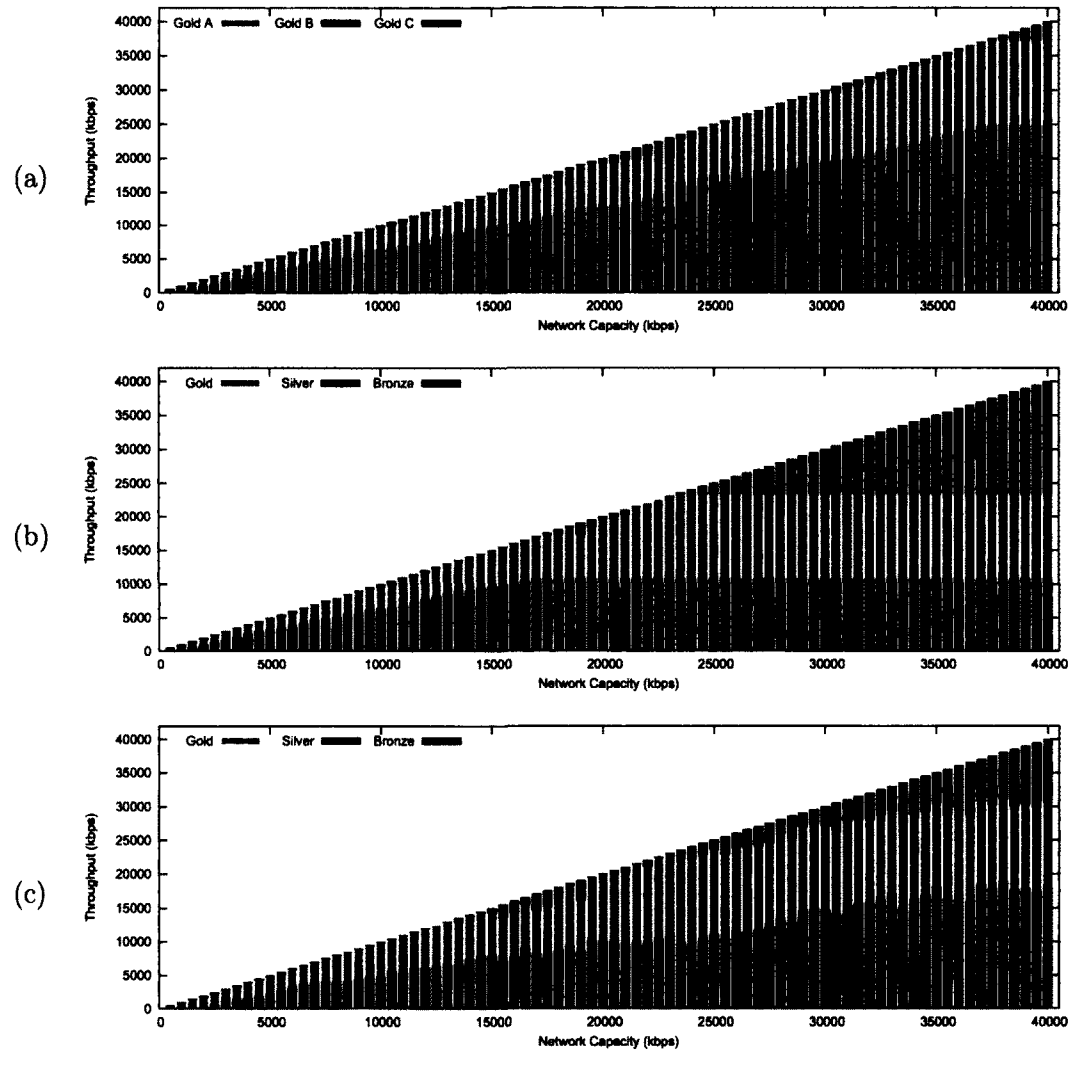


Figure 5-7: Per-CoS goodput distribution for: (a) single CoS unbounded manifest files, (b) per-CoS bounded manifest files, and (c) multiple CoS unbounded manifest files, $W = 3$.

	Bandwidth Usage
single CoS, unbounded manifest file	97.36%
per-CoS bounded manifest file	81.19%
multiple CoS, unbounded manifest file	88.96%

Table 5.4: Bandwidth usage percentage comparison for single CoS unbounded manifest files, per-CoS bounded manifest files, and multiple CoS unbounded manifest files, $W = 3$.

	Bandwidth Usage
single CoS, unbounded manifest file	94.89%
per-CoS bounded manifest file	63.96%
multiple CoS, unbounded manifest file	81.77%

Table 5.5: Bandwidth usage percentage comparison for single CoS unbounded manifest files, per-CoS bounded manifest files, and multiple CoS unbounded manifest files, $W = 5$.

the best overall utilization percentage, but that our approach provides a higher utilization percentage than the bounded manifest file case, providing access to excess capacity to lower CoS clients, while still enforcing CoS differentiation.

Figure 5-8 (a) shows a simulation with five groups of clients: A, B, C, D, and E, all with the same CoS. As expected, the distribution of bandwidth between clients, regardless of CoS, is fairly even. In Figure 5-8 (b), just as in Figure 5-7 (b), the each group of clients is assigned a different CoS and per-CoS manifest files are provided to clients of each group. Each CoS manifest file advertises a successively lower maximum bitrate such that $b_{max} = b_{B-1-W+w_c}$. We can again see that as each CoS hits its maximum bitrate, the goodput flattens and under-utilization of the network occurs. Figure 5-8 (c), shows the bandwidth distribution for our intelligent download abort scheme. As with Figure 5-7 (c), Figure 5-8 (c) clearly shows the differentiation in bandwidth distribution between the five classes of service, unlike the single CoS case shown in Figure 5-8 (a), and exhibits no hard cap on the total network throughput, unlike the bounded manifest case shown in Figure 5-7 (b). Table 5.5 shows the aggregate utilization percentages across all network capacities corresponding to Figures 5-8 (a)-(c). We can clearly see that our approach provides a significantly higher utilization percentage than the bounded manifest file case, while continuing to CoS differentiation.

