Approaches to Adaptive Bitrate Video Streaming

Conor Cahir

Submitted in fulfilment of the requirements for the degree of
Master of Science by Research

School of Computing Science

College of Science and Engineering
University of Glasgow

March 2014

Abstract

In this work, I use ns-3 simulations to compare and evaluate different approaches to web based adaptive bitrate (ABR) video streaming. In particular, I look at the difference between client pull and server push based approaches, the effects of media formatting parameters such as chunk duration and number of encoding rates, and the implementation of bandwidth estimation and request scheduling strategies. I find that client pull applications with a 2 second chunk duration are very inefficient with bandwidth compared to applications using a server push based approach. The reasons for this stem from the effect of frequent idle periods at chunk boundaries, which are absent with server push, on the behaviour of TCP. Increasing the chunk duration to 10 seconds makes a significant difference to client pull applications and allows them to perform at a level much more comparable with server push applications. I also find that ABR applications in general are vulnerable to suffering from encoding rate instability, a result that echoes findings from a number of recent studies. This problem seems to stem from the difficulty of selecting a suitable encoding rate based on transfer rates observed at the application layer. Effective remedies for encoding rate instability include ensuring that the system is not over provided for in terms of the number of available encoding rates, and using an averaging function, such as the harmonic mean, over a series of recent transfer rates in order to filter out short term fluctuations in the estimate of available bandwidth. I also show that a simple request scheduling strategy can be used to avoid over buffering and the associated problems, but that periodic request scheduling can introduce further problems related to fairness when multiple ABR flows compete. Finally, I show that a hybrid of client pull and server push, which I call pull selective, can offer a useful compromise between the two, by matching the performance characteristics of server push whilst maintaining the low server overheads and scalability attributes of client pull.

Acknowledgements

First and foremost I would like to thank my supervisor, Dr Colin Perkins, for his patience and expert guidance throughout this project. Secondly, I would like to thank Morag and Michael Cahir, for being good parents and for supporting me through university. I would then like to thank all of the teaching, administrative, and technical staff from the University of Glasgow's School of Computing Science and College of Science and Engineering, for all of their excellent work. Finally, this project has been made possible in part by a gift from The Cisco University Research Program Fund, a corporate advised fund of Silicon Valley Community Foundation. Thank you to Cisco, and in particular Ali C. Begen, for giving me this opportunity.

This work is dedicated to my girlfriend Neetu, who has been waiting patiently.

Table of Contents

List of Figures

Chapter 1

Introduction

This dissertation describes the results of a series of experiments looking at adaptive bitrate (ABR) video streaming applications using the ns-3 network simulator. The aim of these experiments was to compare and contrast different approaches to implementing ABR video streaming systems, and discover what the advantages and disadvantages are of each. In this chapter I outline the motivations behind this project, present a formal thesis statement, and give a brief outline of the remaining chapters.

On demand video streaming services have become immensely popular on the web and video content now accounts for the dominant share of traffic on the Internet [2]. Users flock to sites such as YouTube, BBC iPlayer, and Netflix, where they can access a back catalogue of content at their leisure, without having to leave home or wait for downloads, and often for free or at little cost. Rather than waiting for an entire file to download then opening it with a media player, streaming allows the user to begin watching almost immediately by playing the content back as it is downloaded. When the user hits play the client will start downloading frames until it has buffered up a small duration of content locally, enough to last only until new frames can be fetched, then playback will begin. Assuming the rate at which content can be downloaded is greater than or equal to that at which it is consumed during playback, the playback buffer will not shrink and the user will be able to watch the entire video this way without interruption. However, a problem arises when the previous condition is not met and the player is unable to retrieve content fast enough to keep up with playback. This can be a common issue due to the high-volume nature of video content and especially in areas where Internet connections are slow or multiple users share a single connection.

To avoid frequent playback interruptions and give users the opportunity to stream videos even when their network is congested, many service providers now offer players which use ABR streaming. ABR systems trade video quality in return for playback fluency. When frames are not being fetched fast enough to maintain playback, lower quality video is used and therefore less data sent for the same duration of content. To achieve this effect, content is split into chunks of a few seconds duration and each chunk encoded at multiple bitrates. After each chunk is transferred either the client or server, depending on the implementation, can select a suitable encoding rate the for the next chunk by using previous transfer rates to gauge the state of the network.

But ABR streaming is still a relatively new technology, and one which comes with its own set of problems. Along with the proliferation of ABR video players on the web there are now also a number of studies highlighting areas where they fall short. Common issues include unfair sharing of bandwidth and highly fluctuating video quality. From the many options available, it is not yet clear what are the best approaches to implementing an ABR system that mitigate these problems. The motivation for this project is therefore to use simulations to model and investigate the properties of different approaches in an attempt to shed some light on the area, and in a way that might not be feasible using a lab setup.

1.1 Thesis Statement

Despite becoming a popular way to deliver video over the web, there are no clear guidelines available for developers who are faced with a myriad of design choices when implementing an ABR video streaming service. On top of this, recent studies have shown that ABR systems are prone to suffering from issues concerning stability and fairness when multiple streams compete over a bottleneck [3, 1, 4, 5, 6]. I assert that a systematic survey of ABR systems, exploring the advantages and disadvantages of different implementation approaches, is needed as a step towards tackling this situation. I will to carry out this survey using network simulation software to model and evaluate various ABR applications. In particular, I will investigate the tradeoffs between client pull and server push systems, bandwidth estimation techniques, download request scheduling strategies, and media formatting parameters. The results of this survey will provide guidance to help developers make educated choices in these areas when implementing an ABR video streaming system.

1.2 Document Outline

Chapter 2 gives a summary of the relevant background knowledge, along with a survey of some recent literature in the field. In Chapter 3 I describe the tools and methodology I use, and outline specifically what simulations I will be looking at. Chapters 4, 5, 6, and 7 each look at a different set of simulations, including an overview of the simulations, their results, and a discussion followed by conclusions. Finally, in Chapter 8, I conclude with a recap of the project and a summary of my results and contributions.

Chapter 2

Background

Streaming video content has become one of the most common applications of the Internet, and web based systems are a popular way to deliver this service. Many of these systems use ABR streaming, which varies the video encoding rate according to changing network conditions, in order to overcome the limitations of streaming over TCP and allow streams to continue, without playback interruptions, in the presence of congestion. ABR streaming is now a popular research topic and many studies have found problems with existing solutions, but it is not yet clear how best to implement an ABR system that avoids these problems. This dissertation describes a survey of ABR streaming approaches that I have carried out as an effort to help answer that question.

In this chapter I will describe how ABR streaming systems work and discuss previous studies that have looked at them. I will give an overview of Internet video services in general, expand on the brief explanation of ABR streaming given in the introduction, discuss some of the different ways these systems can be implemented, and outline areas where previous studies have found issues. In the end I will show that it is not yet clear what is the best way to implement an ABR system.

2.1 Web Based Video Streaming

Internet video streaming services come in various flavours. Broadly speaking, two types of service are common today, each with advantages and disadvantages in terms of both end user experience, deployment, and infrastructure costs. These are managed services running over dedicated networks, and ‘over-the-top’ web based streaming services.

Many Internet Service Providers (ISPs) now offer commercial video streaming services designed to compete with traditional cable and satellite television. These use dedicated network infrastructure and protocols and are managed by the ISP from end to end to ensure that performance and Quality of Service (QoS) demands are met. Examples include BT Vision [7] and Virgin Media's TiVo service [8]. Video content is transferred using UDP, with multicast for live broadcasting and unicast with caching for video on demand. Video traffic can also be prioritised over other network traffic, for example using DiffServ [9]. This is an all-out approach to Internet video that allows the ISP to offer a very similar experience to what users have become familiar with through traditional broadcast television. The predictability of using UDP over a managed and optimised network allows for fast channel changes and high quality video delivery. However, this service can only be offered by ISPs since they have the access to underlying network infrastructure needed to implement many of the features described above. In particular, this approach requires deployment of IP multicast, which is a non-trivial change for many networks.

The second approach to Internet video, which is popular amongst many content providers, is 'over-the-top' web based streaming. These services deliver content over existing, unmanaged, best effort IP networks, using standard web protocols. This approach has much lower implementation costs than a managed service, requiring only that the service provider deploy standard web distribution infrastructure, which users can then access via their web browser to stream content. Examples include Netflix, BBC iPlayer, and YouTube. Many over the top services use the Hypertext Transfer Protocol (HTTP) to deliver content. This allows existing infrastructure, such as web caches and content delivery networks (CDNs), to be reused, and also provides easy traversal of the Network Address Translation (NAT) devices that are common in home users' networks. The downside of this approach is that the networks can be unpredictable and TCP, which is the underlying transport protocol for HTTP, was never designed with high performance real time applications in mind. TCP comes with built in congestion control mechanisms [10]. On the one hand this is an advantage since it removes that burden from the service provider, but it also affects performance. TCP flows begin slowly and build up speed gradually as acknowledgements are returned from the client. When packet loss is detected, the lost packets must be retransmitted, and the loss is taken as a sign of congestion so the sending rate is lowered again. This leads to unreliable timing and will often have the effect of causing playback stalls, leaving the user with the option of either watching a 'jumpy' video, or pausing playback to wait while more content is buffered up. Both are detrimental to the user's viewing experience. Relatively long startup buffering times also mean this approach is ill suited for both live content and services wishing to offer fast channel switches.

But the flexibility and low entry costs of HTTP based video streaming have driven a surge in popularity amongst content providers, leading researchers to look for ways to overcome TCP's restrictions and improve the performance of these systems. One effort that stands out is the trend towards using bitrate adaptation, where the encoding rate of content can vary to accommodate the current conditions on the network. Such systems are beginning to coalesce under the banner of Dynamic Adaptive Streaming over HTTP (DASH) [11], a new ABR standard that provides a framework for implementing adaptive bitrate players capable of talking to any server that also implements the standard.

2.2 Adaptive Bitrate Streaming

Outlined briefly in the introduction, ABR video streaming is a technique that allows streamed video quality to vary over the lifetime of a stream to meet changing conditions on the network. This allows users to maintain their viewing, at a lower quality, even when networks are congested. ABR streaming, therefore, trades video quality in return for continuous playback in the face of congestion. Many different types of system fall under this description, since all that is required to meet the definition of ABR video streaming is that the video encoding rate can vary over the lifetime of a stream. In this work, however, I only consider systems using HTTP to transfer content that is pre-encoded at several discrete rates, since that is the approach that fits best with the needs of on demand web based video streaming and is gaining traction with commercial services. Other use cases for ABR include live streaming, where content can be encoded on the fly at any chosen rate, for example, or in systems running over managed networks where more information might be available to help guide the choice of encoding rate.

For ABR systems using HTTP with pre-encoded content, the basic approach is to split content into a series of small consecutive files (chunks), each encoded beforehand at multiple rates, then select an appropriate encoding of each chunk when the time comes. This can be achieved either by having the client monitor playback and transfers, then request each chunk from the server at the desired rate, or with a server monitoring transfers and pushing each chunk out to the client after the initial request is received. I refer to these paradigms as client pull and server push throughout. Monitoring transfer rates allows the client or server to gague network conditions and estimate the available bandwidth in order to choose an appropriate rate for each chunk. When the network is congested, chunks encoded at a lower rate will be used. This means that the video quality is reduced, but less data needs to be sent for the same duration of content and hopefully the client can receive more frames before running out of buffered content to play, thus avoiding a playback interruption.

Whether to adopt the client pull or server push approach is one of the first decisions that has to be made when implementing an ABR system. Both approaches have inherent advantages and disadvantages, however there is a trend towards client pull. This can be seen in many commercial systems and is also evidenced in the development of the DASH standard, which implies a client pull based system [11]. Because of this, many studies have focused on client pull, and the properties of server push systems are less well known [1, 3, 12, 4]. As of yet there have been no useful direct comparisons of the two, despite the interesting tradeoffs that exist. Implementing a client pull based system requires only a basic web server hosting the content chunks, along with a manifest file. A client can then provide a streaming service by simply downloading consecutive chunks for playback at suitable rates. This is a very flexible approach, enabling seamless server migration in case of failure for example, and has relatively low implementation costs. A server push implementation on the other hand requires a server that is modified to accommodate push streams and ABR logic. In this case, the client makes a single request for the stream and the server then pushes consecutive chunks out to the client while monitoring transfers and handling rate adaptation. This approach is naturally less scalable since the server must now maintain extra state and perform calculations for each connected client. However, as I will later demonstrate, server push systems can also have a significant performance advantage over client pull systems. Client pull flows tend to be characterised by 'on-off' behaviour. New TCP connections are created, or old ones left idle, while requests are sent to the server between chunk downloads. Both result in TCP re-entering its slow start phase and thus reducing any throughput that had been built up. The server push approach avoids these idle periods, since the server can always begin sending the next chunk immediately, and a single continuous flow will make much more efficient usage of the network for the same volume of data.

For client pull systems there is also the option to use a separate TCP connection for every chunk or request each chunk through a single long lived connection. Although each chunk does not imply a new TCP handshake with the latter approach, connections left idle for long enough will still lower their congestion window and have to build up speed when they begin sending again. It is not clear how this will affect the dynamics of an ABR flow. Finally, a third approach, which can be viewed as a hybrid of client pull and server push, is to have the client make requests only when a rate change is necessary. In this case the server continues sending at the previous rate until it receives a request for a new rate, which will be used after the next chunk boundary. This approach offers the scalability of client pull with less TCP connection overhead, and also allows the server to continue sending data without idle periods spent waiting for new requests. I refer to this approach as client pull with selective requests.

After choosing between these options, developers can then take a number of paths to implementing an ABR system, depending on other important design choices that have to be made. These include the implementation of various components required to perform bitrate adaptation and an appropriate way of formatting content into chunks. Again, the implications of these choices for performance and user experience are not well understood and I use this survey as a chance to investigate some of the key decision areas.

2.3 Application Components

An ABR system can be viewed a set of components that work together to perform adaptation logic and provide a streaming service to the user. The transfer mechanism being used will determine whether certain tasks are handled by the client or the server, and therefore which set of components are required and how they should be implemented. Core components include a bandwidth estimator, to monitor transfers and calculate rates, and a rate chooser, to select a suitable encoding rate given the current state of the system. For client pull systems a request scheduler is also needed to handle the timing and sending of requests to the server. Each of these components are described in more detail, along with common implementation approaches, in the following:

Bandwidth Estimation

The simplest and most obvious approach to bandwidth estimation is to base estimates on the transfer rates for each chunk, and this is what most commercial systems appear to do. At the client, this can be achieved by simply monitoring download rates, and on the server by timing the interval between TCP's data acknowledgements for the first and last packets of a chunk. However, TCP's behaviour, mostly stemming from congestion control mechanisms, hides much of the network's true state and leads to a number of issues for rate based bandwidth estimation. Two problems in particular are instability and underestimation, which are the focus of much of the discussion in [3]. Underestimation of the available bandwidth occurs as a result of TCP's cautious approach to sending data, starting off slowly and only increasing the sending rate gradually as packets are acknowledged by the client. The problem is worse for smaller chunks, which have less time to build up and maintain a higher sending rate, so that selecting a lower rate due to underestimating the bandwidth only compounds the issue. This is referred to as the ‘downward spiral effect’ in [3].

Instability results from the tendency for TCP’s sending rate to vary over time, even in the steady state but particularly when competing flows are present and packets are being lost regularly. This means transfer rates observed over short time spans will fluctuate, which could also translate into unnecessary fluctuations in video quality if the developer is not careful. A simple measure to counteract this problem is to introduce an averaging function over recent transfer rates to smooth out the bandwidth estimate. For example, [1] consider using the harmonic mean of the 20 most recent transfer rates, and it seems likely that some commercial systems will take a similar approach. There is a downside to smoothing the bandwidth estimate like this, however, which is that genuine changes in network conditions will take longer to filter through, and the system may not be able to react in time to prevent a stall.

Encoding Rate Selection

Having estimated the bandwidth, the next step is to consider the overall state of the system and choose a suitable encoding rate for the next chunk. A basic approach is to choose the highest available rate that is lower than or equal to the bandwidth estimate. A more sophisticated approach might consider recent changes and whether the rate is increasing or decreasing. For example, one may wish to lower the rate more aggressively and back off exponentially while growing linearly, allowing for steady growth when conditions are favourable while ensuring that the system reacts quickly to congestion when it is detected.

In [1], the authors suggest a statefull approach to encoding rate selection, which tries to mitigate the problems associated with underestimation and the bias towards lower encoding rates. With this approach, the number of bandwidth estimates required to trigger an up-switch is proportional to the current encoding rate. It should therefore be easier for the system to move away from lower encoding rates, where smaller chunk sizes can make it difficult to get a high bandwidth estimate. On top of this, each up or down switch can only be to the next rung on the ladder, even when bandwidth estimates suggest a greater change is possible. On the client there is also playback information available to help guide this decision. If the playback buffer is small then the situation is more urgent and the rate should be lowered immediately, despite the risk of making an unnecessary change.

Request Scheduling

For client pull based systems a request scheduling strategy is also needed. The impatient approach of requesting each new chunk immediately after the previous one has finished downloading may not be optimal. It is easy to imagine a situation where a client greedily buffers up lots of content at a low encoding rate when it could have waited a little longer to find better network conditions and use a higher rate. On top of this, over buffering can be considered wasteful from the hosts point of view if a user decides to leave prematurely. This issue is also considered in [1], which suggests scheduling requests periodically to maintain a certain duration of playback buffer. The authors find, however, that the periodic nature of requests, resulting from this scheduling approach, can lead to synchronisation issues and unfair allocation of encoding rates between competing ABR flows. In the end they show that introducing a small random offset to the target duration on each request is enough to mitigate this problem.

These are the main components required to implement and ABR streaming system, and I have only given a brief overview of some of the different options available. Due to limitations on the number of simulations I can run and cover adequately in this dissertation, I have chosen to focus on investigating the implementation of both bandwidth estimation and request scheduling components. I describe in more detail exactly which implementations I consider in Section 3.3.

2.4 Media Format

A final aspect of the system requiring careful consideration is media formatting. When preparing content for ABR streaming one must choose a chunk duration and a suitable set of encoding rates to make available. Both choices can affect the systems behaviour, however there have been no studies looking specifically at these parameters and how they impact the dynamics of an ABR stream.

From the information that is available, most commercial systems use a chunk duration in the 2 second to 10 second range. Using a short chunk duration makes sense, since the system must be able to react quickly to congestion, but making it too short, less than 2 seconds for example, can leave the system bogged down with overhead traffic generated from the requests. A shorter chunk duration will also result in more idle periods as the server waits for requests, which I will later demonstrate are damaging to performance in a client pull based system, and exacerbate the problem of bandwidth underestimation. Longer chunks, on the other hand, will have time to settle at a higher rate and make better use of bandwidth. Due to the way video encoding schemes work, longer chunks will also achieve better compression ratios and result in less network traffic overall. Video content is compressed by selecting landmark frames and then storing only the changes found in successive frames. The landmarks, called I-frames, are sent periodically and contain information for an entire frame. Compression is then achieved by sending deltas, called P-frames, which only store information on what has changed since the previous frame. Since TCP provides lossless transport, I-frames are only required at the beginning of a chunk. This means doubling the chunk duration will halve the number of I-frames required, resulting in greater compression. Having a chunk duration much longer than 10 seconds, however, starts to defeat the purpose of adaptation since the system must be able to switch rate quickly, if needed, and that can only happen at chunk boundaries.

The set of encoding rates that are available will also affect the dynamics of an ABR stream. More rates might mean a more granular system that is able to react to changes on the network, or it could just lead to more unnecessary rate switches. There are also practical concerns since the volume of content hosted on the server is proportional linearly to the number of encoding rates available. The content is duplicated once for each available rate.

2.5 Commercial Systems using ABR

HTTP adaptive bitrate streaming has become a popular approach to delivering video over the Internet and many content providers now offer such a service. These include; Netflix [13], Hulu [14], Vudu [15], YouTube [16], and BBC iPlayer [17]. Several well known software giants have also implemented ABR enabled players, including Microsoft's Smooth Streaming [18, 19], Apple's Quicktime [20], and HTTP Dynamic Streaming from Adobe [21]. Implementation details vary. For example, [3] presents a detailed study of three of these services. Each uses a different set of encoding rates, and chunk durations vary between four and eight seconds. All three use a client pull mechanism. However, only one client requests each chunk through a new connection while the other two maintain a persistent TCP connection between client and server. Further implementation details are not clear, but it can only be assumed that similar differences exist. Recent studies have shown that many of these existing ABR systems suffer from problems relating to fairness and encoding rate stability when clients compete for bandwidth on a congested network [3, 1, 4, 5, 6].

2.6 Summary

In this chapter I have discussed various Internet video architectures and outlined the motivations behind the development of HTTP based adaptive bitrate streaming technologies. I also discussed some of the ways in which ABR systems can be implemented and how different approaches have the potential to affect the overall system performance. This included the tradeoffs between client pull and server push implementations, the importance of different components required for rate adaptation, and the effects of content formatting parameters. Many commercial players using ABR streaming are now available, with various approaches to implementation, and studies have shown that a number of them suffer from issues relating to fairness and stability when networks are stressed and multiple flows compete. Underlying many of these issues is the fundamental difficulty of estimating bandwidth on top of TCP. Yet despite much interest and a catalogue of observed problems, the effects of certain design choices on the systems' performance are still not well understood. In particular, there has been very little focus on studying server push based systems, and likewise for the consequences of decisions concerning media formatting parameters. In short, there does not exist any kind of concrete guidance for those wishing to implement an ABR system, who will at some point have to make these types of decision.

To tackle this problem I have carried out a survey of the area using simulation software. In the next chapter, I outline my approach to carrying out this survey, describe precisely which areas I will investigate, and outline the simulations I will look at in later chapters.

Chapter 3

Survey Methodology

ABR streaming is becoming a popular way to deliver video content over the Internet, but there are many ways to implement these systems and no clear guidelines exist for developers. To tackle this I have carried out a survey of the area, using the ns-3 network simulator to evaluate and compare different approaches. The four main areas I will investigate are the tradeoffs between push and pull based systems, the effects of media formatting parameters, bandwidth estimation techniques, and different request scheduling strategies. Systems are evaluated according to measures of their fairness, encoding rate stability, and efficiency.

In this chapter I describe the methods I used to carry out this survey. I will discuss the merits of a simulation approach, along with its limits and the need for validation, and the software I use. I then outline the simulations covered in later chapters, including detailed descriptions of the variables I investigate, how simulations are setup, and how their output is analysed.