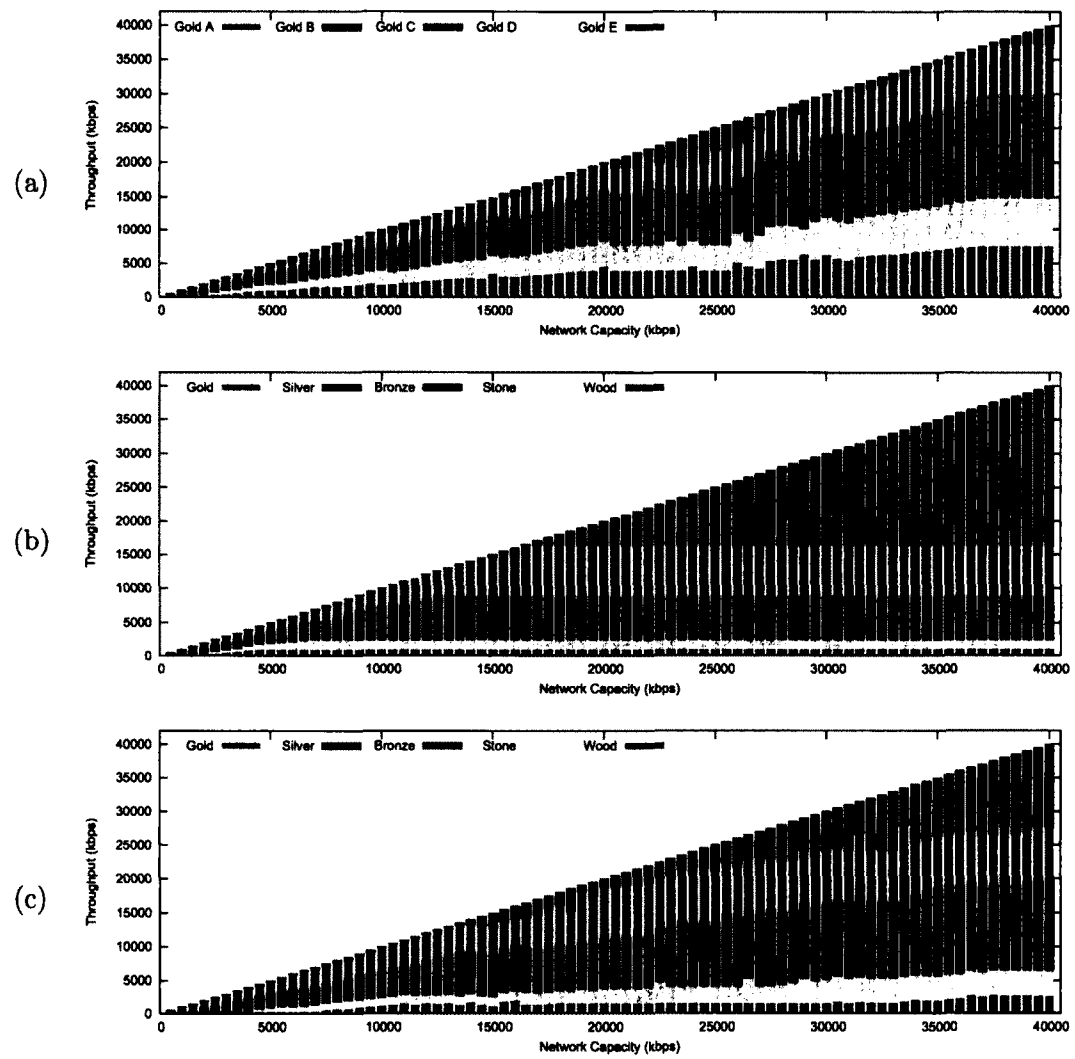


Figure 5-8: Per-CoS goodput distribution for: (a) single CoS unbounded manifest files, (b) per-CoS bounded manifest files, and (c) multiple CoS unbounded manifest files, $W = 5$.

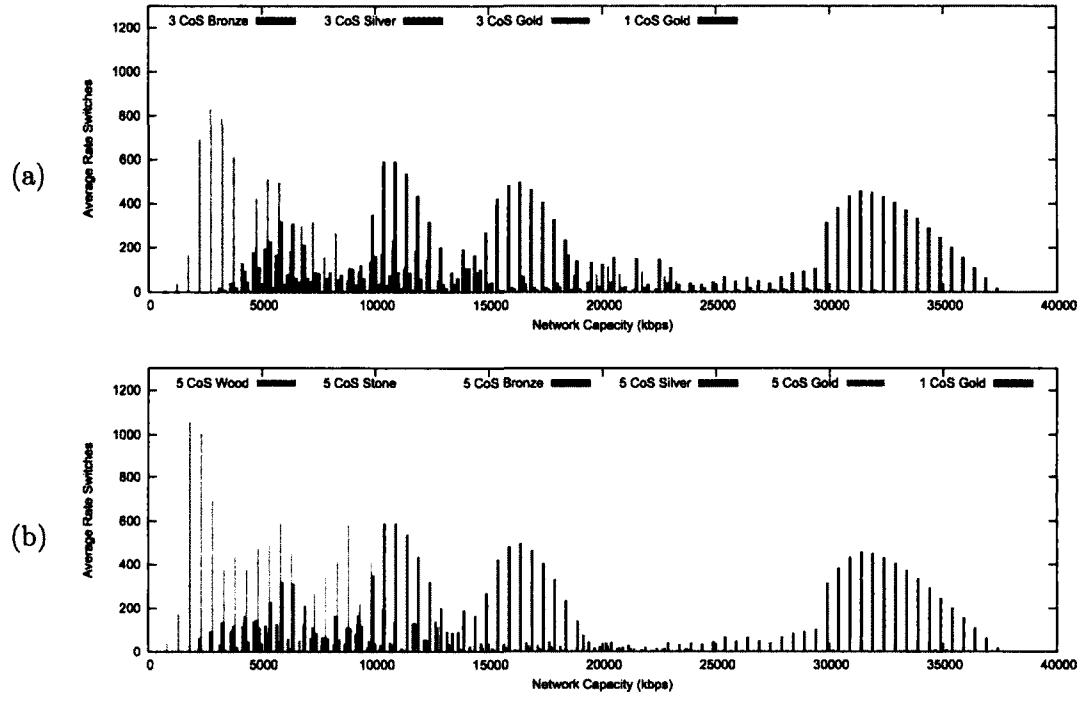


Figure 5-9: Average number of client rate switches per CoS for: (a) single CoS vs. multiple CoS $W = 3$, and (b) single CoS vs. multiple CoS $W = 5$.

5.7.5 Bitrate Consistency Comparison

Beyond the ability to provided differentiated CoS, our rate adaptation algorithm can provide improved bitrate consistency. Reducing bitrate thrashing decreases badput and can improve playback continuity. Figures 5-9 (a) and (b) show the average number of rate switches per client for each CoS in the $W = 3$ and $W = 5$ multiple CoS cases, respectively, and compares them to the single $W = 1$ CoS case. In the majority of cases, bitrate switches occur less than 6% of the time, where the 100000 second runtime results in 10000 segment requests and the average number of rate switches being less than 600 is less than 6%. Though the multiple CoS cases show elevated (~10%) rate switch counts at the low end of the network capacity spectrum, the performance outside of the low bandwidth corner case shows much better consistency than the single CoS case. As expected, the rate switch spikes directly correlate to the visible badput in Figures 5-2 (a)-(c).

Examining the data in Figures 5-9 (a) and (b), we can see that in the multiple CoS cases, bitrate switches are most prevalent when there is not yet clear bitrate differentiation between classes, i.e., at low network capacity, before all clients have stabilized above the lowest bitrate. The rate switches are dominated by the gold CoS, due to it having the most aggressive rate adaptation parameters. At higher network capacities, latching prevents excessive bitrate thrashing, in the multiple CoS cases. However, for the single CoS case, because all clients have equal access to bandwidth, bitrate thrashing is more prevalent at encoded bitrate inflection points, as each client competes to be the first to up-switch to the next bitrate. The lack of excess bandwidth, due to the lack of success and abort backoffs in the single CoS case, prevents latching from having the same rate switch reduction effects as in the multiple CoS cases.

5.8 Network Throughput Dependency

Unlike the network proxy case described in Chapter 4, which has access to network capacity information, individual clients rely on segment download statistics to estimate the network throughput available to them. In the rate adaptation algorithm defined above, the estimated throughput is used in the up-switch decision making process. Throughput is also he primary factor in the download abort decision making process. As such, traditional per-packet traffic shaping schemes, along with server-side schemes like the one detailed in Chapter 2, can affect client rate selection decisions. While network-level traffic shaping rarely isolates individual streaming sessions, especially in the case of HTTP adaptive streaming, where sessions include many disjoint TCP connections, higher level intelligent devices like the network proxies described in Chapter 4 could be used to modify throughput, using pacing schemes similar to the ones used by the streaming servers described in Chapter 2. An HTTP adaptive streaming session-based traffic shaper could be used to influence client rate selection decisions, i.e., reducing throughput to induce down-switches or increasing throughput to induce up-switches. Further investigation into the use of delivery pacing to control client rate adaptation is another interesting area for future research.

5.9 Summary of Rate Adaptation Algorithm Contributions

Though the development of the client rate adaptation architectures described in Chapter 3 required implementation of a rate adaptation algorithm, a simple threshold-based algorithm, somewhat biased to the greedy side, was more than sufficient for those needs. As we investigated CoS enforcement schemes with respect to the network proxy architecture discussed in Chapter 4, our focus began to shift toward the definition of a distributed CoS enforcement scheme. Concerned with the scalability of the centralized network proxy-based approach, we turned to the client to investigate what controls were available and began to investigate the influence of the rate adaptation algorithm.

The first contribution of our client rate adaptation algorithm was the formal modeling of a rate adaptation algorithm that is fully discretized at a segment level, and completely separate from the lower layer network model. Our model enforces the separation of application layer controls from the transport layer information that is often hidden by different device OS platform APIs. In working with the various client architectures discussed in Chapter 3, it became clear that not only were TCP-based statistics unreliable for predicting bandwidth at the segment time scale, but, in most cases, TCP statistics were unavailable to external application developers. By defining an application callback architecture for our rate adaptation algorithm, we are able to abstract the networking and timer infrastructure, typically provided by the OS, out of the rate adaption model and into our generic network simulation model. While much of the research into HTTP adaptive streaming continues to focus on TCP-level optimizations, our segment-level treatment highlights the capabilities available at the segment time scale.

As with the network proxy scenarios discussed in Chapter 4, we continued to focus on developing an infrastructure for managed OTT video delivery, with support for large numbers of unmanaged OTT clients. Using only parameters within the application's scope of control, we began to investigate fair bandwidth distribution using less greedy clients, but with the firm restriction that clients not be aware of and not have to communicate with other clients in the network. The second contribution of our rate adaptation algorithm

research was our ability to create a system of CoS-aware fair use OTT video streaming clients using only minor modifications to our basic segment-level rate adaptation scheme. We also investigated optimization techniques for reclaiming excess bandwidth while still maintaining CoS-based fair use enforcement. The third contribution comes from our multi-client simulation environment which has allowed us to gain a much deeper understanding of the true impact of burst download interleaving, and has opened up an array of new research questions and opportunities with respect to segment-based network optimization.

Chapter 6

Conclusion

For well over a decade, researchers have been investigating methods for enhancing the capabilities of the RTP streaming protocol to improve the content delivery reliability with the ultimate goal of improving the quality of the rendered content. The RTP protocol was defined with the specific intent of delivery real-time content. To minimize latency, content generated in real-time must be delivered just-in-time. Any content that is lost in transmission must be ignored as the stream must go on. With these constraints in mind, RTP was designed to use frame-based packetization and unreliable UDP transport to enable "graceful degradation", i.e., the ability to ignore lost frames. While graceful degradation is necessary for real-time content and offers advantages in extremely challenging networking environments, under normal circumstances RTP-based delivery simply creates the possibility for unnecessary degradation. Researchers have long regarded RTP as a one size fits all video delivery protocol, applying it equally to live and interactive real-time video, as well as VoD and time-shifted broadcast video. Though RTP is a capable video delivery protocol, it is not optimal in all cases.

A large majority of consumers get their video from either a television provider (MSOs and MNOs) or over the Internet. In general, neither of these distribution mechanisms use RTP. MSOs and MNOs typically distribute content using MPEG2-TS over UDP. Though a standard exists for transporting MPEG-TS inside RTP over UDP [98], it is generally considered an unnecessary overhead [99]. For Internet-based video distribution, the formerly closed RTMP protocol [100] continues to be the dominant method for Web-based delivery distribution, while HTTP adaptive streaming protocols like HLS and Microsoft®

Silverlight™ Smooth Streaming are the preferred methods for mobile devices and OTT video delivery services. Where RTP uses graceful degradation to cope with network interruptions, HTTP adaptive streaming relies on bitrate adaptation and playback buffer elasticity to deal with changes in network conditions without randomly discarding data. With the growing popularity of HTTP adaptive streaming technology, MSOs and MNOs have begun to migrate toward OTT video delivery, both as a way to lower operational expenses, as well as a way to offer new services to their customers who have grown to expect content availability on their mobile devices.

MSOs and MNOs are accustomed to having complete control over an entire network of managed devices. Each linear television channel is provisioned for bandwidth and multicast distribution over a QAM, while individual STBs and modems are registered and provisioned for on-demand bandwidth requirements. Moving to an OTT model with unmanaged clients poses a significant issue for operators who wish to deliver their own "on-deck" (i.e., operator managed) services. For "off-deck" services (i.e. services not administered by the network operator), the operator simply sees a generic data connection. Traditional traffic management techniques can be used to throttle individual clients. Because the service is off-deck, there is no concern about any impact to quality which may occur from rate limiting. With on-deck services, however, operators want to ensure high QoE so that customers will continue to pay for the service. In these on-deck cases, the ability to manage OTT delivery in a way that ensures high QoE is highly desirable.