3.1 Simulation

There are a number of ways to carry out a survey like this, each with advantages and disadvantages. One approach would be to study existing commercial ABR systems, possibly with additional monitoring and reporting functions added to the client, as in [3] and [1]. This has the advantage of being genuine. There is no need to worry about creating an accurate model of the properties of the network, its traffic, or the applications themselves. The downside is that having little control over the network can make it difficult to find suitable conditions for experiments that can be easily repeated. On top of this, experiments run in real time, lasting for the same duration as the content, and cannot easily be parallelised. Running two experiments simultaneously means setting up and managing two separate physical clients on appropriate networks. Another issue is that this approach is restricted to investigating existing services, and there may not be perfect information available on how these are implemented.

A second option is to create a laboratory setup, using software to imitate network properties such as bandwidth and end to end delay, then write and investigate your own applications. This gives the experimenter full control over both the network and the applications being investigated. Experimenters using this approach will be responsible for generating all network traffic, including competing flows and realistic background traffic, but applications still run on a real network stack. Since the experimenter is in control of the traffic, conditions can be tweaked to analyse different scenarios and experiments can easily be repeated. But physical restraints on what can be achieved remain. Parallel instances require more machines and experiments still run in real time.

A third approach, which I have chosen, is to run experiments using network simulation software. With simulation, rather than probing something that already exists, the idea is to recreate a detailed enough model of it in a software environment that still displays the relevant properties of the original and so can be studied with relevance. The advantage of simulation is that the entire process can be controlled programmatically, the difficulty is creating sufficiently accurate models to study. Using network simulation software means experiments can easily be automated and repeated. Thousands of simulations can be run with a single command, and in parallel using multiple cores or distributed over a network. Experiments run inside a simulated time frame and can be made faster in real time by applying more processing power. Since large scale repetition is now feasible, statistics can be used to filter out noise and make results more robust. Finally, in a simulation, any interesting aspect of the environment can easily be monitored and traced, with the output piped through further analysis scripts or stored for later processing. But simulation is also well known to be difficult [22, 23]. Failure to accurately model the subtleties of network protocols such as TCP will render the simulator ineffective and its results irrelevant. This means it is necessary to validate simulations by comparing results with other studies and examining the simulator's behaviour under well known conditions. It is also important to remember the role of simulation and how it can fit into a bigger picture, and be honest and forthright about its limitations. I present validation arguments throughout the later chapters by referring back to other studies in this area and discussing how my results compare to theirs whenever possible.

3.2 Software

For simulation software I have chosen ns-3, a discrete event network simulator written in C++ [24]. Open source, extendible, and actively maintained, ns-3 allows researchers to write networked applications through a non blocking sockets interface. These can then be 'installed' on nodes connected together to form the desired network with Python or C++ programs. Ease of use and active development, with a large open source community surrounding it, make ns-3 a suitable choice for this project. Its popularity in the research community also means the existing models in ns-3, such as TCP implementations and queueing algorithms, have evolved under scrutiny and are already well tested.

For my work, I have extended ns-3 with a module containing generic client and server application models and various implementations of the core ABR system components. Different ABR systems can be created by matching clients and servers with the appropriate components plugged in. I also use a scripting tool that allows me to specify network and application parameters in external text files, which are then parsed inside ns-3 to create and run the corresponding simulation. To facilitate the running and analysis of several thousand simulations I have developed a set of python and shell scripts. These include a framework for running simulations in parallel, over multiple machines or a single machine with many cores, and a set of scripts for processing and analysing the output of simulations to produce graphs and summary statistics. Graphs are generated using the matplotlib Python plotting library [25].

3.3 Variables

In the previous chapter I introduced the various components and parameters of an ABR streaming system. These included the client and server transfer mechanism, bandwidth estimation and rate selection, request scheduling, and media formatting. I will now define specifically which components and parameters I investigate, and how they can be combined together in different ways to implement an ABR system.

For transfer mechanism I define four separate cases: client pull through a single connection (pull single), client pull through multiple connections (pull multiple), server push (push), and client pull with selective requests (pull selective). These allow me to demonstrate the effects of both push versus pull and using a single TCP connection versus multiple TCP connections with client pull. I also hope to demonstrate that the hybrid pull selective approach shares the performance advantages of server push, with lower overheads and the scalability of client pull. Existing ABR systems favour the client pull approaches and there is relatively little work available on server push. The flexible nature and scalability of client pull may outweigh performance advantages of server push, but the hybrid approach could offer an attractive compromise between the two.

The application components I consider are bandwidth estimation and request scheduling. For bandwidth estimation I compare three alternatives: using the previous chunk's rate estimate as the bandwidth estimate (previous chunk), using an exponentially weighted moving average (EWMA) of all previous chunk estimates, and using the harmonic mean of the last 20 chunk estimates (HM20). A previous study has already shown that smoothing the rate estimate, using the harmonic mean of the 20 most recent chunk transfer rates, can improve encoding rate stability [1]. This offers an opportunity both to validate that conclusion and test the method under new conditions. At the client, transfer rates are obtained by simply measuring the download rate directly. On the server the task is more complicated, and measuring the time between successive writes at the application layer is not accurate enough. If there is sufficient space in the TCP send buffer then a large volume of data can be sent in a single write at the application layer, but only a fraction of this is actually forwarded on to the network immediately. Instead I measure the time between TCP's data acknowledgements for the first and last packets of a chunk, which is more accurate since the acknowledgements are clocked by the network. This is not difficult to implement in ns-3, but could also feasibly be implemented in a real world scenario by adding a kernel module that notifies the application layer when TCP's acknowledgements arrive. For example, [26] discusses similar ideas, and [27] explores the concept of integrated layer processing.

For the request scheduling component, again, I compare three implementations. One that sends each request immediately after the previous chunk has downloaded (immediate), one that send requests periodically to maintain a 30 second playback buffer (periodic), and one that sends requests periodically, as above, but with a small random offset introduced to avoid repetitive cycles and synchronisation problems (random periodic). Periodic request scheduling is designed to try to avoid problems associated with over buffering, but has also been found to cause further issues relating to fairness when ABR flows compete. This is demonstrated in [1], which also shows that random periodic request scheduling can help to mitigate these issues. In Section 2.3, I also discussed different strategies for selecting an encoding rate given the bandwidth estimate. For practical reasons, I was not able to investigate any of the different approaches described there, and all applications studied in this dissertation simply select the highest rate that is less than or equal to the most recent bandwidth estimate.

All three bandwidth estimation approaches are applicable under the four transfer mechanisms. Request scheduling components only apply with pull single and pull multiple. For server push there is only the initial request from the client and with pull selective the flows behaviour is not tied to the arrival of new requests, which are likely to be well spaced since they only occur when a rate change is needed. Figure 3.1 summarises the different ways in which these approaches can combine to form an ABR streaming system. Each cell in the diagram represent one feasible implementation of an ABR system.

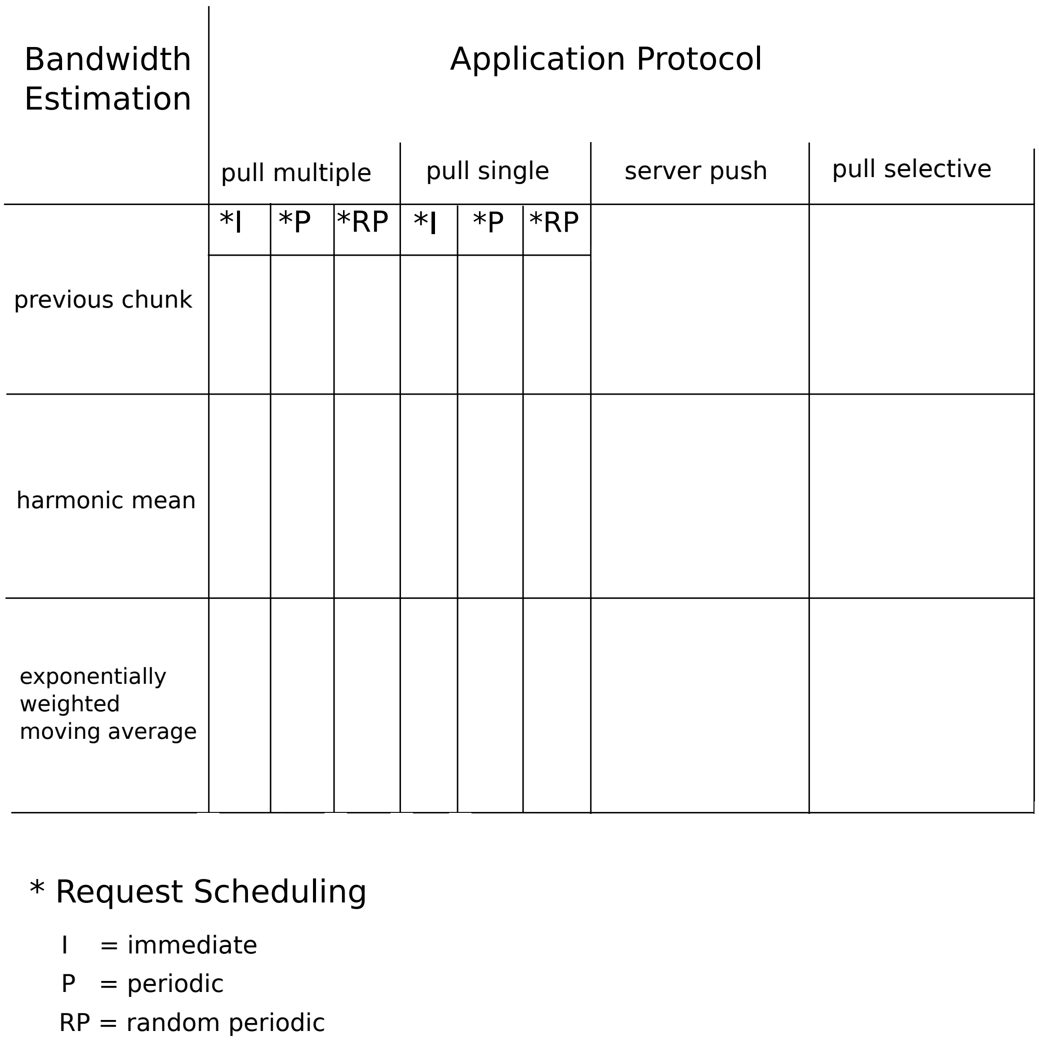


Figure 3.1: Application implementation space

The final set of parameters I investigate concern media formatting. The effects of chunk duration and encoding rate spacings have not been well researched, and different approaches can be seen in the wild. I compare 2 and 10 second chunk durations to highlight the benefit of increasing chunk duration with client pull systems. Longer transfers reach higher speeds and give more reliable rate estimates. The values used were chosen to match the lower and upper end of both what seems sensible and what can be found in commercial systems. For encoding rates I have chosen a set of 7 linearly spaced rates (600, 1200, 2000, 3000, 4000, 5000, 6000 kbps) and a smaller set of 4 exponentially spaced encoded rates (600, 1200, 3000, 6000 kbps). More choice intuitively sounds better, but more rates means more space on the server and more opportunities to change rate. Linearly spaced rates are an obvious first choice, but I also felt that it would be interesting to try covering the same range with a smaller set of rates, spaced exponentially. Exponential spacing should mean that the system is naturally more cautious at lower rates, always requiring more of a change to switch up than to switch down. The precise values are chosen to coincide with those available in a data set containing real content prepared for ABR streaming [28]. Both chunk durations and sets of encoding rates are interchangeable giving four sets of media parameters. All four sets of media parameters can be used along with any of the applications described above.

3.4 Network and Traffic Models

In order to properly evaluate and compare different ABR systems, they each need to be tested under the same set of scenarios reflecting typical real world usage. This means defining an accurate model of the network, and realistic cross traffic models to compete against ABR flows in different scenarios. Background noise traffic can also be used to introduce some randomness and make simulations less deterministic. In this section I describe the network used in my simulations and how it is configured, then discuss the different types of artificial traffic that I generate.

The network setup I use, shown in Figure 3.2, is intended to model the typical conditions found when users stream video to a machine on their home network. On the far right I have a number of potential clients connected via 100 mbps Ethernet cables to the home router device. The bottleneck link connects the home router via and 8 mbps ADSL line to the local exchange. The UK average is slightly higher, at around 12 mbps [29], but 8 mbps provides a nice bottleneck width for the range of encoding rates I am using. To select an appropriate delay for the bottleneck, I measured round trip times to the next hop beyond the router from machines on three different home networks. Beyond the exchange lies a long, high capacity link representing the backbone network connecting outlying regions to a major Internet exchange center, for example Glasgow to London. Various potential content hosting servers are then connected to this second routing point via high capacity links, each to be matched with a single client at the other end. Necessarily, this model hides much of the complexity of the network beyond the local exchange. However, the focus is on what happens when streams compete over a bottleneck link, and it is assumed that the upstream network can easily handle the load.

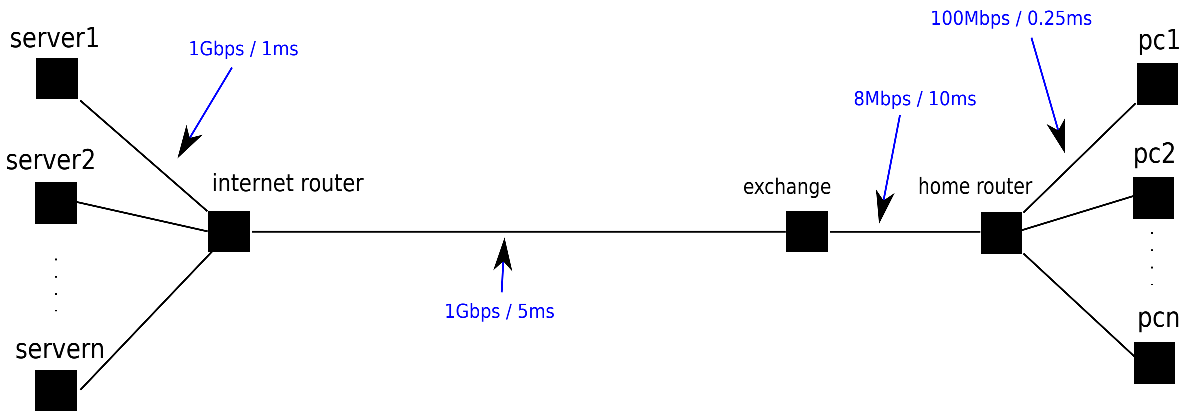


Figure 3.2: Network setup

Queues on routers are sized according to the bandwidth delay product rule and handle packets using a tail drop algorithm [30, 31]. ns-3 is relatively new and has limited TCP models, with some features still missing such as SACK and Nagle's algorithm [32, 33]. The most modern implementation available, which I use in all simulations, is TCP New Reno [34]. New Reno contains most of the core features of modern TCP responsible for its effects on network traffic, including the four standard congestion control mechanisms (slow start, congestion avoidance, fast retransmit, and fast recovery [10]). More information on the implementation and verification studies can be found on the ns-3 website [35].

Having modelled the network, I need to generate realistic traffic to send over it. My simulations use three main types of traffic: traffic generated by ABR streams, continuous and long lived TCP flows imitating large file transfers, and randomised 'bursty' flows to mimic the effects of background noise. Each application is tested in six different scenarios. This covers two different types of competition traffic, separate instances of themselves and continuous long lived TCP flows, each with one, two, and three flows competing against the main ABR flow. Each flow is between a single client and server, as depicted at either end of Figure 3.2. More competition traffic models are possible, and ideally traffic could be generated based on a broad study of the characteristics of home user network traffic, but with the time and resources available these scenarios make for a sensible starting point.

The final ingredient for a useful simulation is background noise. On its own, the simulator is completely deterministic, and a simulation run twice with the same parameters will yield identical results on both occasions. This makes repetition and statistical analysis impossible, which in turn makes it very difficult to trust the results as being representative of reality. To solve this issue I have taken two steps. The first is to add a small random offset to the starting times defined for applications inside simulation scripts. This allows me to define roughly when an application will start in a given simulation, but ensures they do not always start at exactly the same time relative to one another. Randomised start times is enough to make repetitions of the same simulation behave differently, but the destiny of every flow is still predetermined at the beginning of a simulation. The second step I have taken to reduce determinism is to add an extra flow, with randomly determined behaviour, to every simulation. These are lightweight, sending only enough packets to interfere at the routers but not dominate other flows, and generated by a simple client continually requesting a random number of bytes with random length idle intervals between. The number of bytes and length of interval are both bound by maximum and minimum values, which can be modified to tweak the volume and behaviour of flows. With some basic calibration I found that 1 to 10 kilobyte transfers at 1 to 10 second intervals is enough to add some non-deterministic variation to simulations, without having a noticeable effect on the overall outcome. However, this also ties in with my previous point about possibilities for further competition traffic models. A better approach would be to generate this kind of cross traffic statistically, based on a study of real traffic, but with the time and resources available my solution offers a reasonable compromise.

3.5 Evaluation

The output of simulations is analysed to characterise each application in terms of fairness, stability, and efficiency. In this section I describe the various traces produced by each simulation, and how multiple traces from repetitions of a single simulation can be analysed and collapsed into summary statistics that describe the behaviour of that application.

The two main types of output I consider for each simulation are packet traces at endpoints and chunk records from each of the clients involved. A packet trace is simply a record of the times at which packets were received by the client along with their size. This can be used to plot the throughput for each client over the course of the stream, but more importantly by considering the number of bytes received by two or more competing clients over a fixed time interval a measure of fairness can be generated. Each client also maintains a record for every chunk in the stream. This includes the encoding rate that the chunk was sent at, along with a note of the times at which the request was sent (for client pull), the first bytes were received, and the last bytes were received. Looking at the pattern of values for consecutive chunks' encoding rates offers a way to calculate stability. Chunk records can also be used to generate encoding rate distributions that give further insight into the stream's behaviour.

In addition to packet traces and chunk records it is also possible to measure and plot a number of interesting aspects of the system for extra insight. A record of packets being enqueued, dequeued, and dropped at the incoming interface is available for each node. Of particular interest here is that of the node representing the local exchange where we expect to find the queue filling up and packets being dropped as competing streams reach the bottleneck. It is also possible to plot the value of TCP's congestion window at each of the server nodes, useful for validating the behaviour of ns-3's TCP models as well as understanding the behaviour of flows.

Finally, I trace playback buffer occupancy over time for each ABR client, and use this information to verify that adaptation is working and applications are not stalling in situations where fixed rate flows would. Buffer occupancy increases whenever a new frame is received, whilst also decreasing at a constant rate to simulate playback. In a real system, with content encoded using I-frames and P-frames, I-frames will be larger than P-frames and each P-frame will also vary in size depending on what is happening on screen during those frames. Since I use dummy data and do not consider real content, these effects are not simulated and it is assumed that each frame is the same size and so each byte corresponds to the same increase in playback buffer occupancy. This simplification should not have enough of an effect to undermine the usefulness of my results, since the model still captures the behaviour of the system at the chunk level fairly accurately. With real content, the size of chunks will also vary, since the duration of each chunk is fixed but the actual encoding rate achieved will depend on what is happening on screen. I therefore have to assume that the average chunk size will normally be a good reflection of the targeted encoding rate. It would be difficult to simulate the latter effect accurately, using dummy data, without making some assumptions about the nature of the variance anyway, and I maintain throughout that future work using real content is needed to fully corroborate my results.

3.5.1 Calculating Fairness

Throughput records from two competing clients can be used to calculate a fairness value for that simulation. In a completely fair system we would expect to see two competing flows receive roughly the same number of bytes over any reasonable length of time. Fairness can therefore be measured by comparing the number of bytes received by each flow during the common interval in which all flows compete for bandwidth. Jain's fairness index provides a way of doing this, and is a commonly used metric for this type of work [36]. When a resource is shared amongst N competing users, each receiving x_n units, Jain's index allows us to calculate the fairness of that distribution using the formula shown in Equation 3.1:

$$J(x_1, x_2, \cdots, x_N) = \frac{(\sum\limits_{i=1}^{N} x_i)^2}{N * \sum\limits_{i=1}^{N} x_i^2} \tag{3.1}$$

This can be applied by treating the number of bytes received by each stream during the competition interval as the resource distribution. After repeating a simulation with the same parameters many times, I calculate a Jain's index value for each repetition and present the results as a boxplot for those parameters.

It is worth highlighting some issues regarding the behaviour of Jain's index. The first thing to note is that Jain's index for N values is bound between 1 and $1/N$. For example, for two competing flows, a Jain's index value of 0.5 represents the worst possible distribution of resources. For three competing flows, the worst possible distribution returns a Jain's index of 0.333... and so on. A Jain's index of 1 occurs when each value in the distribution is equal. The different lower bounds for different values of N mean that it is difficult to use Jain's index to compare two distributions with a different number of values, or in my case, two simulations with a different number of competing flows. A second issue with Jain's index is that it requires a significant disparity between the values in the distribution for the index to fall noticeably lower than 1. Figure 3.3, showing how the Jain's of two values behaves as the ratio of one value to the other changes, highlights this issue. For the index to fall to 0.9 requires that one value be half that of the other. This means that Jain's index values are often bunched near the top, and can be difficult to compare even when there are significant differences. To help tackle this I plot Jain's index values between 0.8 and 1 on the y-axis, since I never see it fall below 0.8 in any of my simulations.

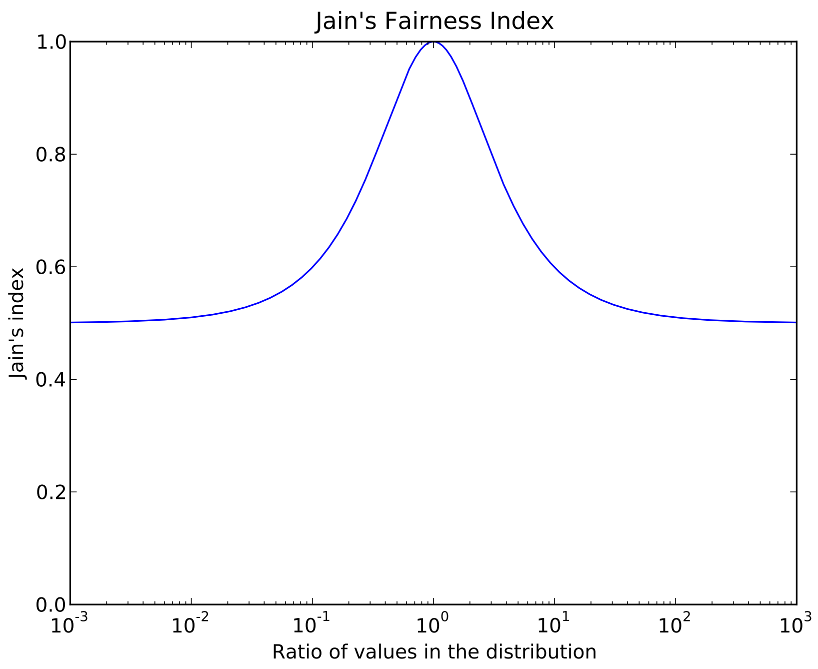


Figure 3.3: Behaviour of Jain's index for a distribution with two values

Despite these issues, Jain's index is widely used and easy to implement, and it would require a lot of work to validate an original metric. A final point to make about my approach to measuring fairness is that it will naturally report unfairness when a flow that is rate limited at the application layer, such as an ABR flow with idle periods between requests, competes against one that is not, such as a long lived TCP flow. In this case, it could be misleading to simply state that the ABR flow and long lived TCP flow do not compete fairly. However, I can still use this approach to highlight and quantify the difference between two types of ABR flow, one that is rate limited at the application layer and one that is not. The former should result in a lower Jain's index value when competing with a long lived TCP flow than the latter.