Our research has followed the evolution of HTTP-based content delivery, from our early work with HTTP streaming servers, through the development of our client rate adaptation architecture, to our network proxy-based traffic management schemes, and culminating with our distributed CoS enforcement scheme. Though we have surveyed a large portion of the prior work in the RTP space, we have approached HTTP-based content delivery with a clean slate. Taking the primary advantage of streaming delivery, i.e., pacing, and removing the primary disadvantage of streaming delivery, i.e., silent data loss, we embarked on a journey to understand the barriers to an efficient (i.e., paced), high quality (i.e., lossless),

and manageable HTTP-based video delivery protocol.

6.1 Server Pacing

We originally implemented our Zippy HTTP streaming server to provide a scalable option for server-paced progressive download and succeeded in providing better scalability than the de facto standard Apache HTTP server for video downloads. Our high concurrency architecture for serving long lived connections included session pacing for more even bandwidth distribution, as well as intelligent bursting to help preload client playback buffers and to support client-side paced downloads. Though HTTP adaptive streaming is generally driven by client requests and an assumption that commodity HTTP servers will not perform pacing, it does not mean that server-side (or proxy-based) pacing has become obsolete. Pacing can still play a role in server and network scalability, there is just a new bounds on time scale for pacing delays. Where we went from zero pacing to pacing based on video bitrate, there is still a fair amount of middle ground in which to investigate the effects. Understanding the effects of limited pacing on HTTP adaptive bitrate clients may prove server-side (or proxy-based) pacing to be a viable tool for OTT video delivery client management.

6.2 Client Rate Adaptation

Our data proxy-based client rate adaptation architecture was developed to address the need for a common software architecture, that could be used across device and OS platforms, to implement different rate adaptation schemes and algorithms. The primary concern was the ability to test rate adaptation on client devices which did not natively support any of the existing HTTP adaptive streaming protocols (i.e., HLS or Smooth Streaming). We abstracted the generic components that all rate adaptation clients need, namely: an HTTP download module for retrieving segmented data, a data proxy module for interfacing with the native media player, and a control module for monitoring downloads, monitoring play-

back, performing rate selection, and managing the interface to the application and viewer. Due to the widely disparate native media player interface support, the data proxy module was the least generic component. We were able to categorize four types of native media player interfaces: HTTP, RTSP, local file (byte-based) data source, and frame-based data source. The data proxy also encapsulated content decryption logic and support for a number of cipher algorithms, including: AES-128, RC4, HC-128, and fixed key.

We developed two novel rate adaptation schemes to integrate with legacy media player interfaces. The first uses stitched media files to address media players that only understood legacy container formats, e.g., 3GP and MP4. Individual bitrate encodings are concatenated into a single stitched media file. Content retrieval is performed using HTTP Range GETs and rate adaptation is performed using seek operations to jump to an alternate encoding. The second uses RTP segment files to address media players that only supported the RTSP and RTP protocols. An RTP encapsulated stream is recorded and segmented for each encoding. The segmentation is similar to the HLS protocol, only the segment files contain RTP and RTCP frames rather than MPEG2-TS data. Rate adaptation is performed transparently from the perspective of the player, similar to the other RTP stream splicing methods [101]. These two schemes use very different approaches to both content retrieval and rate switching. The stitched media file approach uses pseudo-segmentation (i.e., virtual segment boundaries enforced in the HTTP Range requests), while the RTP segment approach uses separate physical files. The stitched media file also uses native media player controls to issue seek commands for rate switch, while the RTP segment approach simply switches encodings without notifying the media player at all. We can see from these implementations the flexibility and versatility of the our client rate adaptation architecture. Having a variety of individual platform implementations provided us with both interesting data points on the magnitude of disparity between different devices and OS platforms, as well as an invaluable set of actual devices and platforms from which to test our network proxy architecture and our rate adaptation algorithms.

6.3 Network Proxy Bitrate Override

Using the knowledge we gathered through the development of the Zippy HTTP streaming server and the numerous client rate adaptation architectures we turned our attention to an intelligent network proxy which could recognize and optimize HTTP adaptive streaming sessions. One of the unique considerations for detecting and monitoring HTTP adaptive streaming sessions is that each segment request is typically performed over a different TCP connection. Traditional mechanisms for classifying flows based on network 5-tuples (i.e., source IP address, destination IP address, source port, destination port, and protocol) does not work for HTTP adaptive streaming sessions. HTTP adaptive streaming session must be correlated based on the content which is being requested and the temporal proximity of the requests. Under normal playback conditions, a request for a given segment should be followed by a request for the subsequent segment (as defined by the manifest file) within L seconds, where L is the segment duration. The first step in creating an HTTP adaptive streaming session-aware network proxy is defining an algorithm for classifying sessions based on information gleaned from individual requests. Our scheme uses URI prefix information from the manifest files, coupled with client identifiers to correlate intra-session requests.

To optimize individual HTTP adaptive streaming sessions, we developed the concept of rate selection override, with the goal of optimizing overall network utilization and providing equal QoE to all users. We observed the greedy nature of clients causing congestion which impacted the QoE of all clients. Our goal was to eliminate excessive congestion caused by over-aggressive clients, while still being able to differentiate based on CoS. Our rate selection override scheme was based on the interchangeable nature of segment files in HLS, DASH, and our own RTP segment schemes. Given the knowledge about the available bitrates and segment locations (as defined in the manifest files), the network proxy is able to intercept requests and perform simple URL rewrites or HTTP 302 redirects to override the client bitrate selection. We implemented two modes: full override mode which hides all bitrate information from the client allowing the network proxy to make all rate selection decisions, and bandwidth capping mode which advertises all bitrates to the client allowing the client

to make rate selection decisions based on localized network conditions while the network proxy only overrides the selected bitrate if the network bandwidth cap has been exceeded. Both modes have valid deployment uses cases, but the generalized model for centralized traffic management coincides with the typical management practices of large scale network operators. Understanding the need for manageability in OTT video delivery services, our network proxy-based rate selection override architecture provides a basis for addressing those needs.

6.4 Distributed CoS Enforcement

Having successfully demonstrated the ability to perform CoS enforcement using a centralized network proxy-based approach, we set out to find a distributed method for enforcing CoS differentiation. Though P2P-based media player clients are known to implement discovery methods and communicate with each other to acquire content and exchange network information, most typical media players are independent and do not rely on any inter-player coordination. Our objective was to define a distributed algorithm for providing fair access to network resources, on a CoS basis, that did not require individual clients to be aware of the existence of other clients or the specific network activities of other clients. We assume that each client is made aware of its CoS, and that each client is able to glean information about its own network resource allocation, but no other inputs from the operator, the network, or the other clients would be available to the client. Client awareness of their own network resource availability is typically an inherent part of the rate adaptation algorithm as clients measure the download time of each segment to determine the sustainability of a given bitrate. Our approach was to weight the inferred bandwidth measurements in order to adjust the perceived network availability, relative to the CoS.

We started by defining a segment-level rate adaptation model. Adhering to a segment-level abstraction provides two benefits: it works within the client application's scope of control (i.e., not assuming control over low-level networking functions hidden by the OS), and it represents the proper time scale, relative to the client playback buffer, over which to

amortize bandwidth estimates. HTTP downloads are bursty, and bursts can occur at any time during a segment download. For frame-based schemes like RTP, where the minimum unit of granularity (i.e. a frame) is very small (on the order of one or tens of IP packets), instantaneous throughput measurements are a reasonable estimate of the network conditions that will be experienced by the next frame. However, with HTTP adaptive streaming, where the minimum unit of granularity is a complete segment (on the order of hundreds or thousands of IP packets), making rate adaptation decisions on the basis of instantaneous throughput measurements is less justifiable as instantaneous bandwidth measurements over the course of the entire segment download are likely to vary significantly.

Using the segment level abstraction allowed us to recognize the value of segment download aborts. Though other rate adaptation algorithms work at a segment level, they do not consider the necessity of the segment download abort [43]. Completing a download which is taking much longer than the segment duration is a losing effort, especially for live streams where the client will only fall further and further behind. This is an issue that we also dealt with in the Zippy HTTP streaming server, where falling behind required catch-up bursting. In the client rate adaptation algorithm case, the segment download abort allows the client to more aggressively apply "catch-up" bursting (i.e., download of a lower bitrate segment). In our standard rate adaptation algorithm, segment download abort timeouts are set to be equal to the segment duration. For class of service differentiation, we use more aggressive segment download abort timeouts for lower CoS clients. Whenever a segment download is terminated (either due to the segment download completing or a segment download abort occurring), a backoff (pacing) delay, relative to the CoS, is inserted before the next download is initiated. A natural backoff (pacing) also occurs whenever a client's queue fills. The net effect is a limitation on the amortized bandwidth consumed by clients of a given CoS, without limiting the peak burst throughput. The backoff between downloads create pockets of network availability that allow high throughput bursts of which any client can take advantage, regardless of their CoS. The resultant interleaving of bursted segment downloads represents a more appropriate traffic pattern for multi-client HTTP adaptive streaming.

The strict distribution of bandwidth that results from using segment download aborts and inter-segment request backoffs can cause under-utilization of network resources when the intervals between bitrates are large. We introduce a random up-switch initiation technique which allows clients to be more aggressive in claiming excess bandwidth, but only up to the level of the next higher CoS (to prevent random priority inversion). Overly aggressive up-switch attempts, however, can negatively impact all clients by increasing congestion, so we introduced random up-switch initiation value latching to slow the rate of aggression and help to find a network usage equilibrium point. Though other optimizations for various client and network scenarios undoubtedly exist, with our CoS differentiated segment-based rate adaptation algorithm and flexible network model, we have provided a basis for additional network usage optimization research.

6.5 HTTP Adaptive Streaming

Over the course of this research, we have had the opportunity to work with a variety of emerging and rapidly evolving technologies. We have witnessed and contributed to the launch of a new area of research, namely this new class of HTTP adaptive streaming video delivery protocols which break the mold of traditional RTP-based streaming. While our network proxy-based and distributed rate adaptation algorithm-based CoS differentiation schemes have enhanced our understanding of these new problem areas, many interesting questions remain. Using the foundation which we have laid out, we can continue to investigate the network implications of segment-based delivery, the paradigm shifts required to optimize throughput on segment time scales, the enhancements required to implement OTT client monitoring and management, and the trade-offs involved in optimizing rate adaptation algorithms.

Bibliography

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," Internet Engineering Task Force (IETF), RFC 2616, June 1999.

[2] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: a transport protocol for real-time applications," Internet Engineering Task Force (IETF), RFC 3550, July 2003.

[3] H. Schulzrinne, A. Rao, and R. Lanphier, "Real time streaming protocol (RTSP)," Internet Engineering Task Force (IETF), RFC 2326, April 1998.

[4] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnson, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: session initiation protocol," Internet Engineering Task Force (IETF), RFC 3261, June 2002.

[5] C. Riede, A. Al-Hezmi, and T. Magedanz, "Session and media signaling for IPTV via IMS," in *Proceedings of the 2008 ACM International Conference on Mobile Wireless Middleware, Operating Systems, and Applications (MOBILWARE 2008)*, February 2008, pp. 20:1–20:6.

[6] K. J. Ma, R. Bartoš, and S. Bhatia, "Scalability of HTTP streaming with intelligent bursting," in *Proceedings of the 2009 IEEE International Conference on Multimedia & Expo (ICME 2009)*, June 2009, pp. 798–801.

[7] K. J. Ma, M. Li, A. Huang, and R. Bartoš, "Video rate adaptation in mobile devices via HTTP progressive download of stitched media files," *IEEE Communications Letters*, pp. 320–322, March 2011.

[8] K. J. Ma, M. Mikhailov, and R. Bartoš, "DRM optimization for stitched media file rate adaptation," in *Proceedings of the 2011 IEEE International Conference on Multimedia & Expo (ICME 2011)*, July 2011.

[9] B. Wang, J. Kurose, P. Shenoy, and D. Towsley, "Multimedia streaming via TCP: An analytic performance study," in *Proceedings of the 2004 ACM Multimedia*, October 2004.

[10] A. Goel, C. Krasic, and J. Walpole, "Low-latency adaptive streaming over TCP," *ACM Transactions on Multimedia Computing Communications and Applications*, vol. 4, no. 3, August 2008.

[11] S. Tullimas, T. Nguyen, and R. Edgecomb, "Multimedia streaming using multiple TCP connections," *ACM Transactions on Multimedia Computing Communications and Applications*, vol. 2, no. 2, May 2008.

[12] B. Wang, W. Wei, Z. Guo, and D. Towsley, "Multipath live streaming via TCP: Scheme, performance and benefits," *ACM Transactions on Multimedia Computing Communications and Applications*, vol. 5, no. 3, August 2009.

[13] S. Sen, J. Rexford, and D. Towsley, "Proxy prefix caching for multimedia streams," in *Proceedings of the 1999 IEEE International Conference on Computer Communications (InfoCom 1999)*, March 1999, pp. 1310–1319.

[14] K. Kalapriya and S. K. Nandy, "Throughput driven, highly available streaming stored playback video service over a peer-to-peer network," in *Proceedings of the 2005 IEEE International Conference on Advanced Information Networking and Applications (AINA 2005)*, March 2005, pp. 229–234.

[15] E. Kusmierek, Y. Dong, and D. H. C. Du, "Loopback: Exploiting collaborative caches for large scale streaming," *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 233–242, April 2006.

[16] S. Deshpande and J. Noh, "P2TSS: Time-shifted and live streaming of video in peer-to-peer systems," in *Proceedings of the 2008 IEEE International Conference on Multimedia & Expo (ICME 2008)*, June 2008, pp. 649–652.