3.5.2 Calculating Stability

The stability of a stream refers here to a measure of how the encoding rate varies throughout the duration the stream. It is not enough to simply calculate some measure of the variance in rates seen, such as the standard deviation, since this does not take into account a chunk's location in the stream. For example, imagine a stream resulting in an even distribution between two different encoding rates. Such a distribution could be produced by a stream that only changes rate once in the middle, with the first half of chunks using one rate and the second half using the other, or by a stream where every chunk alternates between the two encoding rates. Clearly one is stable and one is highly unstable, but both would return the same standard deviation. Instead, I use the following formula where $N_{switches}$ is the number of rate switches and N_{chunks} the number of chunks:

$$S = 1 - \frac{N_{switches}}{N_{chunks} - 1} \tag{3.2}$$

If every new chunk switches rate from the previous, the fraction becomes 1 and the index is 0. Conversely, if there are no rate switches the fraction evaluates to 0 and the index will be 1. Note that the maximum number of rate switches is equal to $N_{chunks} - 1$, since the first chunk does not count as a switch. To evaluate stability, I apply this index to the main ABR flow of each simulation. Again, values for repetitions of a single simulation are collapsed into a boxplot showing the range of values seen for those parameters.

3.5.3 Encoding Rate Distributions

Despite not being sufficient on their own to gauge stability, for reasons explained above, encoding rate distributions can complement both the stability and fairness indexes and provide useful insight into what happened during the simulation. For example, it occurs later that a simulation gives an unexpectedly high stability index, but on inspection of the encoding rate distribution it becomes clear that this is only because the system performed so poorly that it rarely used anything other than the lowest possible rate. Encoding rate distributions are calculated by counting the number of chunks seen at each rate, over all repetitions of a simulation, then normalising to give the value for each encoding rate as a percentage of the total number of chunks transferred in all repetitions. A favourable distribution should have a narrow peak close to the fair share bandwidth.

3.5.4 Playback Buffer Occupancy Distributions

It is no good highlighting that a particular modification improves fairness or stability, without also demonstrating that it does not affect the system's ability to adapt and avoid playback interruptions. For this purpose, I calculate distributions of the amount of time the playback buffer occupancy spent in eight different states over the lifetime of a stream. The possible states considered are *equals zero*, *greater than 60 seconds*, and anywhere in one of the six

10 second intervals between those two. After tallying the amount of time spent in each of these states, over all repetitions of a single simulation, the results are presented as a distribution over all eight states, with height on the y-axis showing the time spent in that state as a percentage of the total duration of all simulations. For a simulation that continues to buffer without ever stalling, we expect to see no bar in the *equals zero* state, followed by a roughly even spread over the next six states, and a peak in the *greater than 60 seconds* state. This pattern arises as a result of the playback buffer spending roughly an equal amount of time in each of the 10 second interval states as it rises steadily. The peak in the final state happens because the values for several 10 second intervals are combined into a single value (i.e., (60s, 70s], (70s, 80s], and so on.)

3.6 Outline of Simulations

The remainder of this dissertation presents five different groups of simulations, each with a specific set of purposes. These are: calibration and demonstration of initial performance issues, investigation of the effects of chunk duration and encoding rate spacing, demonstrating the positive effects of using a smoothing function on bandwidth estimates, investigation of the effects of different request scheduling strategies, and finally a comparison of selected implementations chosen to combine the best features or match the parameters of systems studied in previous work. I will also explore and discuss the push vs pull tradeoffs throughout all of these chapters and try to highlight the benefits of the hybrid pull selective approach. The final chapter will recap the motivations and aims of the project, discuss what simulations from each chapter have shown, and summarize the conclusions and contributions of this work. The following list gives a more detailed overview of each of the five simulation chapters:

Chapter 4 - Baseline Simulations

In Chapter 4, I present the first set of simulations, which are chosen to highlight problems and set a baseline from which to improve. Four applications are tested covering each of the four transfer mechanism, but using only the most basic ABR components where they are applicable (previous chunk bandwidth estimation and immediate request scheduling). A two second chunk duration is used with the set of seven linearly spaced encoding rates. By competing each of these applications against both separate instances of themselves and long lived TCP flows, with varying numbers of flows, I will introduce three distinct points. The first is that client pull approaches are less efficient than the server push and hybrid approaches, and are noticeably discriminated against when competing with long lived TCP flows. The second is that all four mechanisms behave poorly in terms of stability and exhibit highly variable video quality. The final point is that there is little noticeable performance difference between server push and pull selective.

Chapter 5 - Media Formatting Parameters

Chapter 5 extends the baseline simulations by simulating the same four applications studied in Chapter 4, under the same scenarios, using different media parameters. In this chapter I introduce the ten second chunk duration and smaller set of encoding rates. Simulations from chapter 4 will be repeated using each of the three new combinations this gives. Here I hope to demonstrate two things. The first is that using a longer chunk duration can improve the performance of client pull and help mitigate the problems it has regarding fairness and bandwidth efficiency. The second aim of this chapter is to make the point that using fewer encoding rates will naturally lead to more stable streams.

Chapter 6 - Smoothing Functions

In Chapter 6, I begin looking at the implementation of different ABR components, starting with bandwidth estimation. Here, I will demonstrate the effect of applying averaging functions over a series of recent transfer rates, in order to smooth bandwidth estimates, and show how this can improve poor encoding rate stability witnessed in previous chapters. I will consider both a harmonic mean of the 20 most recently observed transfer rates, and an exponentially weighted moving average of all previous transfer rates.

Chapter 7 - Request Scheduling Strategies

In Chapter 7, I continue looking at ABR components, this time investigating the use of more sophisticated request scheduling strategies with client pull based systems. I consider both periodic request scheduling, which is intended to avoid over buffering, and random periodic request scheduling, which is intended to mitigate fairness and synchronisation issues introduced by periodic request scheduling. In the beginning, I have difficulty reproducing results from [1], but after changing my parameters to match those used in that study more closely, I am able to replicate their results more closely, and also provide evidence to suggest that the problems they encounter may be sensitive to certain parameters.

In Chapter 8, I present validation arguments for my simulations, and discuss how my results relate to other work in this field. Chapter 9 ends with a summary of my findings and conclusion of the project.

3.7 Summary

In this chapter, I introduced my methods for carrying out a survey of different approaches to ABR video streaming. I chose to use simulation software, as an alternative to lab methods, because of the opportunities it offers to run large scale experiments, and the ns-3 network simulator was an attractive choice mainly due to its open source nature, ease of use, and popularity amongst other researchers. I then discussed the specific implementations of ABR systems that I will investigate, and how my survey will help developers make informed decisions when implementing ABR systems by shedding light on three key areas where little information is currently available. These were, the tradeoffs between client pull based systems and server push based systems, the effects of more sophisticated ABR components such as bandwidth estimators and request schedulers, and the effects of media formatting parameters. I also introduced a hybrid option to the push versus pull decision, which I hope to demonstrate can match the performance of server push with the scalability of client pull. I then discussed the output of simulations, and explained how I evaluate them using Equations 3.1 and 3.2 to measure fairness and stability. Encoding rate distributions and playback buffer occupancy distributions are also used to give further insight and detect playback interruptions. Finally, I outlined the simulations that are covered in each of the following chapters. In the next chapter I begin my survey with the first set of simulations, setting a baseline by investigating the four most basic applications with a 2 second chunk duration and linear encoding rates.

Chapter 4

Baseline Simulations

Before setting out to suggest ways to improve ABR systems, I first need to demonstrate that problems exist. In this chapter I investigate and compare the most basic applications that can be implemented under each of the four transfer mechanisms described in Chapter 3, using a default set of media parameters, in order to demonstrate the inadequacy of these simplistic approaches. These results will then act as a benchmark for applications, against which applications studied in later chapters will try to improve, by using different parameters and introducing more sophisticated ABR components. Throughout the chapter, I will try to emphasize three key points that the simulations presented in this chapter demonstrate. The first is that the two client pull applications are less efficient than the server push and pull selective approaches, and are noticeably discriminated against when competing with continuous TCP flows. The second point I will demonstrate is that all implementations suffer from poor encoding rate stability, and the final one is that the hybrid pull selective approach behaves almost identically to server push and shares its performance advantages over client pull. I will demonstrate these points by analysing the output of 2,400 simulations using a combination of encoding rate distributions, fairness box plots, and stability box plots. In the following sections I describe the specific details of the simulations covered in this chapter, before presenting their results and ending with a discussion and summary of my findings and conclusions.

4.1 Details

I begin by comparing four application models representing the most basic implementations under all four transfer mechanisms. These are: client pull through multiple connections (**pullm-i-prev**), client pull through a single connection (**pulls-i-prev**), server push (**push-prev**), and client pull with selective requests (**pullsl-prev**). Each application uses ‘previous chunk' bandwidth estimation and, where applicable, 'immediate' request scheduling. This accounts for the applications represented by the first, fourth, seventh, and eighth cells from the left in the top row of Figure 3.1 in Chapter 3, and allows me to make an initial comparison of different transfer mechanisms and demonstrate areas where all approaches fall short. Smoothing functions and periodic request scheduling strategies will be explored in Chapters 6 and 7.

Simulations in this chapter all use the same set of default media parameters. These are the **2 second** chunk duration and linear encoding rates **(600, 1200, 2000, 3000, 4000, 5000, 6000 kbps)**. For applications, choosing a default implementation comes naturally. The most basic implementation under each transfer mechanism, with only basic ABR components (i.e., immediate request scheduling and previous chunk bandwidth estimation), is the obvious choice for a starting point, but the same logic does not apply to media formatting parameters. Without knowing how the applications will behave beforehand, there is no compelling reason to begin with a 2 second chunk duration rather than 10 second chunk duration or vice versa. Likewise for the available encoding rate sets. These default media parameters were chosen simply because they seemed like an intuitive place to begin, but this decision proves useful in the end since the small chunk duration and linear encoding rates exacerbate stability and fairness problems. This means I can more clearly demonstrate the differences between client pull and server push, and the positive effects of using better ABR components in later chapters. In the next chapter I will extend these simulations to use the 10 second chunk duration and exponential encoding rates.

Each application model is simulated competing against both one, two, and three separate instances of itself, and one, two, and three continuous TCP flows. Note that in plot titles, 'N flows' means that there are N total flows competing. This means either N ABR flows, or 1 ABR flow and $N - 1$ continuous TCP flows. All ABR streams transfer 10 minutes worth of content, which is 300 chunks at 2 seconds per chunk, and long lived TCP flows run until all ABR flows have stopped. Four applications models, tested in six different scenarios, with each individual simulation scenario repeated 100 times, gives 2,400 simulations in total. Output from all repetitions of a single simulation scenario is then collapsed into summary statistics characterising the behaviour of that application in that scenario. In the following sections I present three different analyses of these simulations, looking at fairness, stability, and encoding rate distributions.

4.2 Fairness

Figure 4.1, on page 40, shows six graphs comparing the fairness of my first four applications under six different scenarios. Figure 4.1a shows applications competing against separate instances of themselves, with increasing number of competing flows moving down. In Figure 4.1b applications compete against continuous TCP flows, again with increasing number of competing flows going down the column. Fairness is calculated using the Jain's index, Equation 3.1, applied to the total number of bytes transferred by each competing flow during the interval when all flows are competing. Since each simulation is repeated 100 times, the index is calculated once for each repetition and the result for that simulation given as a box plot.

For ABR vs ABR simulations in Figure 4.1a, with 2, 3, and 4 competing flows, there is no obvious sign of any discrimination, with a Jain's index of close to 1.0 for every application. This is not surprising, since all applications are doing the same thing and Jain's index requires a lot of discrimination for the value to fall noticeably lower than 1. However, for ABR flows competing against continuous TCP flows, it is clear that the two client pull applications have a noticeably lower Jain's index than the others, which remain close to 1.0. Since server push and pull selective flows always send continuously, it makes sense that there is no discrimination when competing against other continuous TCP flows. TCP is designed to allow multiple flows to compete fairly over a bottleneck, assuming they have a similar round trip time and use the same variant of TCP. But for pull multiple and pull single, the idle periods between chunks damage efficiency and allow the continuous flows to dominate. This seems to be due to the fact that TCP's congestion control mechanisms will lower the sending rate by resetting the congestion window, putting TCP back into the slow start phase, after a sufficiently long idle period or when a new connection is created [37]. In the slow start phase, a TCP flow has to build up its sending rate by increasing the congestion window each time a packet is acknowledged by the receiver. The congestion window determines how many packets can be in flight at a given time, and therefore increasing congestion window means TCP can send data at a higher rate. But while the ABR flow is in slow start, any continuous TCP flows sharing the link will now increase their sending rate to fill the extra capacity that now exists since the ABR flow lowered its rate.

This has two consequences. The first is that the interaction between the ABR flow and the continuous TCP flows can damage the initial re-growth of the ABR flow. While the ABR flow is idle, the continuous TCP flows will continue sending packets and filling the buffer at the exchange router. When the ABR flow starts sending again it is likely to experience packet loss because of this, making it more difficult for it to increase its congestion window and become established again. The second consequence, which is more obvious, is that the time spent idle and time spent increasing the rate again, during all of which competing flows have maintained their rate, immediately results in fewer bytes being sent by the ABR flow, compared to the continuous flows, over the same period of time. Even without considering the possibility of continuous flows expanding their rate and actively damaging the growth of an ABR flow in slow start, the on-off nature of the ABR flow makes them inherently less efficient. With a chunk duration of only 2 seconds the idle periods are both frequent, and significant compared to the overall duration of each chunk transfer, and TCP never has enough time to settle at a high rate for any length of time. In Chapter 3 I will demonstrate that increasing the chunk duration has a significant positive effect on the efficiency of client pull ABR flows and their ability to compete with continuous TCP flows.

A final point worth noting from these graphs is that there is very little difference between the server push and pull selective applications in any of the scenarios. This is not surprising because both approaches maintain a continuous TCP flow and the only real difference between them is where adaptation logic is handled. This does not seem to change the dynamics of the flow, but does make the pull selective approach, where the client handles ABR calculations, much more scalable than server push.

4.3 Stability

Figure 4.2, on page 41, shows stability box plots given in the same format as the graphs from Section 4.2. Each compares four applications, competing against themselves in Figure 4.2a and against continuous TCP flows in Figure 4.2b, with increasing amounts of cross traffic going down the columns. Stability is calculated for the main ABR flow in each simulation, using Equation 3.2. Again, a value is calculated for each repetition of a simulation and the result for that simulation given as a box plot of 100 different values.

The first thing to notice from these graphs is that no application performs terribly well under any circumstances. The most stable simulation sits at just above 0.8, giving an average of one rate switch for every 5 to 10 chunks. This is a fundamental problem with ABR streaming systems, and has been observed by a number of studies looking at existing commercial solutions [1, 4, 5, 6]. The issue arises from the fundamental difficulty of estimating the available bandwidth above the HTTP layer [3]. At this level the network is opaque, and TCP's congestion control mechanisms mean that transfer rates only give a rough view of what is happening. Rates are likely to fluctuate over short time scales, making it difficult to choose a suitable encoding rate for the future based on only the transfer rate of the previous chunk.

Poor decisions can also exacerbate the problem and set up a negative feedback loop. For example, underestimating the available bandwidth and choosing a low rate will result in a shorter transfer, making it more difficult to get an accurate bandwidth estimate. For client pull systems shorter transfers also suffer more from the problems described in Section 4.2, which can only make things worse. This is referred to as 'the downward spiral effect' in [3], which investigates the problem and finds that it is exhibited by several commercial players.

Again, server push and pull selective behave very similarly throughout, and both are less stable than the client pull applications. At first this may seem like a victory for client pull, but encoding rate distributions in the next section reveal a different story. In reality, the client pull applications often are only more stable in some circumstances due to their poor performance imposing limitations on their choice of encoding rate. There is less variation since those systems rarely achieve the higher rates.

4.4 Encoding Rate Distributions

Figure 4.3, on page 42, shows encoding rate distributions for all the simulations in this chapter. The rows and columns are as before, with a colour coded distribution for each of the four basic applications inside each graph. Each individual distribution shows the normalised distribution of encoding rates for chunks transferred in 100 repetitions of that simulation.

Encoding rate distributions give further insight into the trends witnessed in fairness and stability plots. For example, although graphs in the Section 4.3 showed that the pull multiple and pull single applications were more stable in many scenarios, the distributions in Figure 4.3 show that this is mainly because they achieve a much smaller range of encoding rates, at the lower end of the spectrum, due to their poor performance. Pull multiple is particularly stable in the top two graphs of Figure 4.2, but the corresponding encoding rate distributions suggest that this is an anomaly of the particular conditions in those simulations. The bottleneck bandwidth and competing traffic are just so that pull multiple applications easily achieve 2000 kbps, but rarely any higher. When more competing flows are introduced stability worsens as the application spends more time using 1200 kbps and 600 kbps chunks. In the following chapter I will demonstrate that improving the efficiency of client pull, by using a longer chunk duration, also results in lower stability since the flows can more easily achieve higher encoding rates.

The distributions also reaffirm that client pull applications are less efficient than the server push and hybrid pull selective applications, which always achieve higher averages, and that they perform even worse when competing against continuous TCP flows. Yellow and red distributions consistently peak further left than blue and green, and yellow and red distributions in Figure 4.3b are all shifted left of their counterpart in Figure 4.3a. Client pull applications make less efficient use of available bandwidth than continuous flows for the reasons discussed in Section 4.2. There is also a smaller, but noticeable, difference between pull multiple and pull single. The latter has marginally better distributions in all but one scenario (2 flows vs TCP). The difference stems from the different ways TCP reacts at chunk boundaries in the two systems, depending on whether a new connection has been made or an old one has been left idle, and may suggest a small efficiency advantage for pull single over pull multiple. Each time a new connection is created TCP resets its congestion window to the initial value. On the other hand, when a connection is left idle, the congestion window will only be reset after a certain period of time, meaning there may be cases in which the window is not reset and TCP can continue sending the new chunk at its previous rate.

Again, the pull selective and server push applications have very similar distributions in each scenario. This supports the theory that changing where ABR calculations are performed alone does not have a significant effect on the dynamics of an ABR flow, so long as a continuous TCP flow is maintained between the client and server.

4.5 Playback Buffer

Figure 4.4, on page 43, shows the behaviour of the playback buffer over time for each simulation, and verifies that adaptation is working. In these graphs, each distribution shows the amount of time spent by an application, over the course of all repetitions for those parameters, in 8 discrete states of playback buffer occupancy. The important thing to notice is that none of the distributions in any of these graphs have a bar in the zero state. This shows that, after startup buffering, buffer occupancy did not fall back to zero in any of these simulations. Since it would not be possible for all ABR flows to stream consistently at the highest possible rate in these scenarios, without consuming content faster than it is being downloaded and eventually stalling while content re-buffers, rate adaptation is working as intended.

The remainder of theses distributions follow roughly a similar pattern. For a playback buffer increasing at a steady rate, we would expect to see roughly equal height bars in each of the 10 second interval states, since as the buffer occupancy grows linearly it will spend roughly the same amount of time in each of these states, followed by a peak in the final state, which covers an open ended interval. The only application that deviates noticeably from this pattern here is the pull selective application. Instead of showing an even spread over the 10 second interval states with a peak in the last state, the distributions for the pull selective applications show a much less even distribution over the 10 second interval states, with a peak around the (10s, 20s] state, and little or no time spent with more than 60 seconds of content buffered. This is one of the only areas where results for server push and pull selective applications are significantly different in my simulations, and suggests that the growth of the playback buffer for pull selective applications was slower and less linear than for the other applications, with more time spent with a smaller buffer occupancy. The rate of growth of playback buffer is linked with the size of the difference between encoding rate and throughput, but it is not clear in particular why there is such a difference between distributions for server push and pull selective applications.

4.6 Discussion

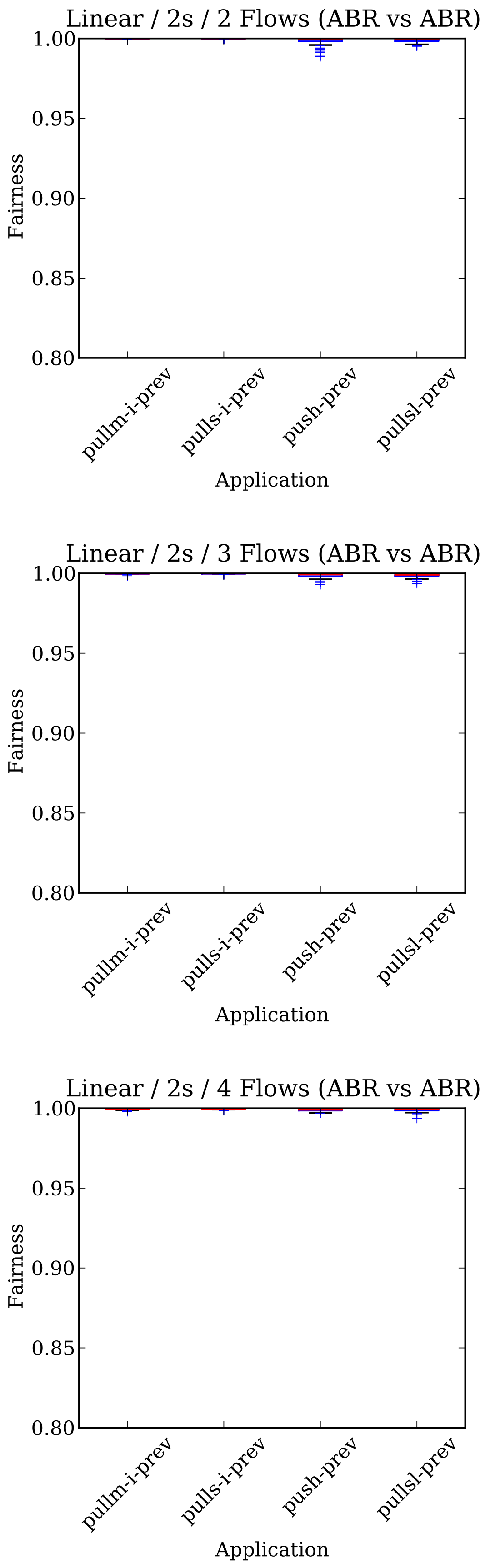
These results highlighted two important issues which later simulations will try to address. The first is that client pull applications are discriminated against when competing against long lived TCP flows, and are less efficient in general compared with server push and pull selective applications. The second is that no application performs very well in any scenario in terms of stability. The results also showed that the hybrid pull selective application performs as well as server push, confirming that where ABR calculations take place has little effect on the stream, as was hoped. Finally, encoding rate distributions suggested a slight efficiency advantage for pull single over pull multiple, due to the difference in how the two applications behave on chunk boundaries, but not enough to make pull single comparable with server push or pull selective.