[17] L. Shen, W. Tu, and E. Steinbach, "A flexible starting point based partial caching algorithm for video on demand," in *Proceedings of the 2007 IEEE International Conference on Multimedia & Expo (ICME 2007)*, July 2007, pp. 76–79.

[18] X. Li, W. Tu, and E. Steinbach, "Dynamic segment based proxy caching for video on demand," in *Proceedings of the 2008 IEEE International Conference on Multimedia & Expo (ICME 2008)*, June 2008, pp. 1181–1184.

[19] W. Tu, E. Steinbach, M. Muhammad, and X. Li, "Proxy caching for video-on-demand using flexible start point selection," *IEEE Transactions on Multimedia*, vol. 11, no. 4, pp. 716–729, June 2009.

[20] W. Ma and D. H. C. Du, "Reducing bandwidth requirement for delivering video over wide area networks with proxy server," *IEEE Transactions on Multimedia*, vol. 4, no. 4, pp. 539–550, December 2002.

[21] S. Chen, B. Shen, S. Wee, and X. Zhang, "Designs of high quality streaming proxy systems," in *Proceedings of the 2004 IEEE International Conference on Computer Communications (InfoCom 2004)*, March 2004, pp. 1512–1521.

[22] ——, "Segment-based streaming media proxy: Modeling and optimization," *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 243–256, April 2006.

[23] ——, "Sproxy: A caching infrastructure to support Internet streaming," *IEEE Transactions on Multimedia*, vol. 9, no. 5, pp. 1064–1074, August 2007.

[24] S. Wee, W. Tan, J. Apostolopoulos, and M. Etoh, "Optimized video streaming for networks with varying delay," in *Proceedings of the 2002 IEEE International Conference on Multimedia & Expo (ICME 2002)*, August 2002, pp. 89–92.

[25] Y. J. Liang and B. Girod, "Network-adaptive low-latency video communication over best-effort networks," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 1, pp. 72–81, January 2006.

[26] Y. J. Liang, J. G. Apostolopoulos, and B. Girod, "Analysis of packet loss for compressed video: Does burst-length matter?" in *Proceedings of the 2003 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2003)*, April 2003, pp. 684–687.

[27] Y. J. Liang, J. Apostolopoulos, and B. Girod, "Analysis of packet loss for compressed video: Effect of burst losses and correlation between error frames," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 18, no. 7, pp. 861–874, July 2008.

[28] P. Baccichet, J. Noh, E. Setton, and B. Girod, "Content-aware P2P video streaming with low latency," in *Proceedings of the 2007 IEEE International Conference on Multimedia & Expo (ICME 2007)*, July 2007, pp. 400–403.

[29] E. Setton, J. Noh, and B. Girod, "Low latency video streaming over peer-to-peer networks," in *Proceedings of the 2006 IEEE International Conference on Multimedia & Expo (ICME 2006)*, July 2006, pp. 569–572.

[30] J. Li, C. K. Yeo, and B. S. Lee, "Peer-to-peer streaming scheduling to improve real-time latency," in *Proceedings of the 2007 IEEE International Conference on Multimedia & Expo (ICME 2007)*, July 2007, pp. 36–39.

[31] R. Alimi, R. Penno, and Y. Yang, "ALTO protocol," Internet Engineering Task Force (IETF), Internet-Draft version 10 (draft-ietf-alto-protocol-10), May 2012, work in progress.

[32] J. Chakareski and P. Frossard, "Rate-distortion optimized distributed packet scheduling of multiple video streams over shared communication resources," *IEEE Transactions on Multimedia*, vol. 8, no. 2, pp. 207–218, April 2006.

[33] Y. Li, Z. Li, M. Chiang, and A. R. Calderbank, "Content-aware distortion-fair video streaming in congested networks," *IEEE Transactions on Multimedia*, vol. 11, no. 6, pp. 1182–1193, October 2009.

[34] W. Tan, W. Cui, and J. G. Apostolopoulos, "Playback-buffer equalization for streaming media using stateless transport prioritization," in *Proceedings of the 2003 IEEE Packet Video Workshop*, April 2003.

[35] Y. Li, A. Markopoulou, J. Apostolopoulos, and N. Bambos, "Content-aware playout and packet scheduling for video streaming over wireless links," *IEEE Transactions on Multimedia*, vol. 10, no. 5, pp. 885–895, August 2008.

[36] R. Pantos and W. May, "HTTP Live Streaming," Internet Engineering Task Force (IETF), Internet-Draft Version 7 (draft-pantos-http-live-streaming-07), September 2011, work in progress.

[37] I. Sodagar, "The MPEG-DASH standard for multimedia streaming over the Internet," *IEEE Multimedia Magazine*, vol. 18, no. 4, pp. 62–67, April 2011.

[38] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP," in *Proc. of the 2011 ACM Conference on Multimedia Systems (MMSys 2011)*, February 2011, pp. 157–168.

[39] K. J. Ma and R. Bartoš, "Rate adaptation with CoS enforcement for cloud-based HTTP streaming," *IEEE Transactions on Multimedia*, (submitted for publication).

[40] R. Kuschnig, I. Kofler, and H. Hellwagner, "An evaluation of TCP-based rate-control algorithms for adaptive Internet streaming of H.264/SVC," in *Proc. of the 2010 ACM SIGMM Conference on Multimedia Systems (MMSys 2010)*, February 2010, pp. 157–168.

[41] C. Liu, I. Bouazizi, and M. Gabbouj, "Segment duration for rate adaptation of adaptive HTTP streaming," in *Proc. of the 2011 IEEE International Conference on Multimedia and Expo (ICME 2011)*, July 2011.

[42] T. Kupka, P. Halvorsen, and C. Griwodz, "An evaluation of live adaptive HTTP segment streaming request strategies," in *Proc. of the 2011 IEEE Conference on Local Computer Networks (LCN 2011)*, October 2011, pp. 604–612.

[43] C. Liu, I. Bouazizi, and M. Gabbouj, "Rate adaptation for adaptive HTTP streaming," in *Proc. of the 2011 ACM SIGMM Conference on Multimedia Systems (MMSys 2011)*, February 2011, pp. 169–174.

[44] H. Yu, E. Chang, W. T. Ooi, M. C. Chan, and W. Cheng, "Integrated optimization of video server resource and streaming quality over best-effort network," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 19, no. 3, pp. 374–385, March 2009.

[45] C. Kao and C. Lee, "Aggregate profit-based caching replacement algorithms for streaming media transcoding proxy systems," *IEEE Transactions on Multimedia*, vol. 9, no. 2, pp. 221–230, February 2007.

[46] R. Kumar, "A protocol with transcoding to support QoS over Internet for multimedia traffic," in *Proc. of the 2003 IEEE International Conference on Multimedia & Expo (ICME 2003)*, July 2003, pp. 465–468.

[47] L. Zhijun and N. D. Georganas, "Rate adaptation transcoding for video streaming over wireless channels," in *Proc. of the 2003 IEEE International Conference on Multimedia & Expo (ICME 2003)*, July 2003, pp. 433–436.

[48] T. Kim and M. H. Ammar, "A comparison of heterogeneous video multicast schemes: Layered encoding or stream replication," *IEEE Transactions on Multimedia*, vol. 7, no. 6, pp. 1123–1130, December 2005.

[49] T. Schierl, T. Stockhammer, and T. Wiegand, "Mobile video transmission using scalable video coding," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 17, no. 9, pp. 1204–1217, September 2007.

[50] V. K. Goyal, "Multiple description coding: Compression meets the network," *IEEE Signal Processing Magazine*, vol. 18, no. 5, pp. 74–93, September 2001.

[51] Y. Wang, A. R. Reibman, and S. Lin, "Multiple description coding for video delivery," *Proceedings of the IEEE*, vol. 93, no. 1, pp. 57–70, January 2005.

[52] P. Fröjdh, U. Horn, M. Kampmann, A. Nohlgren, and M. Westerlund, "Adaptive streaming within the 3GPP packet-switched streaming service," *IEEE Network Magazine*, vol. 20, no. 2, pp. 34–40, March 2006.

[53] T. Schierl, T. Wiegand, and M. Kampmann, "3GPP compliant adaptive wireless video streaming using H.264/AVC," in *Proc. of the 2005 IEEE International Conference on Image Processing (ICME 2005)*, September 2005, pp. 696–699.

[54] J. G. Apostolopoulos and M. D. Trott, "Path diversity for enhanced media streaming," *IEEE Communications Magazine*, pp. 80–87, August 2004.

[55] J. Chakareski and B. Girod, "Server diversity in rate-distorted optimized media streaming," in *Proceedings of the 2003 IEEE International Conference on Image Processing (ICIP 2003)*, September 2003, pp. 645–648.

[56] J. Chakareski and P. Frossard, "Distributed streaming via packet partitioning," in *Proceedings of the 2006 IEEE International Conference on Multimedia & Expo (ICME 2006)*, July 2006, pp. 1529–1532.

[57] ——, "Distributed collaboration for enhanced sender-driven video streaming," *IEEE Transactions on Multimedia*, vol. 10, no. 5, pp. 858–870, August 2008.

[58] T. Nguyen and A. Zakhor, "Distributed video streaming over Internet," in *Proceedings of the 2002 SPIE Multimedia Computing and Networking (MMCN 2002)*, January 2002, pp. 186–195.

[59] ——, "Multiple sender distributed video streaming," *IEEE Transactions on Multimedia*, vol. 6, no. 2, pp. 315–326, April 2004.

[60] M. Guo, Q. Zhang, and W. Zhu, "Selecting path-diversified servers in content distribution networks," in *Proceedings of the 2003 IEEE Global Telecommunications Conference (GlobeCom 2003)*, December 2003, pp. 3181–3185.

[61] A. D. Mauro, D. Schonfeld, and C. Casetti, "A peer-to-peer overlay network for real time video communication using multiple paths," in *Proceedings of the 2006 IEEE International Conference on Multimedia & Expo (ICME 2006)*, July 2006, pp. 921–924.

[62] J. G. Apostolopoulos, W. Tan, and S. J. Wee, "Performance of a multiple description streaming media content delivery network," in *Proceedings of the 2002 IEEE International Conference on Image Processing (ICIP 2002)*, September 2002, pp. II–189–III–192.

[63] J. Apostolopoulos, W. Tan, S. Wee, and G. W. Wornell, "Modeling path diversity for multiple description video communication," in *Proceedings of the 2002 IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP 2002)*, May 2002.

[64] J. G. Apostolopoulos, W. Tan, and S. J. Wee, "On multiple description streaming with content delivery networks," in *Proceedings of the 2002 IEEE International Conference on Computer Communications (InfoCom 2002)*, June 2002, pp. 1736–1745.

[65] F. A. Mogus, "Performance comparison of multiple description coding and scalable video coding," in *Proc. of the 2011 IEEE International Conference on Communication Software and Networks (ICCSN 2011)*, May 2011, pp. 452–456.

[66] D. Renzi, P. Amon, and S. Battista, "Video content adaptation based on SVC and associated RTP packet loss detection and signaling," in *Proc. of the 2008 IEEE International Workshop on Image Analysis for Multimedia Interactive Services (WIAMIS 2008)*, May 2008, pp. 97–100.

[67] T. Schierl, K. Gruneberg, and T. Wiegand, "Scalable video coding over RTP and MPEG-2 transport stream in broadcast and IPTV channels," *IEEE Wireless Communications Magazine*, vol. 16, no. 5, pp. 64–71, October 2009.

[68] Z. Avramova, D. D. Vleeshauwer, K. Spaey, S. Wittevrongel, H. Bruneel, and C. Blondia, "Comparison of simulcast and scalable video coding in terms of the required capacity in an IPTV network," in *Proc. of the 2007 IEEE International Packet Video Workshop*, November 2007, pp. 113–122.

[69] T. Schierl, K. Ganger, C. Hellge, T. Wiegand, and T. Stockhammer, "SVC-based multisource streaming for robust video transmission in mobile ad hoc networks," *IEEE Wireless Communications Magazine*, vol. 13, no. 5, pp. 96–203, October 2006.

[70] C. Hellge, T. Schierl, and T. Wiegand, "Mobile TV using scalable video coding and layer-aware forward error correction," in *Proceedings of the 2008 IEEE International Conference on Multimedia & Expo (ICME 2008)*, June 2008, pp. 1177–1180.

[71] ——, "Multidimensional layered forward error correction using rateless codes," in *Proceedings of the 2008 IEEE International Conference on Communications (ICC 2008)*, May 2008, pp. 480–484.

[72] ——, "Receiver driven layered multicast with layer-aware forward error correction," in *Proceedings of the 2008 IEEE International Conference on Image Processing (ICIP 2008)*, October 2008, pp. 2304–2307.

[73] X. Zhu, R. Pan, N. Dukkipati, V. Subramanian, and F. Bonomi, "Layered internet video engineering (LIVE): network-assisted bandwidth sharing and transient loss protection for scalable video streaming," in *Proceedings of the 2010 IEEE International Conference on Computer Communications (InfoCom 2010)*, March 2010.

[74] S. Chattopadhyay, L. Ramaswamy, and S. M. Bhandarkar, "A framework for encoding and caching of video for quality adaptive progressive download," in *Proceedings of the 2007 ACM International Conference on Multimedia*, September 2007, pp. 775–778.

[75] Y. Li and K. Ong, "Optimized cache management for scalable video streaming," in *Proceedings of the 2007 ACM International Conference on Multimedia*, September 2007, pp. 799–802.