The discrimination seen by pull multiple and pull single applications competing against long lived TCP flows stems from the on-off nature of client pull flows, and highlights a major difference between pull and push based implementations. Server push and pull selective applications both result in a continuous TCP flow, which has time to settle in the steady state and compete well with other TCP flows, making good use of the available bandwidth. However, pull multiple and pull single applications send a new request to the server after each chunk has downloaded. The resulting idle periods, during which the client is not receiving data, make client pull flows naturally less efficient, and continuous cross traffic flows make it more difficult for TCP to build up its rate again after re-entering the slow start. For ABR flows competing against ABR flows, there is no discrimination between competing flows in Figure 4.1a. However, encoding rate distributions in Figure 4.3a show that the client pull flows are still less efficient than server push and pull selective when two ABR flows compete, with much lower average rates achieved. Server push and pull selective applications settle at encoding rates closer to the fair share bandwidth. The short TCP transfers in client pull flows find it difficult to build up to a high throughput, which leads applications to underestimate their share of available bandwidth and select too low an encoding rate. For example, in each scenario in Figure 4.3a, client pull applications should, in theory, be able to average at least one rate higher without going above the fair share bandwidth and making it impossible to continue indefinitely without re-buffering. Choosing a low rate also means that chunks, and therefore transfers, will be shorter and the problem compounds itself. Similar problems with client pull flows were observed in [3], where throughput falls drastically and client pull flows settle at well below the expected encoding rate after a competing continuous TCP flows is started. In Chapter 5 I will demonstrate that increasing the chunk duration is one effective approach to counteracting this problem for client pull applications.

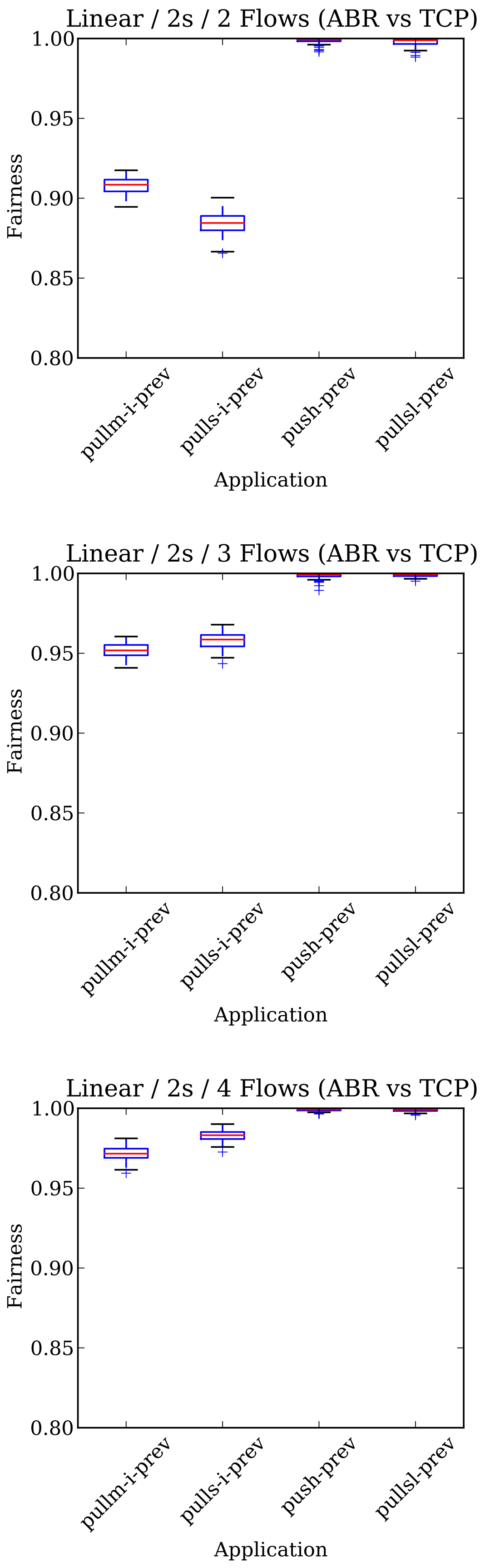
The second point, regarding poor stability, demonstrates a common problem that other studies have found with ABR systems. Instability arises from the difficulty of estimating bandwidth above the HTTP layer, and is evident in every scenario, for all applications. Client pull applications are more stable than server push in most scenarios, but encoding rate distributions show that this is due to limitations imposed on their choice of rate by low throughput. This is confirmed in Chapter 5, when using a longer chunk duration, which improves the efficiency of client pull, also results in a lower stability score for client pull applications. In later chapters, I will show how stability can be improved in all applications, simply by using fewer encoding rates, and also by applying a smoothing function on transfer rates to give more stable bandwidth estimates.

4.7 Summary

The simulations and results outlined in this chapter have highlighted both inherent stability issues with all ABR applications and the efficiency advantage of server push applications over client pull applications. They have also shown that the hybrid pull selective application performs as well as server push. Client pull applications are less efficient and receive lower rates than server push and pull selective, performing even worse when competing against continuous TCP flows. This was shown both in fairness plots using Jain's index and through encoding rate distributions. Stability plots showed that all of these basic implementations suffer from poor stability due to the difficulty of estimating bandwidth above HTTP using only a 2 second sample of the network. All three analyses gave similar results for server push and pull selective in every scenario, and encoding rate distributions showed a slight advantage for pull single over pull multiple. In the next chapter I will extend these simulations to include a longer chunk duration and a new set of encoding rates, in order to backup my findings and show how some of the problems can be alleviated.

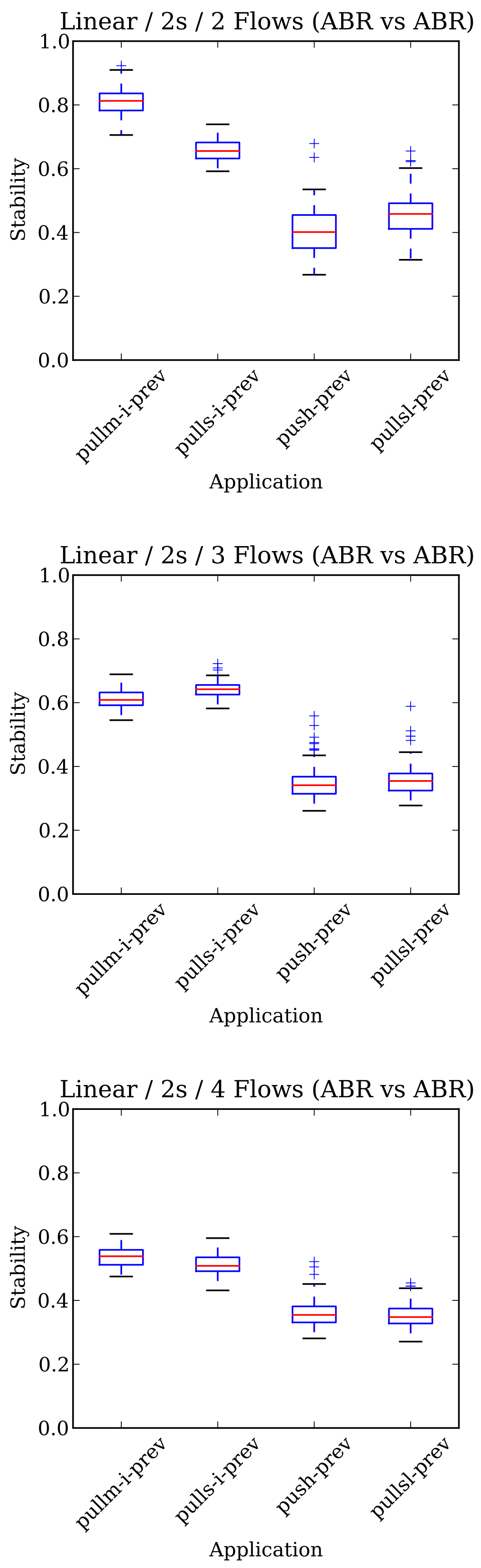


(a) ABR vs ABR

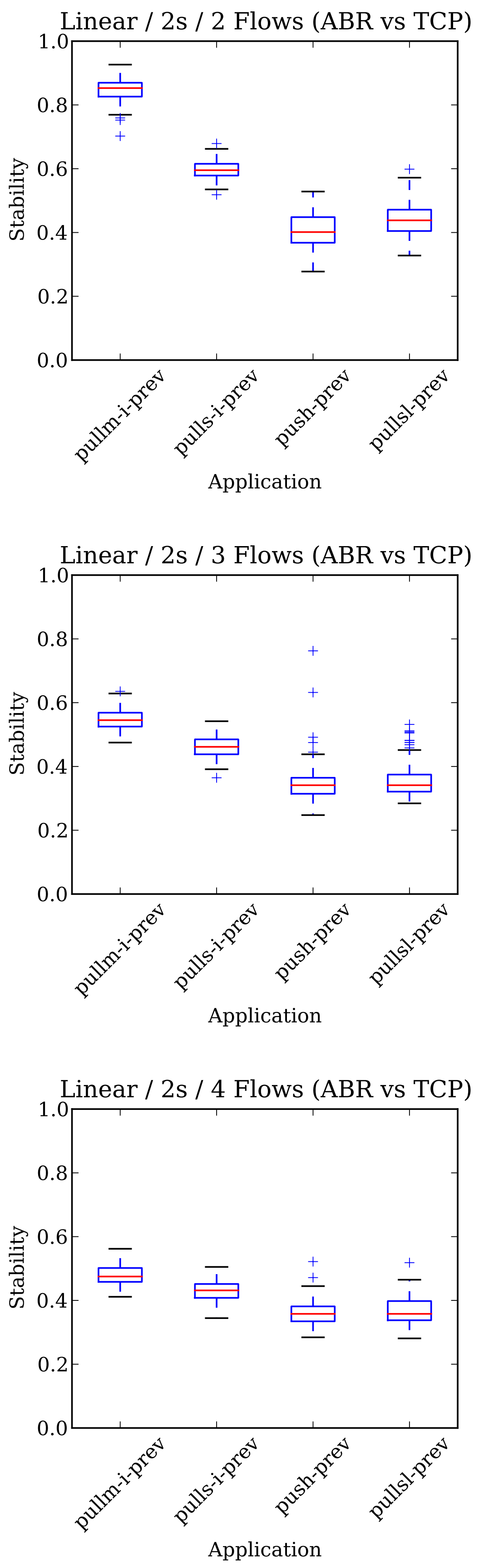


(b) ABR vs TCP

Figure 4.1: Baseline - Fairness

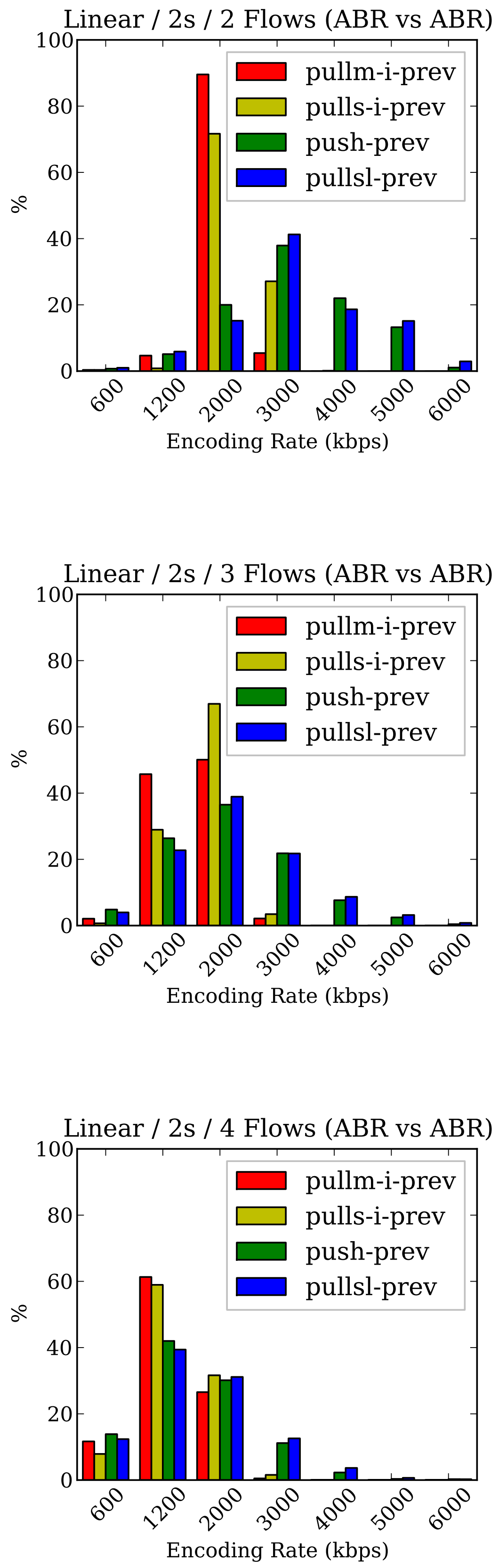


(a) ABR vs ABR

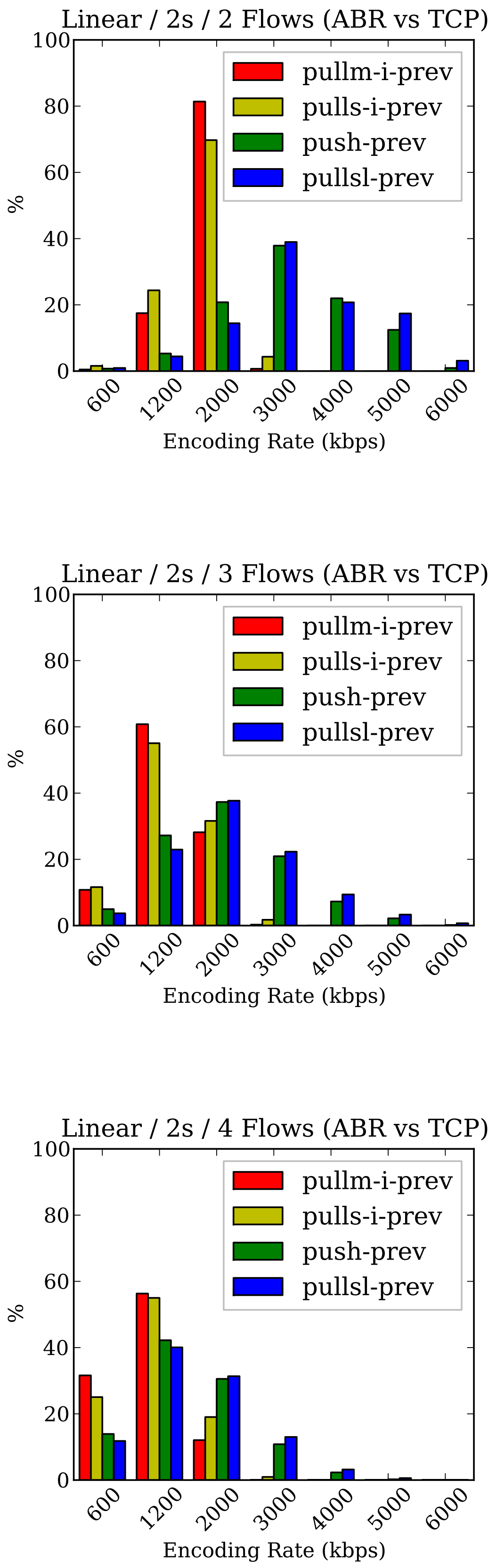


(b) ABR vs TCP

Figure 4.2: Baseline - Stability

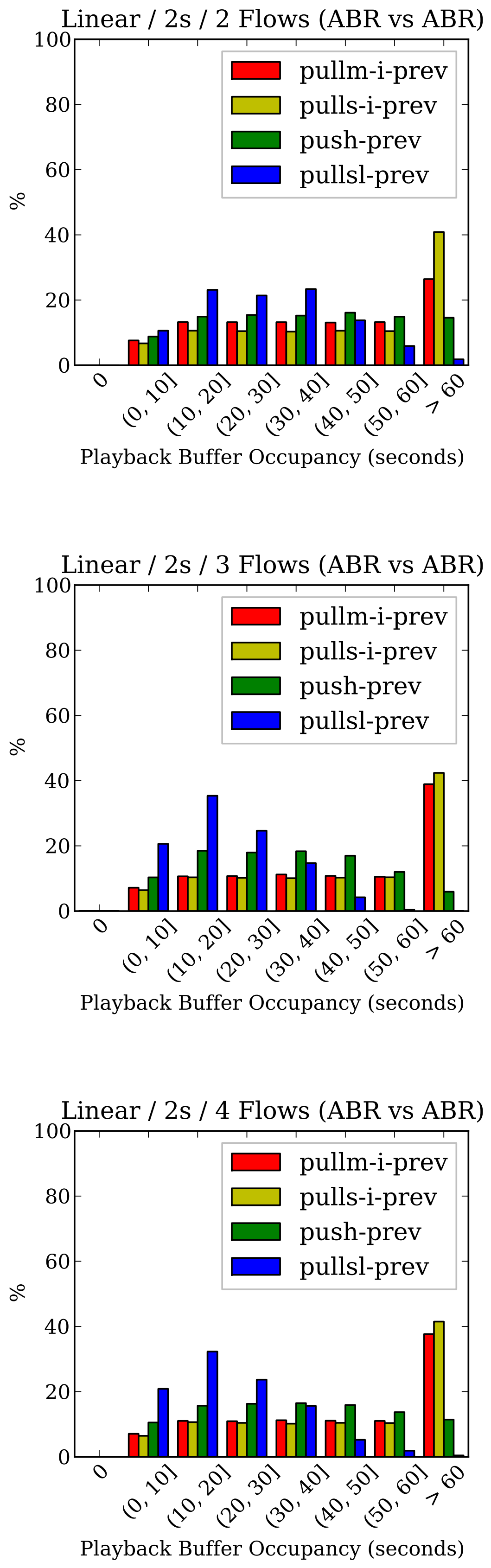


(a) ABR vs ABR

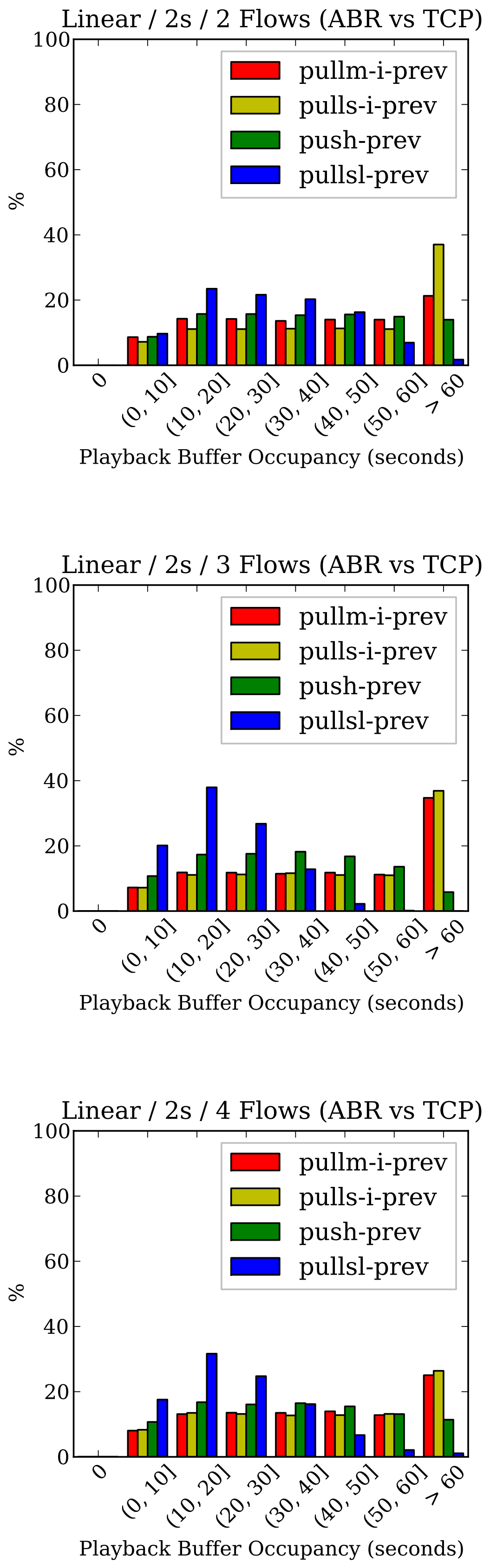


(b) ABR vs TCP

Figure 4.3: Baseline - Encoding Rate Distributions



(a) ABR vs ABR



(b) ABR vs TCP

Figure 4.4: Baseline - Playback Buffer Occupancy Distributions

Chapter 5

Media Formatting Parameters

In Chapter 4 I introduced the first set of simulation results, using the most basic applications possible under each of the four transfer mechanisms, with a 2 second chunk duration and 7 linearly spaced encoding rates. This allowed me to demonstrate two main points, that client pull applications were less efficient than the server push and hybrid applications, especially when competing with continuous TCP flows, and that all of these simplistic approaches performed poorly in terms of encoding rate stability due to the difficulty of picking a suitable encoding rate from bandwidth estimates based on transfer rates seen at the application layer. I also showed that the hybrid pull selective application behaved very similarly to server push, despite maintaining the natural scalability of client pull applications.

In this chapter I extend those simulations by repeating them with different media formatting parameters. I will introduce a 10 second chunk duration and a new set of encoding rates to show how changing these parameters can affect the systems and help to solve some of the problems identified in Chapter 4. In particular I show that a longer chunk duration improves the performance of client pull applications, and using fewer encoding rates can make streams more stable whilst still allowing the system to adapt and avoid playback stalls.