[76] J. Kangasharju, F. Hartanto, M. Reisslein, and K. W. Ross, "Distributing layered encoded video through caches," *IEEE Transactions on Computers*, vol. 51, no. 6, pp. 622–636, June 2002.

[77] S. Gouache, G. Bichot, A. Bsila, and C. Howson, "Distributed & adaptive HTTP streaming," in *Proc. of the 2011 IEEE International Conference on Multimedia and Expo (ICME 2011)*, July 2011.

[78] M. Baugher, D. McGrew, M. Naslund, and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)," Internet Engineering Task Force (IETF), RFC 3711, March 2004.

[79] T. Stutz and A. Uhl, "A survey of H.264 AVC/SVC encryption," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 3, pp. 325–339, March 2012.

[80] V. Swaminathan and S. Mitra, "A partial encryption scheme for AVC video," in *Proceedings of the 2012 IEEE International Conference on Emerging Signal Processing Applications (ESPA 2012)*, January 2012.

[81] S. Lian, Z. Liu, Z. Ren, and H. Wang, "Secure advanced video coding based on selective encryption algorithms," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 2, pp. 621–629, May 2006.

[82] D. Wang, Y. Zhou, D. Zhao, and J. Mao, "A partial video encryption scheme for mobile handheld devices with low power consideration," in *Proceedings of the 2009 IEEE International Conference on Multimedia Information Networking and Security (MINES 2009)*, November 2009, pp. 99–104.

[83] C. Li, C. Yuan, and Y. Zhong, "Layered encryption for scalable video coding," in *Proceedings of the 2009 IEEE International Congress on Image and Signal Processing (CISP 2009)*, October 2009.

[84] M. Asghar and M. Ghanbari, "Cryptographic keys management for H.264 scalable coded video security," in *Proceedings of the 2011 IEEE International ISC Conference on Information Security and Cryptology (ISCISC 2011)*, September 2011, pp. 83–86.

[85] K. J. Ma, R. Nair, and R. Bartoš, "DRM workflow analysis for over-the-top HTTP segmented delivery," in *Proceedings of the 2011 IEEE International Conference on Multimedia & Expo (ICME 2011)*, July 2011.

[86] B. Girod, M. Kalman, Y. J. Liang, and R. Zhang, "Advances in channel-adaptive video streaming," in *Proceedings of the 2002 IEEE International Conference on Image Processing (ICIP 2002)*, September 2002, pp. I–9–I–12.

[87] M. Kalman, E. Steinbach, and B. Girod, "Adaptive media playout for low-delay video streaming over error-prone channels," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 14, no. 6, pp. 841–851, June 2004.

[88] Y. Li, A. Markopoulou, N. Bambos, and J. Apostolopoulos, "Joint power/playout control schemes for media streaming over wireless links," in *Proceedings of the 2004 IEEE Packet Video Workshop*, December 2004.

[89] ——, "Joint packet scheduling and content-aware playout control for video sreaming over wireless links," in *Proceedings of the 2005 International Workshop on Multimedia Signal Processing*, October 2005.

[90] S. C. Hui and J. Y. B. Lee, "Playback-adaptive multi-source video streaming," in *Proceedings of the 2006 IEEE Global Telecommunications Conference (GlobeCom 2006)*, November 2006, pp. 819–922.

[91] A. Argyriou, "Improving the performance of TCP wireless video streaming with a novel playback adaptation algorithm," in *Proceedings of the 2006 IEEE International Conference on Multimedia & Expo (ICME 2006)*, July 2006, pp. 1169–1172.

[92] K. J. Ma, R. Bartoš, S. Bhatia, and R. Nair, "Mobile video delivery with HTTP," *IEEE Communications Magazine*, vol. 49, no. 4, pp. 166–175, April 2011.

[93] A. C. Begen, T. Akgul, and M. Baugher, "Watching video over the Web: Part 1: Streaming protocols," *IEEE Internet Computing Magazine*, vol. 15, no. 2, pp. 54–63, March 2011.

[94] Microsoft, "[MS-WMSP]: Windows Media HTTP streaming protocol specification," June 2010, http://msdn.microsoft.com/en-us/library/cc251059(PROT.10).aspx.

[95] M. Handley, V. Jacobson, and C. Perkins, "SDP: Session Description Protocol," Internet Engineering Task Force (IETF), RFC 4566, July 2006.

[96] Microsoft, "IIS smooth streaming technical overview," March 2009, http://www.microsoft.com/downloads/details.aspx?displaylang=en\&FamilyID=03d22583-3ed6-44da-8464-b1b4b5ca7520.

[97] "Best practices for creating and deploying HTTP Live Streaming media for the iPhone and iPad," Apple, Technical Note TN2224, July 2011.

[98] D. Hoffman, G. Fernando, V. Goyal, and M. Civanlar, "RTP payload format for MPEG1/MPEG2 video," Internet Engineering Task Force (IETF), RFC 2250, January 1998.

[99] "Edge qam video stream interface specification," CableLabs, Tech. Rep. CM-SP-EQAM-VSI-I01-081107, November 2008, http://www.cablelabs.com/cablemodem/specifications/mha.html.

[100] "Real-time messaging protocol (RTMP) specification," Adobe, Tech. Rep., April 2009, http://www.adobe.com/devnet/rtmp.html.

[101] J. Xia, "Content splicing for RTP sessions," Internet Engineering Task Force (IETF), Internet-Draft version 7 (draft-ietf-avtext-splicing-for-rtp-07), August 2012, work in progress.

Appendix A

Abbreviations

- AAC-LC – Advanced Audio Coding Low-Complexity Audio Codec
- AES – Advanced Encryption Standard
- CBC – Cipher Block Chaining
- CDN – Content Delivery Network
- CoS – Class of Service
- DASH – Dynamic Adaptive Streaming over HTTP
- DRM – Digital Rights Management
- ESD – Equal Share Distribution
- FEC – Forward Error Correction
- GOP – Group of Pictures
- H.264 – MPEG-4 Advanced Video Coding (AVC)
- HLS – HTTP Live Streaming
- HTTP – Hyper-Text Tranfer Protocol
- ISP – Internet Service Provider
- IV – Initialization Vector
- JIT – Just In Time
- MDC – Multiple Description Coding
- MNO – Mobile Network Operator
- MSO – Multiple System Operator
- OS – Operating System
- OTT – Over The Top
- P2P – Peer to Peer
- POP – Point of Presence
- QoE – Quality of Experience
- RAN – Radio Access Network
- RC4 – Rivest Cipher 4
- RTCP – Real-time Transport Control Protocol
- RTP – Real-time Transport Protocol
- RTT – Round Trip Time
- SDP – Session Description Protocol
- SIP – Session Initiation Protocol
- SLA – Service Level Agreement
- SRTP – Secure Real-time Transport Protocol
- STB – Set Top Box
- UGC – User Generated Content
- VoD – Video on Demand
- WAN – Wide Area Network