5.1 Details

In the previous chapter I began by comparing four application models representing the most basic implementations under all four transfer mechanism. These were: client pull through multiple connections (**pullm-i-prev**), client pull through a single connection (**pulls-i-prev**), server push (**push-prev**), and client pull with selective requests (**pullsl-prev**). In this chapter I continue with the same four application models and evaluate them using different media parameters. In addition to the **2 second** chunk duration and linear encoding rates (**600, 1200, 2000, 3000, 4000, 5000, 6000 kbps**) used previously, I will now introduce a **10 second** chunk duration and a smaller set of exponentially spaced encoding rates (**600, 1200, 3000, 6000 kbps**). In Chapter 4 I argued that the performance hit on client pull came largely from the effects of frequent idle periods on TCP flows. I will now bring evidence to support this conclusion by showing that client pull applications perform much better when using a 10 second chunk duration. This is expected, since as the chunk duration increases the individual transfers tend more towards behaving like long lived continuous TCP flows, but increased chunk duration also has drawbacks and using anything much greater than 10 seconds would risk not allowing the application to react in time to congestion. In ideal circumstances, I would like to be able to simulate more chunk durations between 2 and 10 seconds, to find out how increasing the chunk duration affects the systems behaviour with more granularity, but time constraints on running simulations prevent me from doing so here. I will also demonstrate that using a smaller set of encoding rates makes ABR streams more stable, mainly because there are fewer rates available and the system will be less likely to switch rate due to short term fluctuations in throughput.

Applications using 2 second chunk durations will send 600 seconds worth of content over 300 chunks, those using 10 second chunks send 1000 seconds worth of content over 100 chunks. The reason for the extra content duration with 10 second chunks is to help ensure that there are enough chunks in the stream for clear stability patterns to emerge. For example, there will be rate switches at the start of most streams, as the system increases its rate initially, and these shouldn't necessarily be taken as a sign of instability. However, with fewer chunks in the stream, these initial rate switches will skew the stability index. Each ABR application from the same simulation will use the same media parameters, and continuous TCP flows run while any ABR applications are still running. The simulations from Chapter 4 are repeated using each of the three new possible combinations of media parameters. Each application is tested against both itself and continuous TCP flows with 2, 3, and 4 competing flows, giving 72 new simulations repeated 100 times. This gives 9,600 simulations covered in total, including the results repeated from the previous Chapter. The results are presented this time to highlight differences between those in the previous chapter, with old results in the first column of each figure and new ones in the second.

5.2 Fairness

Figures 5.1, 5.2, and 5.3, starting on page 53, show fairness plots for the four basic applications with 2, 3, and 4 flows competing. Each figure is split into four subfigures, showing applications tested under each of the four combinations of media parameters, going across the page. The top graph in each subfigure shows ABR flows competing against ABR flows while the bottom shows ABR flows competing with long lived continuous TCP flows. This presentation format is designed to make it easy to compare like for like simulations across the rows, where only the media parameters change. Fairness is calculated using Jain's index, given in Equation 3.1 from Chapter 3, and each individual box plot shows the spread of values calculated for 100 repetitions of a single simulation.

In Chapter 4 I showed, using linear encoding rates and a 2 second chunk duration, that client pull applications perform poorly in some scenarios compared to server push and pull selective in terms of both fairness and overall bandwidth efficiency. Fairness plots in 4.2 clearly showed discrimination against the ABR flow when client pull applications competed against continuous long lived TCP flows. I argued that the reason for this is that frequent idle periods at chunk boundaries cause TCP's congestion window to reset, putting the flow back into the slow start phase, and making it very difficult for it to compete with a flow that sends continuously at a high rate. Towards the end of that chapter I hinted that using a longer chunk duration can improve client pull's performance in this regard.

Figure 5.1a recaps the fairness results from Chapter 4. In the top graph ABR flows compete against identical instances of themselves, with no noticeable discrimination, however, when competing against long lived TCP flows in the bottom graph, both client pull applications are discriminated against. Figures 5.1b, 5.1c, and 5.1d extend these results to show applications tested in the same situations using different media parameters. The bottom graphs of Figures 5.1b and 5.1d highlight the benefit of using a longer chunk duration for client pull applications competing with long lived TCP flows. In these graphs, both showing applications using a 10 second chunk duration, the Jain's index is much closer to 1.0 for pull multiple and pull single applications. In the bottom graphs of figure 5.1c, using a 2 second chunk duration, this time with the exponential encoding rates, the fairness index is still relatively low. This pattern is repeated in Figures 5.2, and 5.3. Client pull applications have a low Jain's index when competing against continuous TCP flows using a 2 second chunk duration, and a much higher value when a 10 second chunk duration is used in the same scenario.

The reason behind these observations follows on from the explanation of why client pull applications are less efficient in the first place. In Section 4.2 I described how idle periods between chunk boundaries affect client pull flows, making them less efficient than server push and pull selective, which both maintain a continuous TCP flow. But when the chunk duration is increased, idle periods become fewer and less significant. They are fewer simply because a longer chunk duration means fewer chunks for the same duration of content, and less significant, because the amount of time spent idle or back in slow start after each chunk does not change, while the overall duration of each transfer increases. As the chunk duration increases, the ABR flow tends more towards behaving like a regular long lived continuous TCP flow, and transfers have more time to build up and maintain a high rate between the idle periods. This effect will also be witnessed in the encoding rate distributions in Section 5.4. The distributions show that, not only do client pull flows compete better against continuous TCP flows when using a longer chunk duration, but they also achieve better rates on average in any given scenario than the same application using a 2 second chunk duration.

5.3 Stability

Figures 5.4, 5.5, and 5.6, starting on page 56, show stability plots in a similar format to the fairness plots from Section 5.2. Media parameters vary from left to right across the page, with ABR vs ABR on top and ABR vs TCP beneath. Stability is calculated using Equation 3.2 from Chapter 3, and each individual box plot shows the spread of values calculated for 100 repetitions of a single simulation.

The stability plots in this chapter demonstrate two main things. The first, is confirmation of an explanation given in Chapter 4 as to why client pull applications are significantly more stable than server push and pull selective in some situations. I suggested that this was simply because poor efficiency was limiting the choice of rates for those applications. I have already demonstrated, in Section 5.2, that increasing chunk the duration has a strong positive effect on the performance of client pull applications. Now notice how, in all three figures, stability across the four applications is always more evenly matched in the second and fourth subfigures than in the first and third. In other words, whenever a 10 second chunk duration is used, client pull applications are much more efficient with bandwidth and therefore able to achieve higher encoding rates, more similar to server push and pull selective. The downside is that this means they also have more opportunity to switch rate and naturally become less stable. This is not necessarily a disadvantage for client pull, since they are not significantly less stable than server push or pull selective, but it does rule out increased stability as an advantage for client pull systems. One could easily draw such a false conclusion from results in Chapter 4. Therefore, the observation from Section 4.3 holds, that all applications have poor stability, and none on their own seem to have any great advantage over the others.

The second thing that can be seen in these graphs is a slight trend towards increased stability when the smaller set of exponentially spaced encoding rates are used. In each figure, application for application, stability is always at least slightly higher, and sometimes significantly higher, in graphs from subfigure c than in those subfigure a, and likewise in graphs from subfigure d than those from subfigure b. This is comparing like for like with only the available encoding rates changing. The reason for this seems to concern the number of rates available. When fewer rates are available there are simply fewer opportunities to switch rate. Intuitively, one might think that more rates would make for a more granular system, able to react better to congestion. More granularity means the system is more likely to be able to use the highest possible encoding rate for the given bandwidth constraints, but the drawback is that more rates will also result in more rate switches. The developer's job here is to find an appropriate balance. Having more encoding rates available also means more space is taken up on the server. However, there is a caveat with this result, and further work is required to confirm whether the effect is due solely to the decreased number of rates, or to the different spacing between rates, or some combination of both. Larger spacing between the higher rates could also account for some of the extra stability, and re-running these simulations with four linearly spaced rates would help to clarify the result.

5.4 Encoding Rate Distributions

The layout of graphs in Figures 5.7, 5.8, and 5.9, starting on page 59, follow the same format as graphs from the previous two sections, this time with four distributions per graph, one for each application showing the distribution of encoding rates chosen over 100 repetitions of that simulation. Two observations from these distributions support claims made regarding both stability and fairness in Sections 5.2 and 5.3.

The first observation is that, in all three figures, the pull multiple and pull single applications have much better distributions in subfigures b and d, where a 10 second chunk duration is used, than those in subfigures a and c where the original 2 second chunk duration is still used. When a 10 second chunk duration is used the client pull applications achieve higher average rates, with distributions that are much more comparable to those for server push and pull selective. They are often still not quite as efficient, but the effect is definitely noticeable. As well as achieving higher average rates, the client pull applications also show more variation in encoding rates when using a 10 second chunk duration, which relates back to the point regarding stability of client pull applications. Their lack relatively high stability with a 2 second chunk duration is a symptom of poor efficiency rather than a natural advantage for the client pull approach. Again, increasing the chunk duration makes the client pull applications more efficient by reducing the number of idle periods and allowing them more time to send at a higher rate in between.

The second observation is that the distributions for all applications always show less variation in subfigures c and d, where the smaller set of exponential encoding rates are used, than in a and b where the larger set of linear encoding rates are used. This is simply a direct consequence of having fewer rates to choose between. For all applications, in all six scenarios, the most favourable distributions come in subfigure d, where the exponential encoding rates are used in conjunction with a 10 second chunk duration. Here we see the highest average rates along with the lowest variation, for reasons described above.

5.5 Playback Buffer

Although I showed in Section 5.2 that increasing the chunk duration improves the efficiency of client pull flows, there is also a danger that in creasing the chunk duration can damage the system's ability to react quickly to changes on the network and avoid playback interruptions. Playback buffer occupancy distributions in Figure 5.10, on page 62, show that using a 10 second chunk duration still allows the system to successfully adapt to prevent stalls. The top row shows ABR flows competing with ABR flows and the bottom shows ABR flows competing with continuous TCP flows, while each column uses a different set of media parameters. Figures 5.10c and 5.10d both use 10 second chunk durations and neither show any applications spending time with an empty playback buffer. There are other noticeable differences between distributions for each of the four sets of media parameters, but this is less important here and reasons behind them are not clear. Playback buffer occupancy distributions for simulations involving 3 and 4 competing flows follow similar patterns, with no stalls, but are not shown here for brevity.

5.6 Discussion

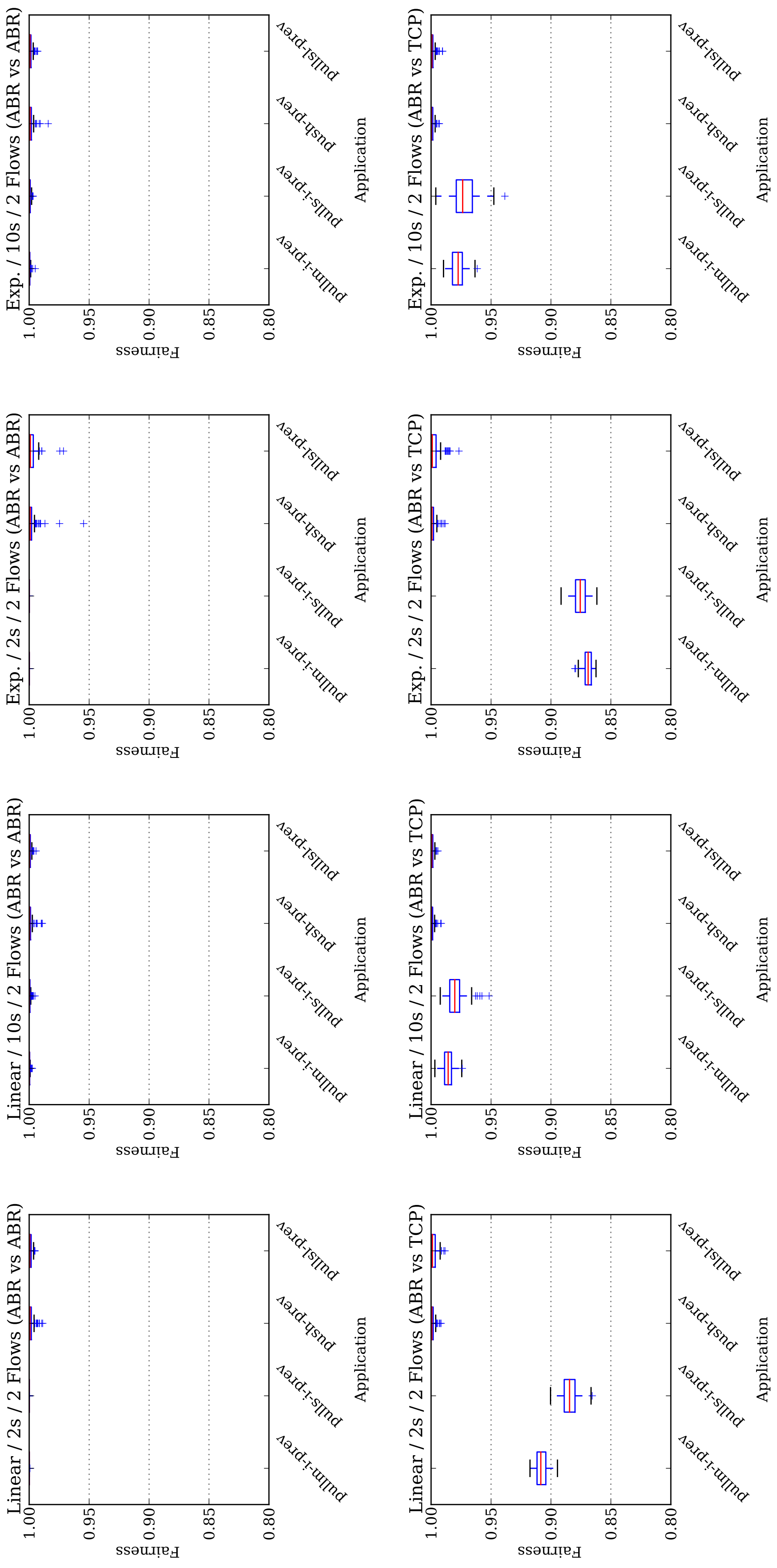
Results in this chapter have highlighted two important tradeoffs to consider when deciding how to format media content for ABR streaming. I have shown that, for a client pull based system, increasing the chunk duration from 2 seconds to 10 seconds can significantly improve the efficiency of client pull flows. With a longer chunk duration, idle periods are less frequent, and transfers for individual chunks achieve and maintain a higher throughput as they start to behave more like long lived transfers. This results in better bandwidth estimates, meaning that a system using rate based adaptation will be able to reach higher encoding rates and stream at a rate that is closer to their fair share of the available bandwidth. Similar results have come from other recent studies, suggesting that a longer chunk duration is beneficial for client pull systems. While investigating the efficiency problems with client pull flows, [3] finds a correlation between the size of a chunk and the throughput achieved when sending it across the network. The authors do this by demonstrating from their experiments both that the size of a chunk is proportional to the encoding rate, and chunks with higher encoding rates achieve better throughputs on average. They also suggest that using a longer chunk duration will improve the performance of client pull systems. The tradeoff with chunk duration, however, is that using too long a chunk duration will impact the systems ability to react quickly to congestion and avoid playback interruptions. In Section 5.5 I showed that using a 10 second chunk duration does not prevent adaptation from achieving its goals in my simulations. Further work in this area could investigate the effects of varying the chunk duration in more detail, both by simulating more intermediate durations between 2 and 10 seconds, and by simulating durations longer than 10 seconds to find out at what point increasing the chunk duration becomes detrimental to the system's ability to react to congestion. This is also mentioned later in Section 8.3, when I discuss scope for future work.

The second tradeoff highlighted by these results concerns the encoding rates that are available for use. Stability plots in Section 5.3 showed a trend towards increased stability whenever the smaller set of four exponentially spaced rates were used rather than the set of seven linearly spaced rates. Encoding rate distributions in Section 5.4 also showed much less variation when the exponential rates were used. It is not clear what difference the spacing makes, and whether the effect on stability is due to mainly to the different number of rates or the different spacing between rates. Further work is needed to clarify this point, by simulating four linearly spaced rates, but having fewer rates available seems to improve stability by reducing the number of opportunities to switch rate. However, having fewer available rates increases the possibility that the system is in a position where the current encoding rate is considerably lower than the available bandwidth, simply because a higher rate that is still lower than the bandwidth is not available. In this case the user experiences lower quality video playback than what the bandwidth conditions will allow. The values of suitable encoding rates will always depend on a number of circumstantial factors, such as the intended content and target device, but developers considering what rates to make available should be aware of these tradeoffs. Although having more options will maximise the chances of streaming at the highest possible rate, not only does that mean more space is required to host the content on the server, but more options will also lead to more fluctuation and a less stable stream, unless further steps are taken to ensure stability.

5.7 Summary

In this chapter I investigated the effects of changing media formatting parameters, and found two important tradeoffs in this area. I showed that increasing the chunk duration from 2 to 10 seconds can improve the performance of client pull applications, and that having fewer encoding rates available naturally makes for a more stable stream. However, too long a chunk duration risks not allowing the system to react properly to congestion, and too few encoding rates makes it difficult to stream at the highest possible rate in any given situation. If taken too far, both undermine the purposes of ABR streaming.

In the next chapter I begin looking into more sophisticated application models, starting with bandwidth estimation techniques. I will extend the four application models used so far to incorporate bandwidth estimators that use averaging functions to estimate bandwidth from transfer rates, and show how this approach can significantly improve stability in all applications.



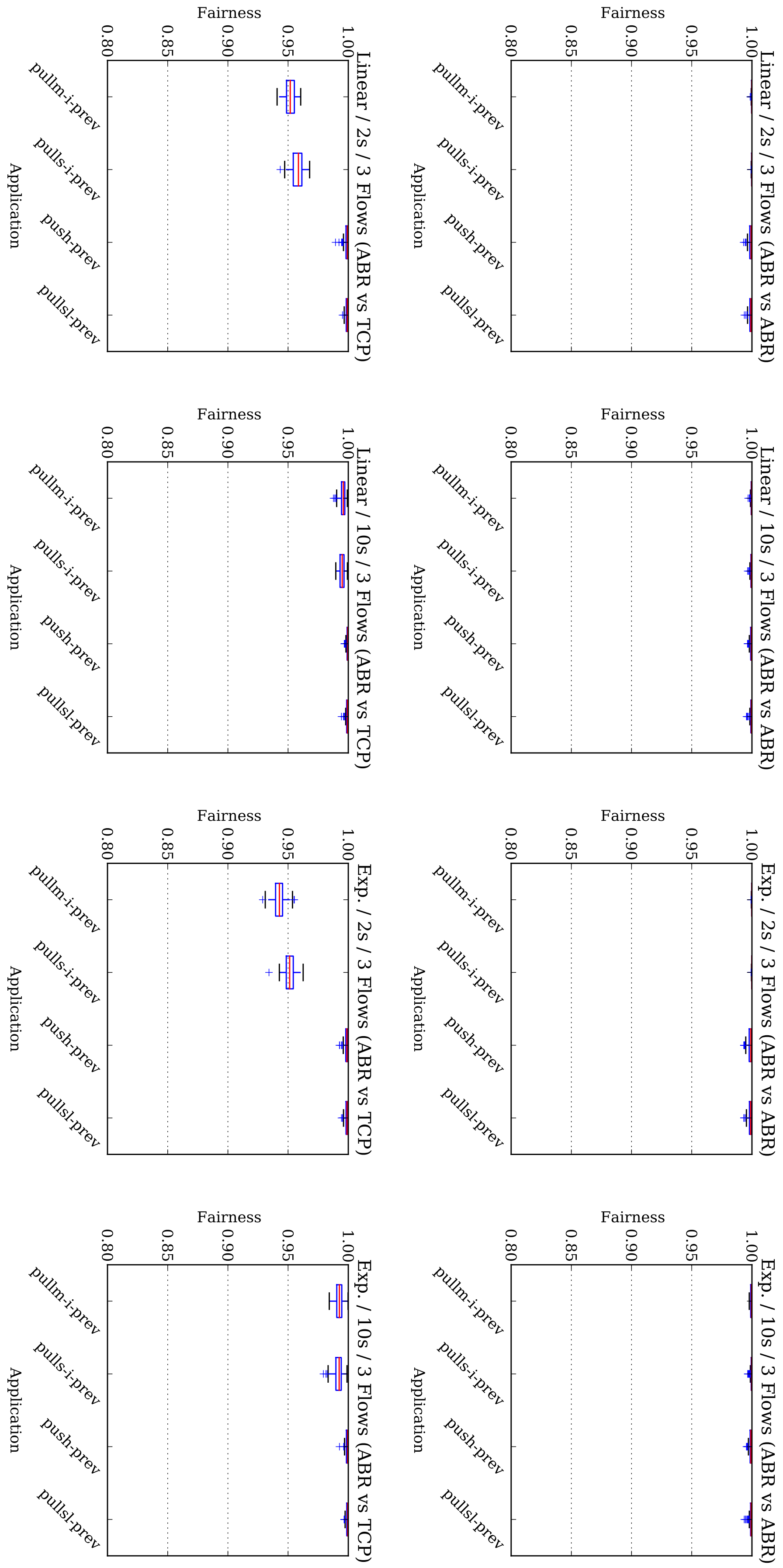
(a) Linear / 2s

(b) Linear / 10s

(c) Exp / 2s

(d) Exp / 10s

Figure 5.1: Media Parameters - Fairness (2 flows)



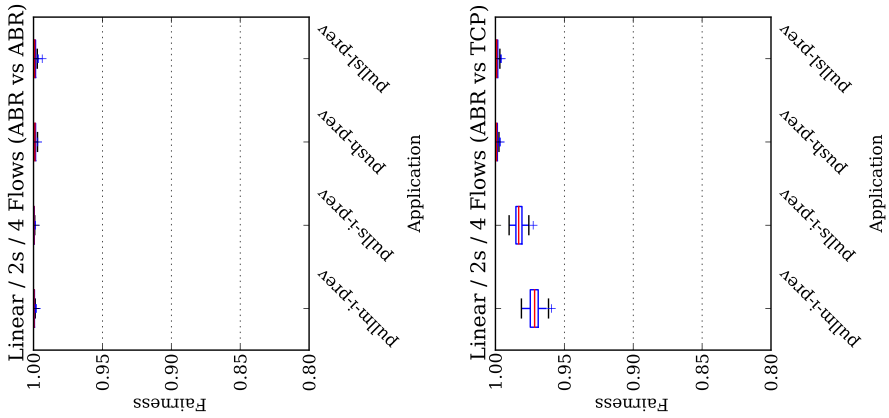
(a) Linear / 2s

(b) Linear / 10s

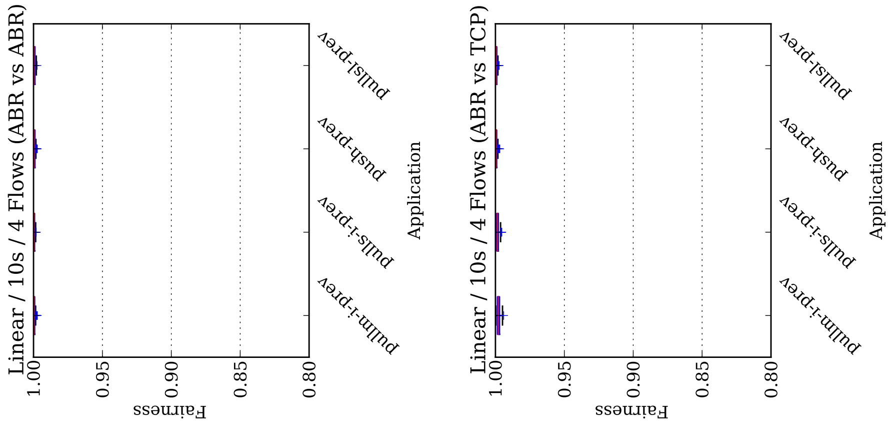
(c) Exp / 2s

(d) Exp / 10s

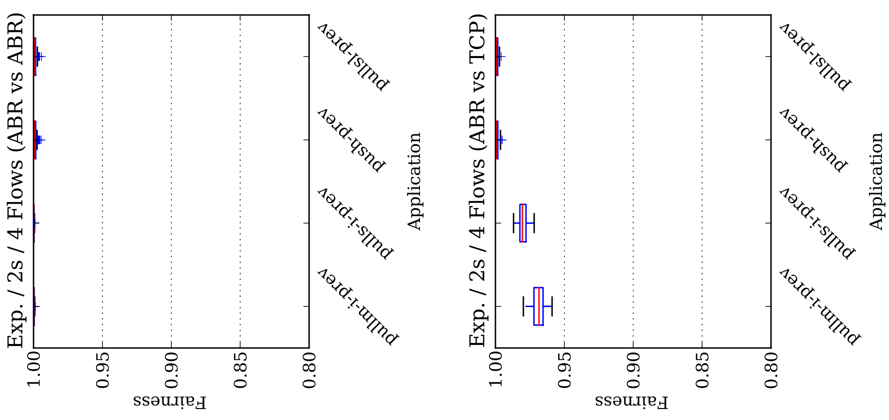
Figure 5.2: Media Parameters - Fairness (3 flows)



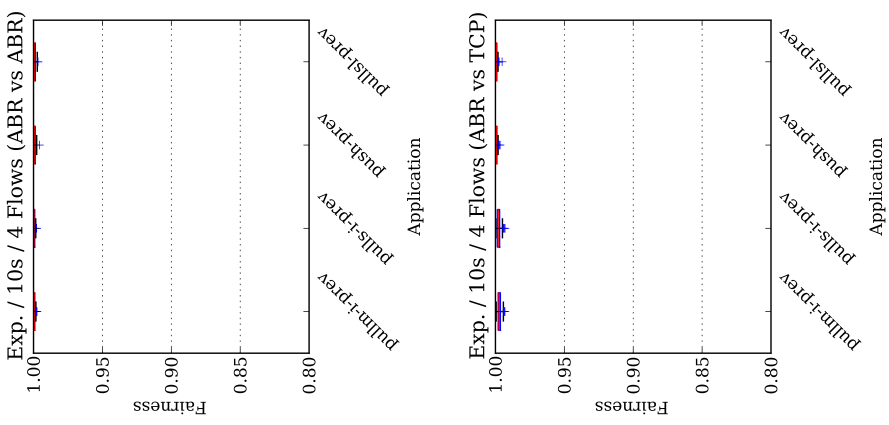
(a) Linear / 2s



(b) Linear / 10s

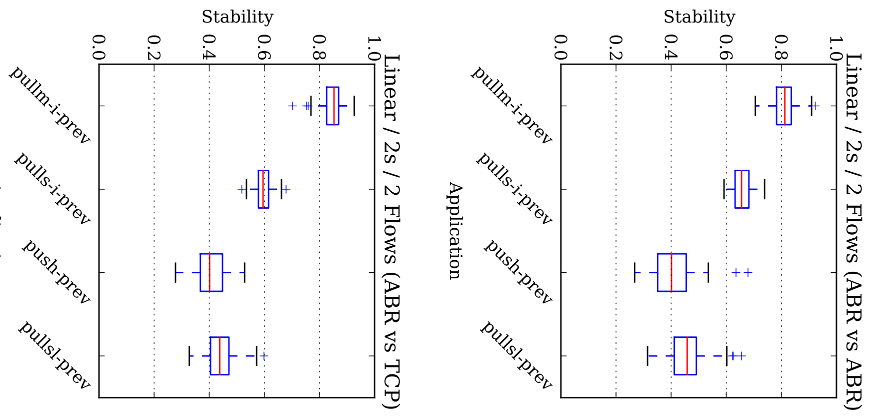


(c) Exp / 2s

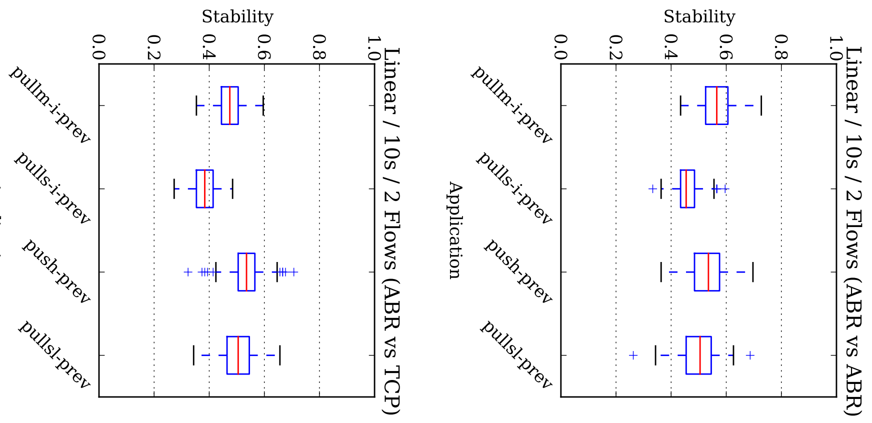


(d) Exp / 10s

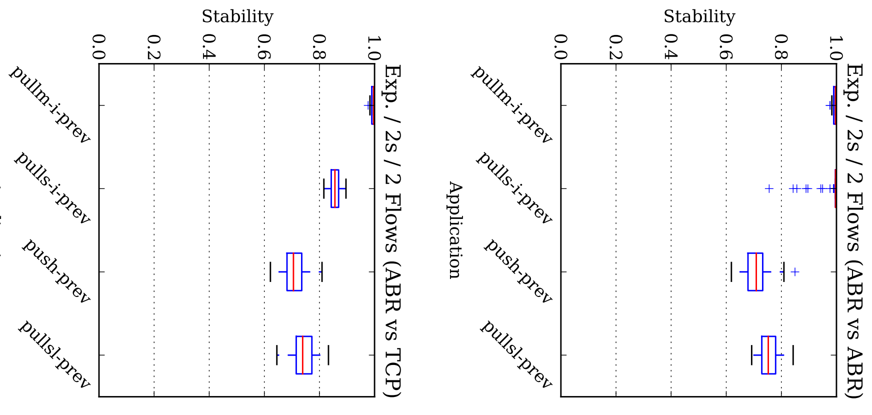
Figure 5.3: Media Parameters - Fairness (4 flows)



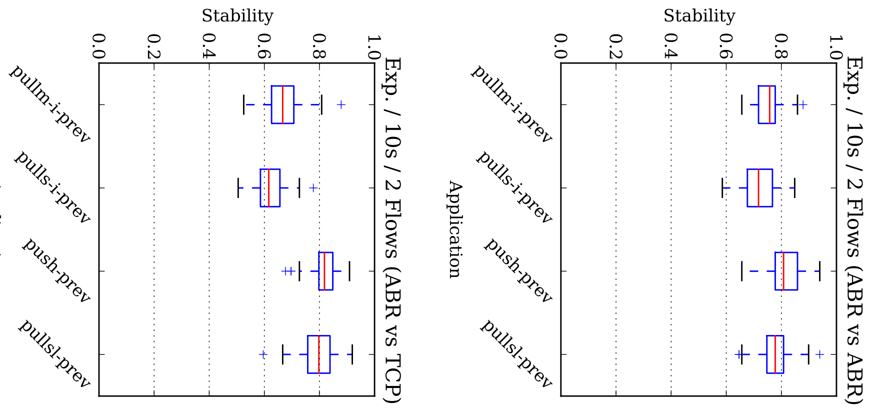
(a) Linear / 2s



(b) Linear / 10s



(c) Exp / 2s



(d) Exp / 10s

Figure 5.4: Media Parameters - Stability (2 flows)

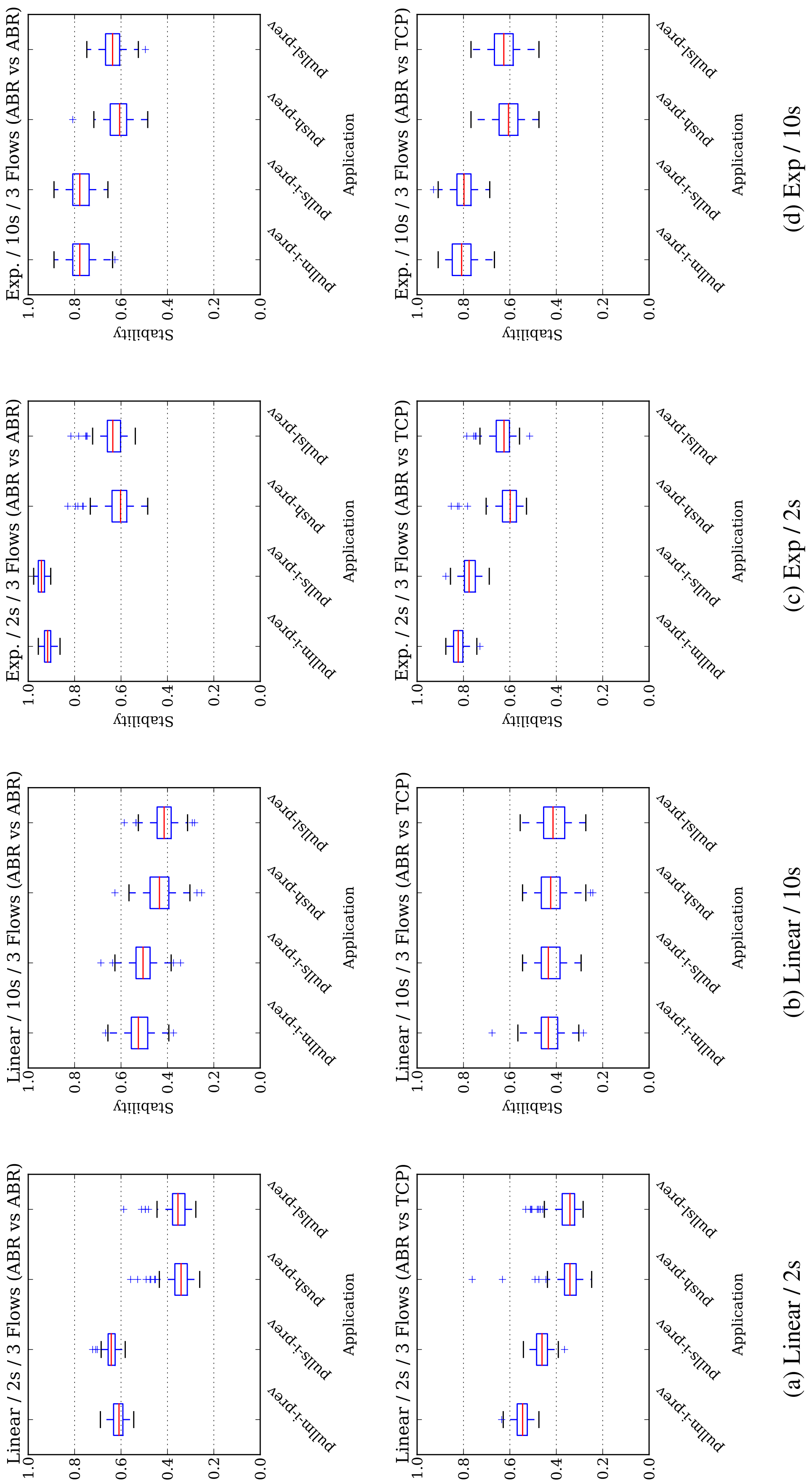
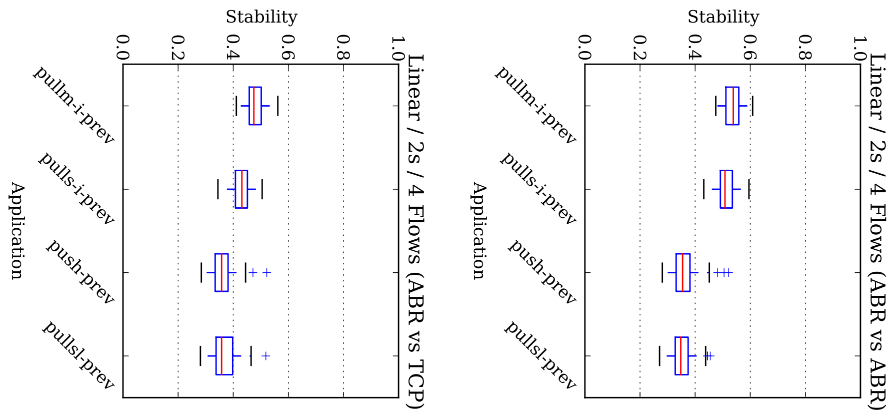
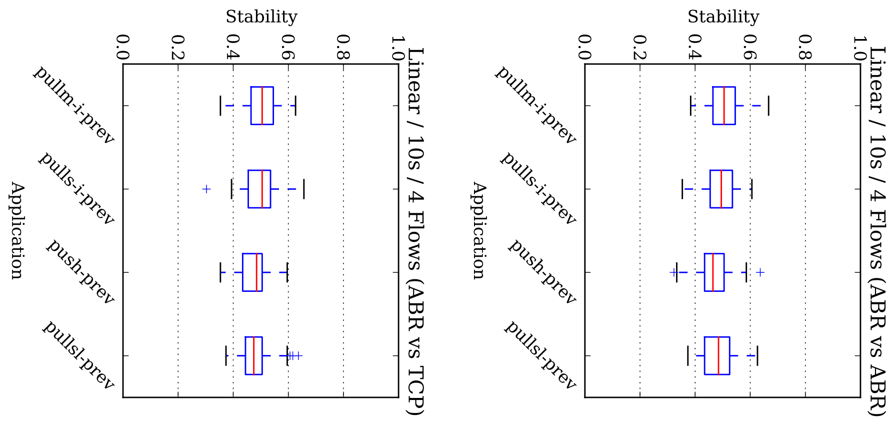


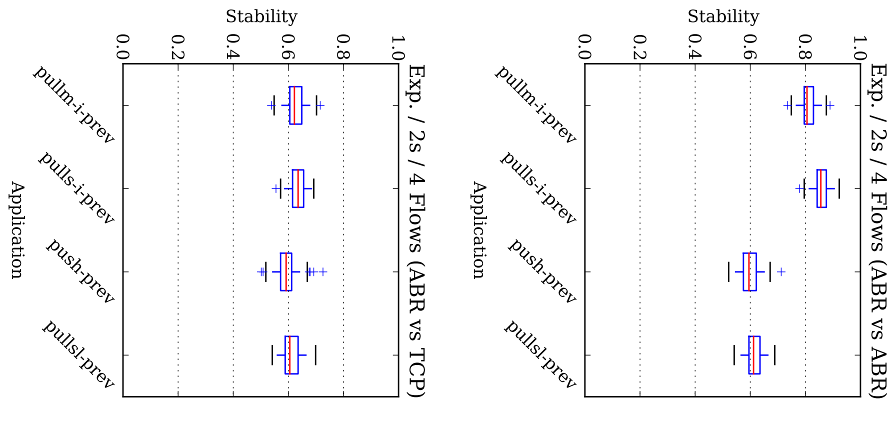
Figure 5.5: Media Parameters - Stability (3 flows)



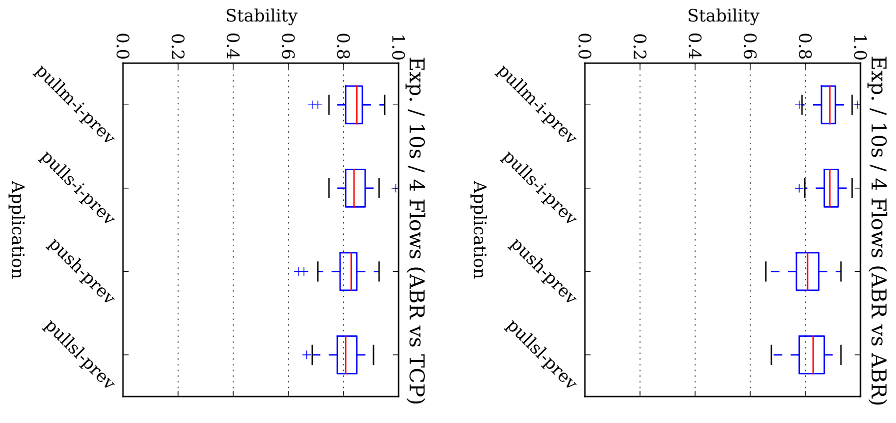
(a) Linear / 2s



(b) Linear / 10s

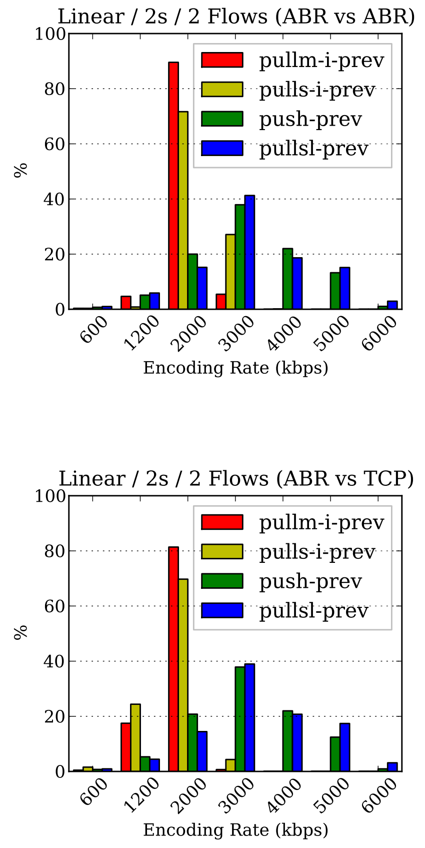


(c) Exp / 2s

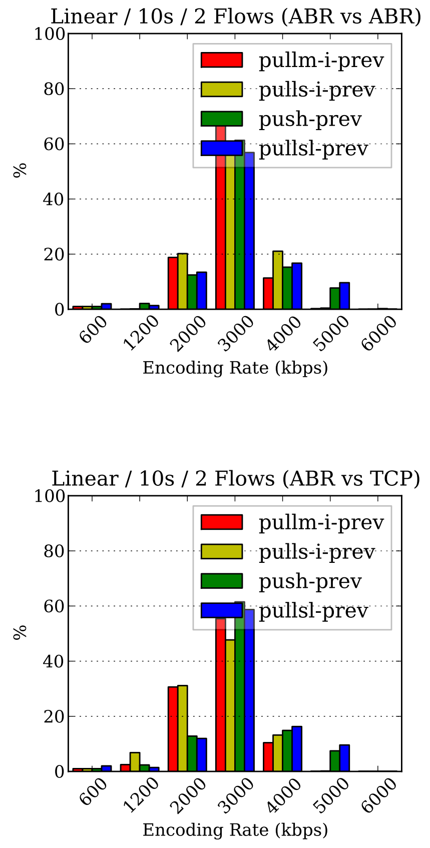


(d) Exp / 10s

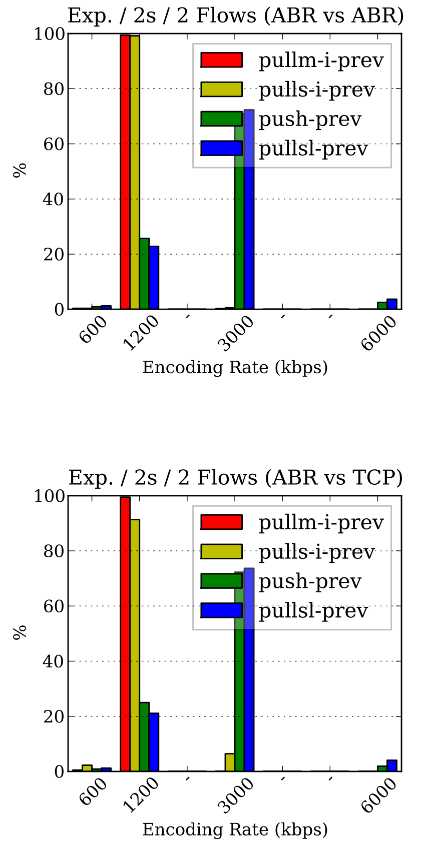
Figure 5.6: Media Parameters - Stability (4 flows)



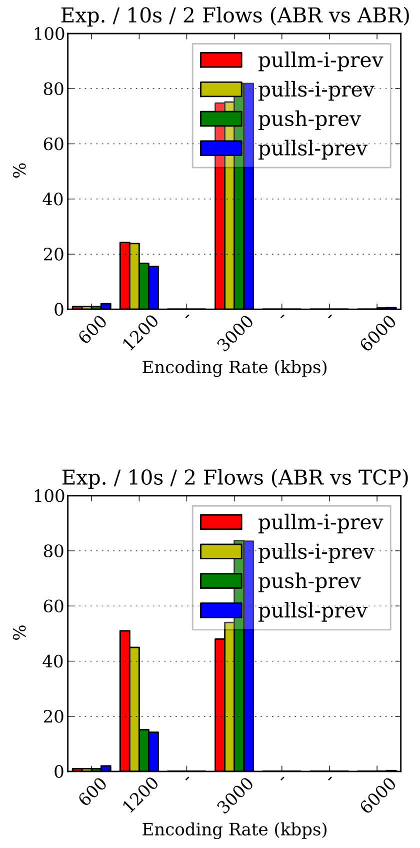
(a) Linear / 2s



(b) Linear / 10s

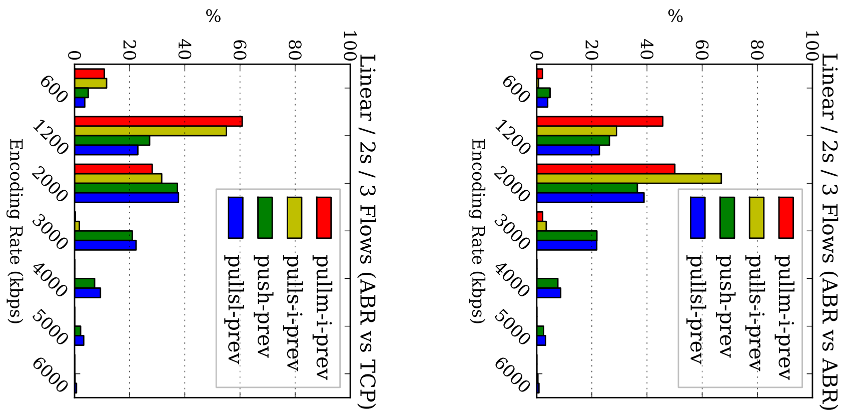


(c) Exp / 2s

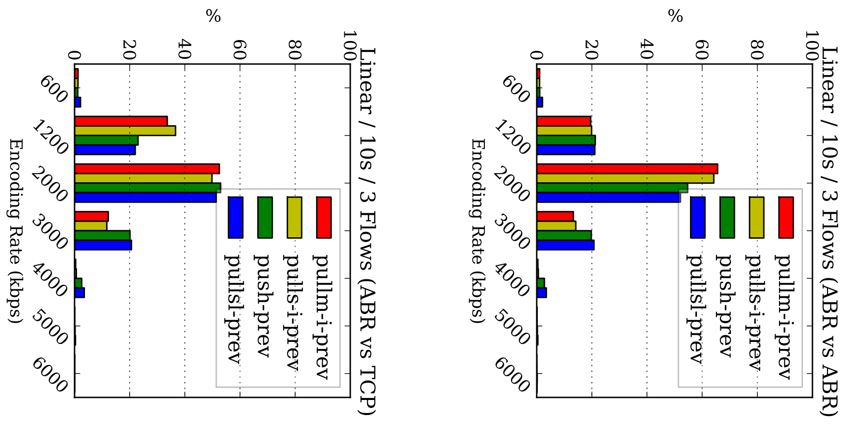


(d) Exp / 10s

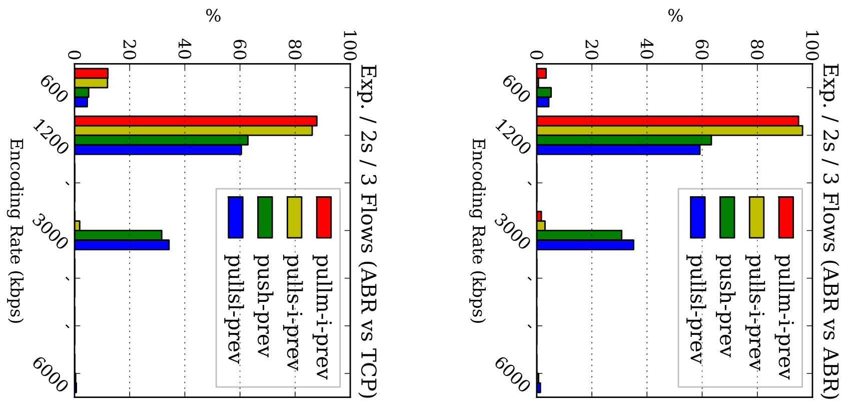
Figure 5.7: Media Parameters - Encoding Rate Distributions (2 flows)



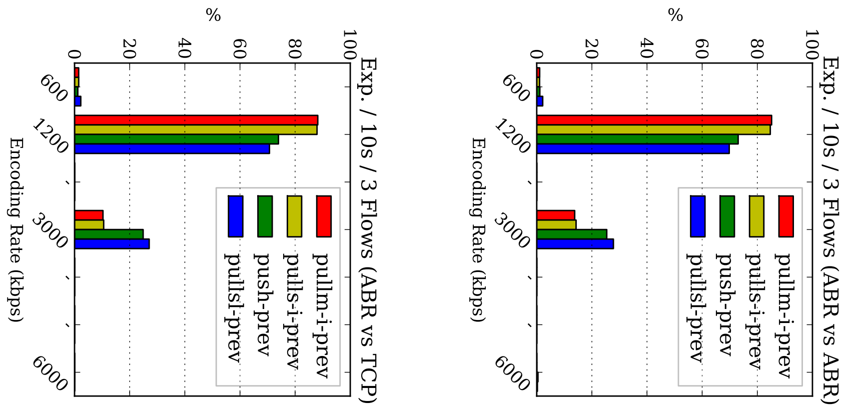
(a) Linear / 2s



(b) Linear / 10s

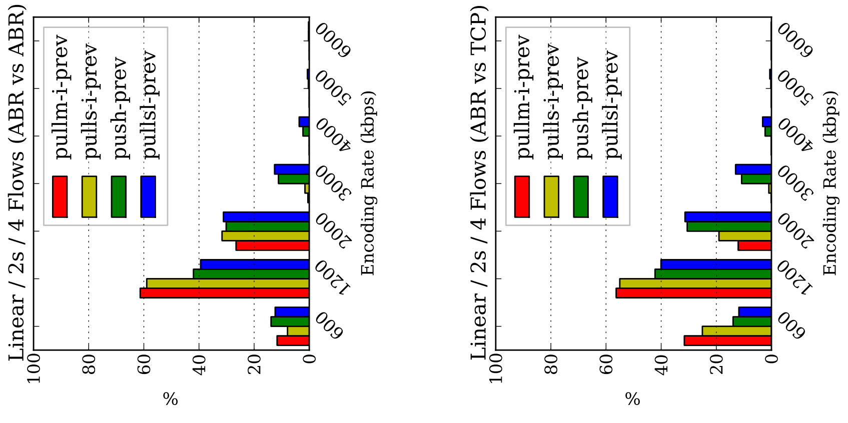


(c) Exp / 2s

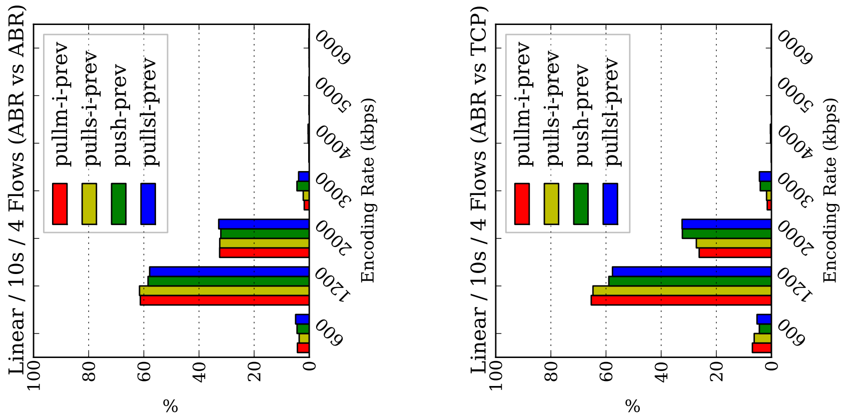


(d) Exp / 10s

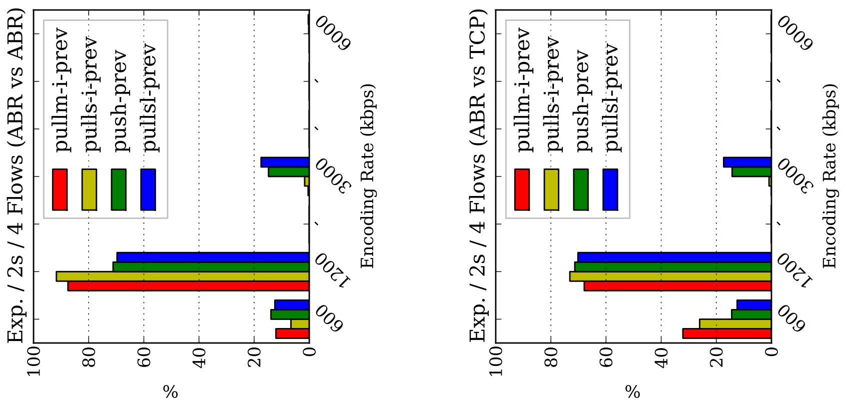
Figure 5.8: Media Parameters - Encoding Rate Distributions (3 flows)



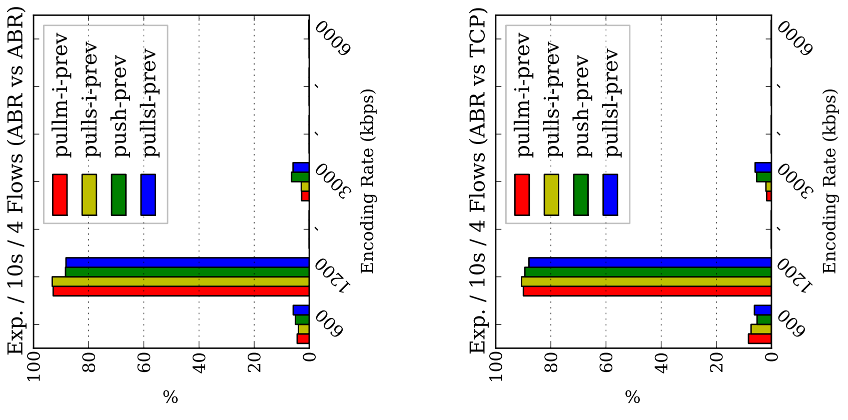
(a) Linear / 2s



(b) Linear / 10s

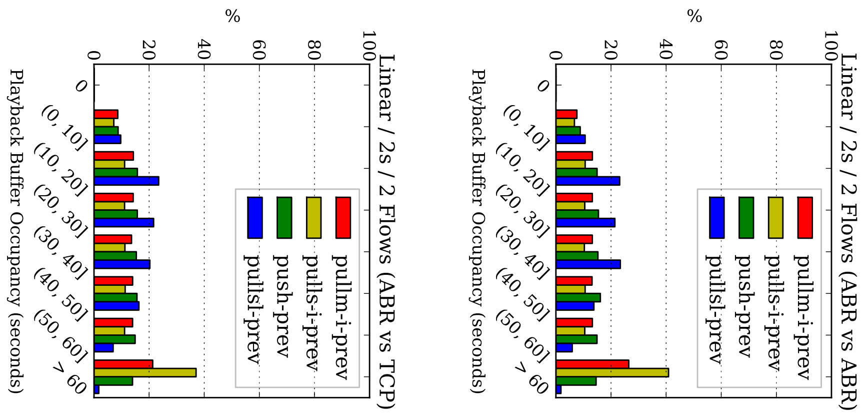


(c) Exp / 2s

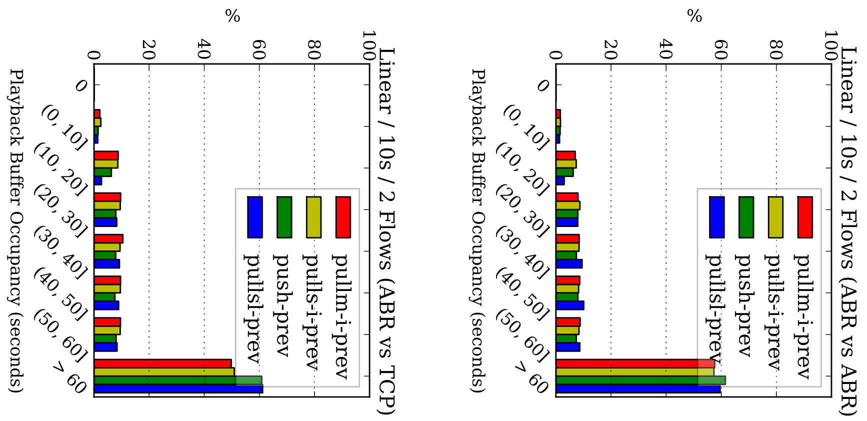


(d) Exp / 10s

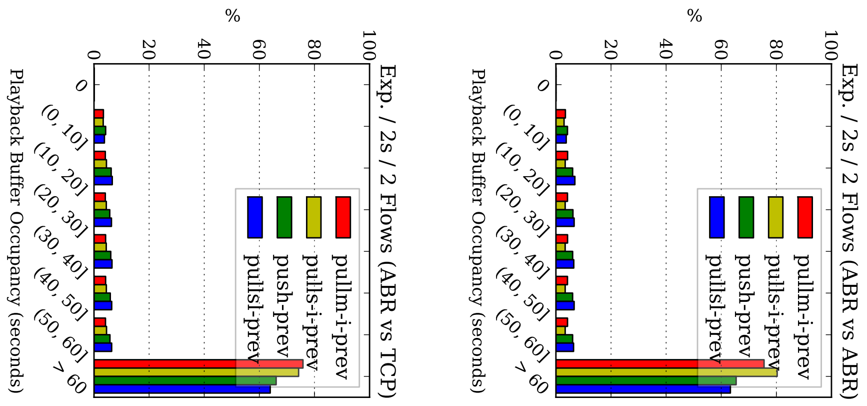
Figure 5.9: Media Parameters - Encoding Rate Distributions (4 flows)



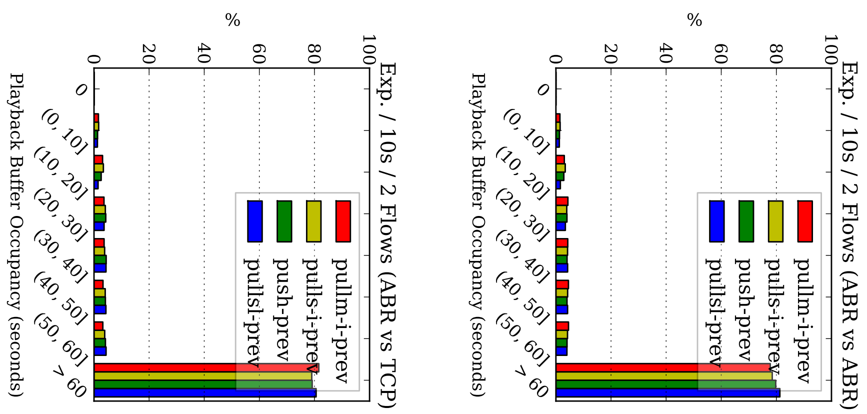
(a) Linear / 2s



(b) Linear / 10s



(c) Exp / 2s



(d) Exp / 10s

Figure 5.10: Media Parameters - Playback Buffer Occupancy Distributions (2 flows)

Chapter 6

Bandwidth Estimation

So far I have demonstrated some of the inherent problems many ABR systems face, and shown how tweaking media formatting parameters can help to tackle these issues. This has been done by considering only the most basic application implementations possible under each of the four transfer mechanisms described in Section 3.3. In this chapter I extend these applications with more sophisticated bandwidth estimation techniques, in order to address the poor encoding rate stability exhibited by all applications in previous chapters. Until now my applications have used the transfer rate of the previous chunk as a bandwidth estimate. Here I will introduce two new approaches, using different averaging functions to smooth a series of recently observed transfer rates, and show that both improve stability across all applications. The averaging functions I use are the exponentially weighted moving average and the harmonic mean.

6.1 Details

Simulations in Chapters 4 and 5 have shown that all basic application models exhibit poor encoding rate stability. This stems from the difficulty of estimating bandwidth above the HTTP layer, where observed rates do not necessarily represent the true network state, and are likely to fluctuate over time when applications compete with other flows. Given that consecutive transfer rates fluctuate significantly, using the transfer rate of the previous chunk as a bandwidth estimate is bound to result in encoding rate instability. A simple idea to tackle this problem is to use an averaging function on a series of recent rate estimates. This should mean that short term fluctuations are filtered out to give a more stable set of rate estimates over time, which in turn will result in less variation in encoding rates. The tradeoff, however, is that abrupt changes in the available bandwidth (as opposed to fluctuations in the observed rate estimates) will now take longer to be noticed, which may hamper the system's ability to react and prevent stalls.

The averaging functions I investigate are the exponentially weighted moving average of all previously observed rates (**EWMA**), and the harmonic mean of the previous 20 chunks' rate estimates (**HM20**). The exponentially weighted moving average is a rolling average that is updated each time a new chunk is downloaded, using the formula shown in Equation 6.1. This is trivial to implement on top of previous chunk bandwidth estimation, and only requires the system to remember what the last value calculated was. The harmonic mean is only slightly more complicated, requiring the previous N (in this case 20) transfer rates to be remembered, to which the formula in Equation 6.2 is applied to give an average rate. The exponentially weighted moving average was chosen mainly for its simplicity, and the harmonic mean was chosen to mirror the approach taken in [1], which finds that using the harmonic mean of 20 previous transfer rates improves encoding rate stability. A value of 0.1 for the α parameter, in Equation 6.1, was chosen as a conservative starting point for the investigation, since the function discounts old values faster with a higher α.

$$Estimate_i = \alpha * Rate_i + (1 - \alpha) * Estimate_{i-1} \tag{6.1}$$

(with α = 0.1)

$$HM(r_1, r_2, \cdots, r_N) = (\frac{1}{N} * \sum_{i=1}^{N} {r_i}^{-1})^{-1} \tag{6.2}$$

I now have three bandwidth estimation approaches, each applicable to all four transfer mechanisms. That gives 12 application models investigated in this chapter, 8 of which are being introduced for the first time. Applications using previous chunk bandwidth estimation are, as before, client pull through multiple connections (**pullm-i-prev**), client pull through a single connection (**pulls-i-prev**), server push (**push-prev**), and client pull with selective requests (**pullsl-prev**). Following the same naming scheme, applications using EWMA bandwidth estimation are labelled **pullm-i-ewma**, **pulls-i-ewma**, **push-i-ewma**, and **pullsl-i-ewma**. Similarly, those using harmonic mean bandwidth estimation are **pullm-i-hm**, **pulls-i-hm**, **push-i-hm**, and **pullsl-i-hm**. For brevity, I can only afford to show simulations using a single set of media parameters in this chapter, and to maintain the theme of improving on the baseline simulations presented in Chapter 4 I revert back to using the default media parameters. These are the **2 second** chunk duration and linear encoding rates **(600, 1200, 2000, 3000, 4000, 5000, 6000 kbps)**. Each ABR flow transfers 600 seconds worth of content over 300 2 second chunks. Again, each application is tested competing against both separate instances of itself and long lived continuous TCP flows, in scenarios with 2, 3, and 4 competing flows in total. This gives 12 applications tested under 6 different scenarios, with each simulation repeated 100 times, giving a total of 7200 simulations represented in total.

6.2 Fairness

Figure 6.1, on page 69, shows fairness plots for all applications competing against both ABR and TCP flows, with only two flows running. Figures 6.1a, 6.1b, and 6.1c show applications using previous chunk, exponentially weighted moving average, and harmonic mean bandwidth estimation respectively. Each individual boxplot shows the spread of values for 100 repetitions of a single simulation. Figures 6.2, and 6.3 follow the same format as Figure 6.1, showing results for 3 and 4 competing flows.

At a glance, it is easy to see that the new bandwidth estimation approaches do not have any significant effect on an applications fairness. In Figure 6.1 each subfigure is almost identical, with the top figure showing no discrimination in any case when ABR flows compete against similar ABR flows and the bottom showing significant discrimination when ABR flows compete against long lived continuous TCP flows. Figures 6.2, and 6.3 follow a similar pattern. This is because the new bandwidth estimation approaches do not have any effect on the performance of client pull. The chunk duration is still small and idle periods between chunks still exist, so that the reasons for poor performance described in Section 4.2 still apply. This was expected, and any slight differences in fairness that can be seen with the new applications can be accounted for by considering that the new bandwidth estimation techniques will affect the encoding rates chosen, which in turn will affect the number of bytes transferred for each chunk.

6.3 Stability

Figures 6.4, 6.5, and 6.6, starting on page 72 show stability plots for all 12 applications, competing against ABR and TCP flows, with 2, 3, and 4 flows competing. The format is the same as that of figures in Section 6.2. Stability calculated again using Equation 3.2 from

Chapter 3, with each boxplot showing the spread of values for 100 repetitions of a single simulation.

This time, the effect of using an averaging function with bandwidth estimation is significant. In each scenario, applications using EWMA and harmonic mean bandwidth estimation are far more stable, with a stability index close to 1.0, than their counterparts using previous chunk bandwidth estimation in the same scenario. As hypothesised in the introduction to this chapter, using averaging functions such as these dampens fluctuations in observed transfer rates to give a smoother and more more stable series of bandwidth estimates. This, in turn, naturally leads to less unnecessary encoding rate switches, resulting in a more stable stream. Encoding rate distributions in the next section add more evidence to support this conclusion.

6.4 Encoding Rate Distributions

Figures 6.7, 6.8, and 6.9, starting on page 75,follow the same format as those in the previous two sections. In this section each graph contains 4 distributions, one per application, each showing the distribution of encoding rates observed over 100 repetitions of a single simulation of that application in the given scenario.

Two important observations can be drawn from studying these distributions. The first is confirmation that using an averaging function to dampen bandwidth improves the stability of all four basic application models. Distributions in the second and third columns, showing applications using EWMA and harmonic mean bandwidth estimation, have noticeably less variation than those in the first column of each figure. For server push and pull selective applications the effect is obvious in Figure 6.7, but less so for pull multiple and pull single applications. This is because poor efficiency with a 2 second chunk duration has already restricted these applications' choice of encoding rates in those scenarios, as I have shown in chapters 4 and 5, however, in Figures 6.8 and 6.9 the effect can be seen more clearly for pull multiple and pull single applications.

The second observation that can be drawn from these graphs is a tentative advantage for the EWMA over the harmonic mean. In each figure, distributions in the second column, for applications using EWMA bandwidth estimation, are at least slightly better than those for the corresponding application using harmonic mean bandwidth estimation in the third column, and often significantly better. Although both show far less variation than applications using previous chunk bandwidth estimation, distributions for applications using the EWMA are always weighted more to the right than their counterparts for applications using the harmonic mean. At this stage more research would be required to strengthen this conclusion, but it seems that the harmonic mean, which favours smaller values in the list, gives a more conservative average than the EWMA.

6.5 Playback Buffer Occupancy Distributions

The danger of dampening the bandwidth estimate using an averaging function is that signals from genuine changes of conditions on the network will take longer to filter through, which could potentially result in a playback interruption if there not enough content buffered already. It is important, therefore, to demonstrate that using an averaging function to smooth the bandwidth estimates does not solve one problem and introduce another. Figure 6.10, on page 78, confirms that using harmonic mean and EWMA bandwidth estimation techniques does not lead to playback interruptions. Figure 6.10 shows playback buffer occupancy distributions for scenarios with 2 competing flows, none of which show any time spent with an empty buffer. Figures for scenarios with 3 and 4 competing flows are not shown for brevity, but they do follow similar patterns. Again, other differences between the shapes of these distribtions are less important and the reasons for those differences are not entirely clear.

6.6 Discussion

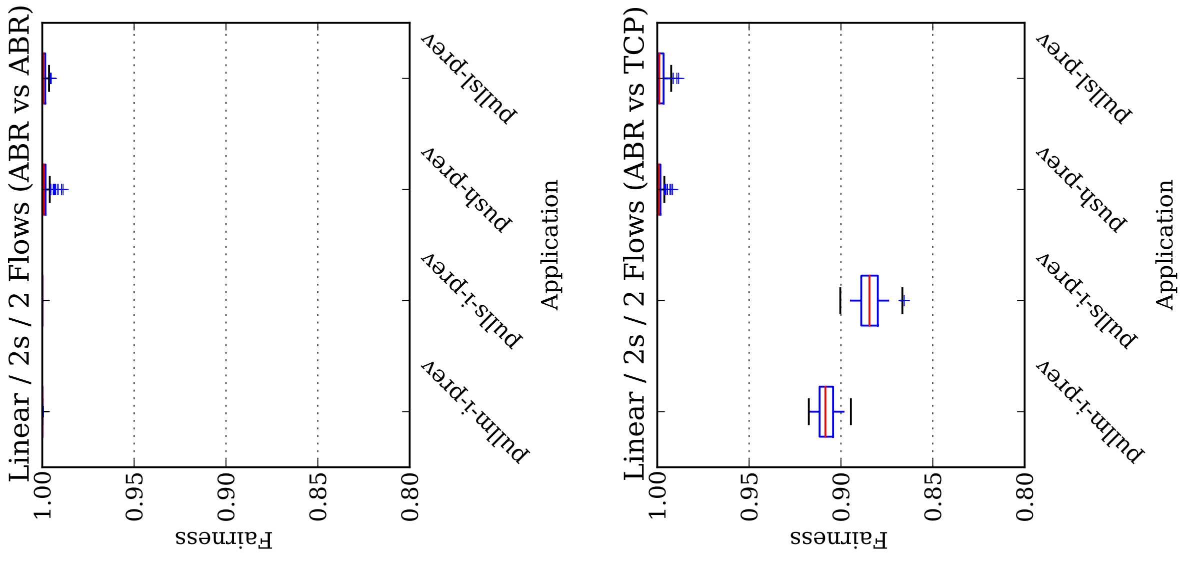
Stability plots and encoding rate distributions from this chapter show that using an averaging function, over a series of recent transfer rates, results in a much more stable bandwidth estimate than simply using the transfer rate of the previous chunk. Transfer rates observed above the HTTP layer hide complexity beneath, and fluctuate over short time scales due to the behaviour of the underlying TCP protocol and its congestion control mechanisms. Using an averaging function can dampen these fluctuations to give a more stable series of bandwidth estimates, which in turn leads to less variation in encoding rate choices and fewer unnecessary rate switches, and a more stable stream gives the user a better viewing experience [38]. Both [3] and [1] also suggest filtering bandwidth estimates, and [1] shows a similar improvement in stablity from their experiments. The tradeoff, however, is that dampening the bandwidth estimates may risk not allowing signals, from genuine changes on the network, to reach the application in time for it to react before stalling. Figure 6.10 confirms that this does not happen in any of my simulations.

Of the averaging functions used in these simulations, using the EWMA of all previous chunks' transfer rates seems to give better results than using the harmonic mean of the last twenty chunks' transfer rates. Both have a similarly positive effect on stability, but the EWMA approach allows the applications to achieve higher average rates. However, it is not clear how changing the parameters of these functions will affect things, and being more conservative could be argued to be a safer approach. Although applications using the EWMA achieved higher average rates than their counterparts using the harmonic mean, this could prevent the system from building an adequate safety net in the playback buffer, thus leaving it at a higher risk of stalling in the future. In this case we trade quality for safety, and finding the correct balance is likely to be a case of trial and error. One thing that remains clear is that using some form of smoothing function with bandwidth estimation based on transfer rates will yield better results than none at all. Further work is needed to understand in detail the effect of both the α parameter in Equation 6.1, and the number of samples averaged in equation Equation 6.2, and to understand better how these functions compare.

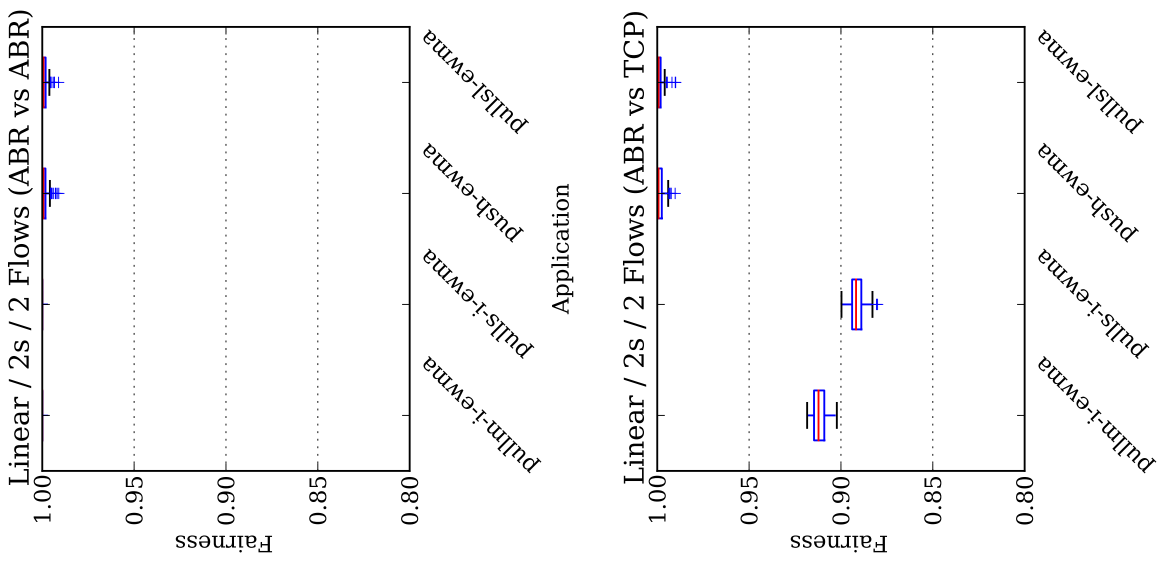
6.7 Summary

In this chapter I have demonstrated that my original four basic application models can be improved by using a more sophisticated approach to bandwidth estimation. Using an averaging function over a series of recent chunks' transfer rates to estimate bandwidth dampens out short term fluctuations, resulting in much more stable streams than simply using the previous chunks transfer rate as the bandwidth estimate. I showed this using both an EWMA of all previous chunks' transfer rates and the harmonic mean of the previous 20 chunks' transfers rates. The EWMA gave marginally higher average encoding rates in my simulations, but this is not conclusive. There also exists a fundamental tradeoff between smoothing the bandwidth signal to achieve better encoding rate stability, and being able to react quickly to genuine congestion. Finding the correct balance will probably require an element of experimentation for the parameters any given system.

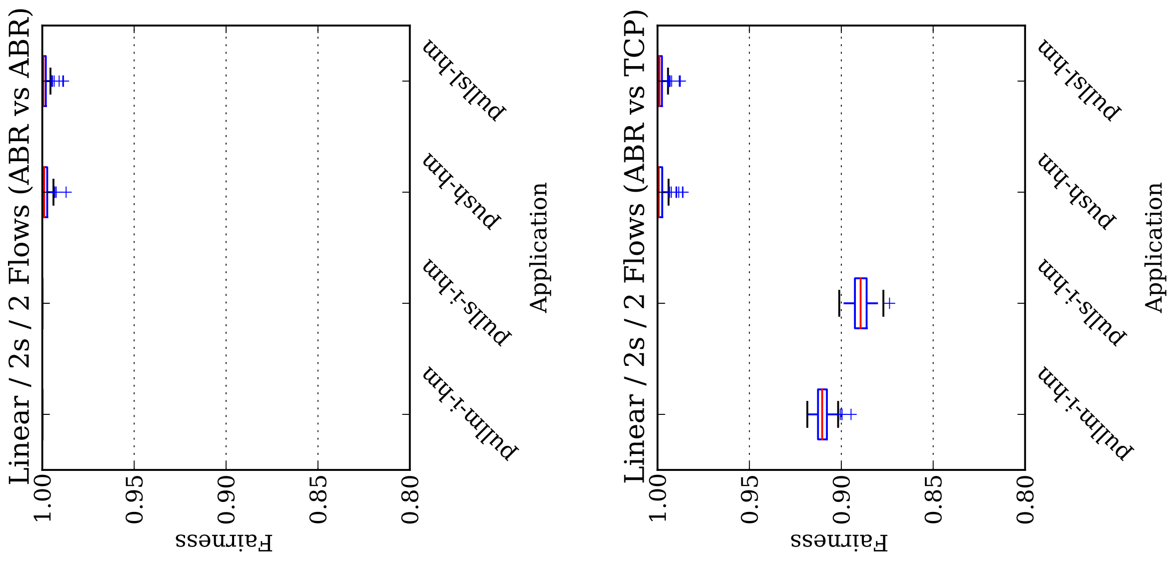
In the following chapter I investigate another component involved in adaptation logic, this time considering the effects of different request scheduling strategies for client pull based applications. Until now all client pull applications have sent requests for the next chunk to the server immediately after each previous chunk has finished downloading. I will explore the reasons why a different approach may be desirable, what better approaches might look like, and how applications using them compare to those tested so far.



(a) Previous Chunk

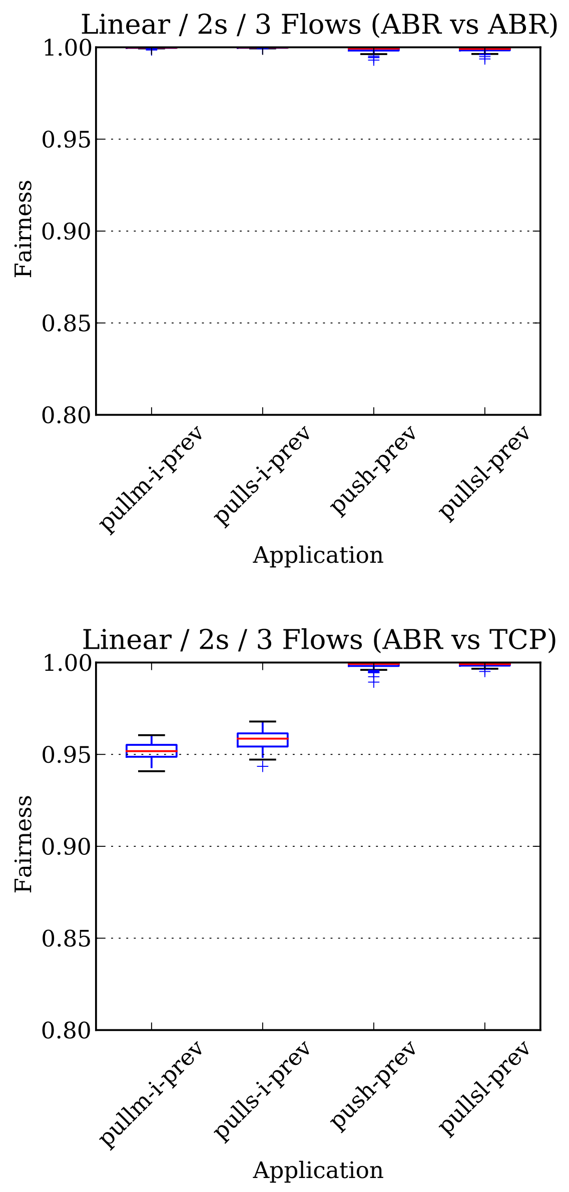


(b) EWMA

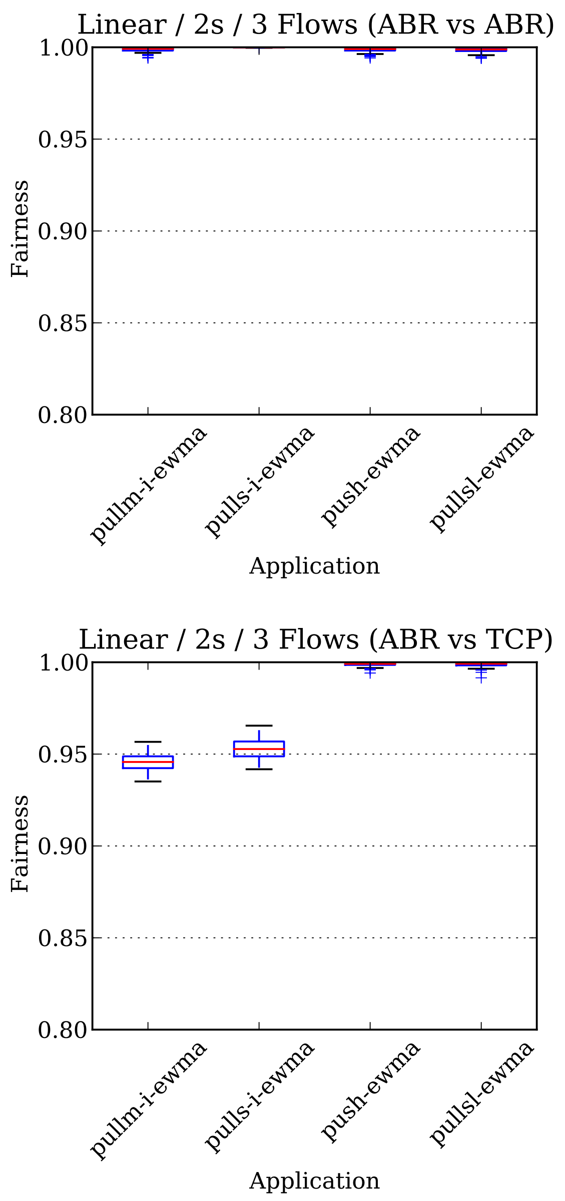


(c) HM20

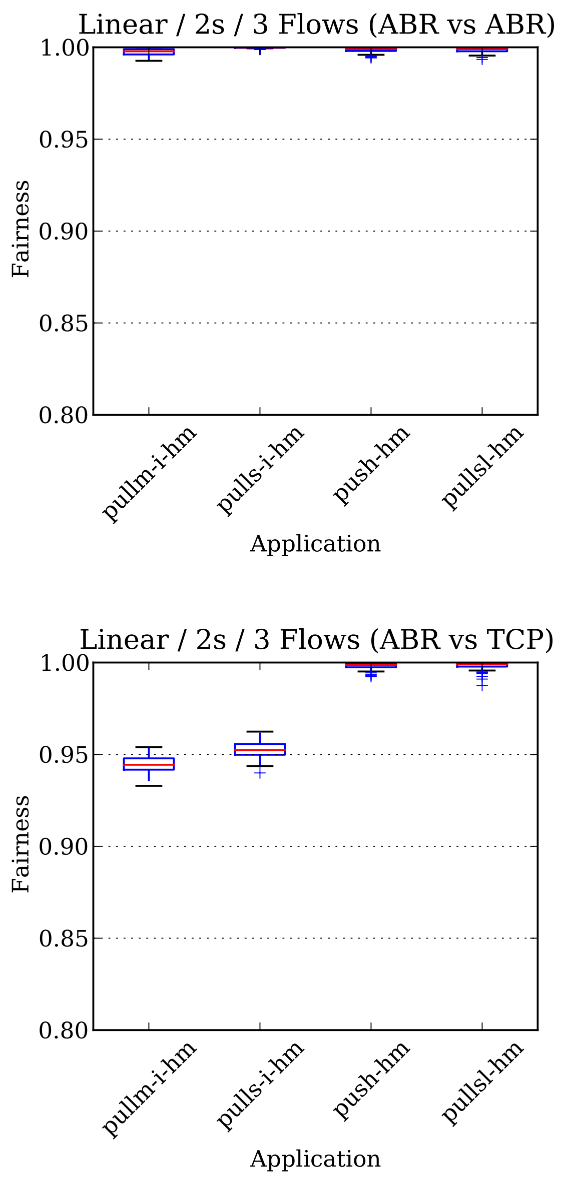
Figure 6.1: Bandwidth Estimation - Fairness (2 Flows)



(a) Previous Chunk

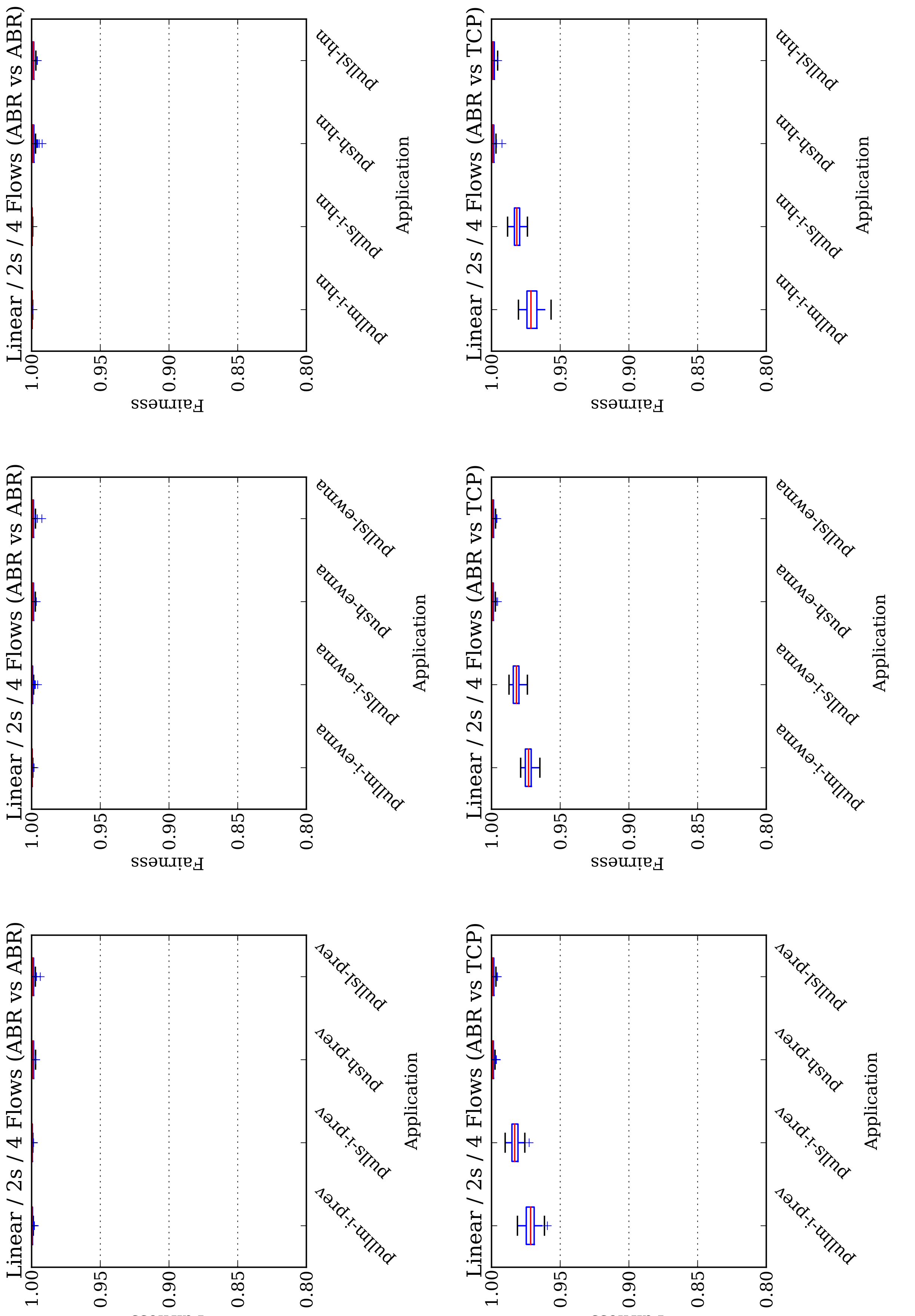


(b) EWMA



(c) HM20

Figure 6.2: Bandwidth Estimation - Fairness (3 Flows)

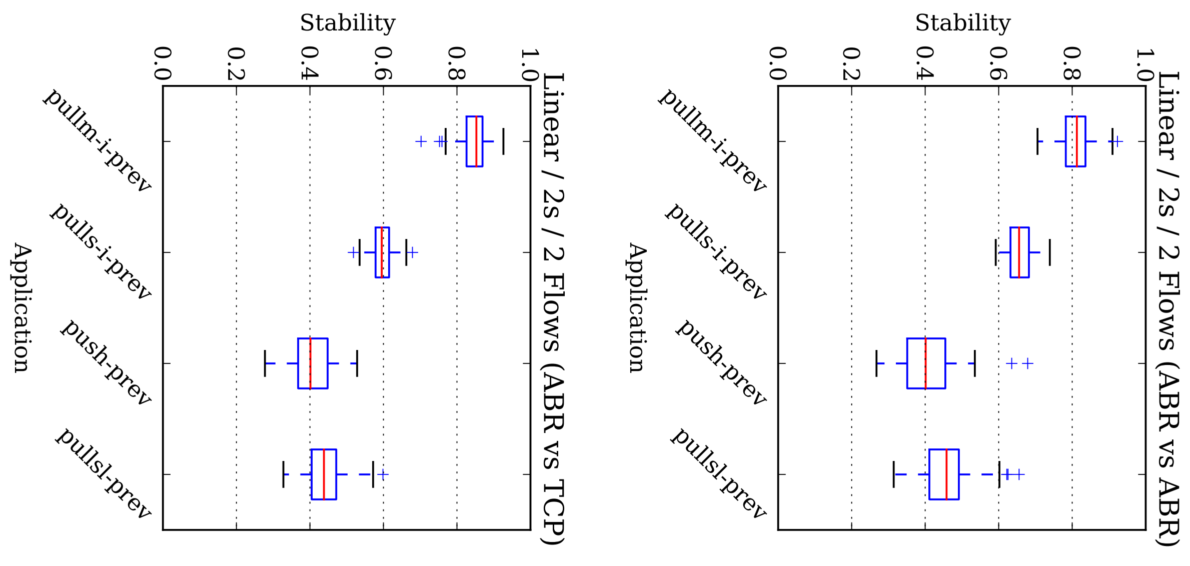


(a) Previous Chunk

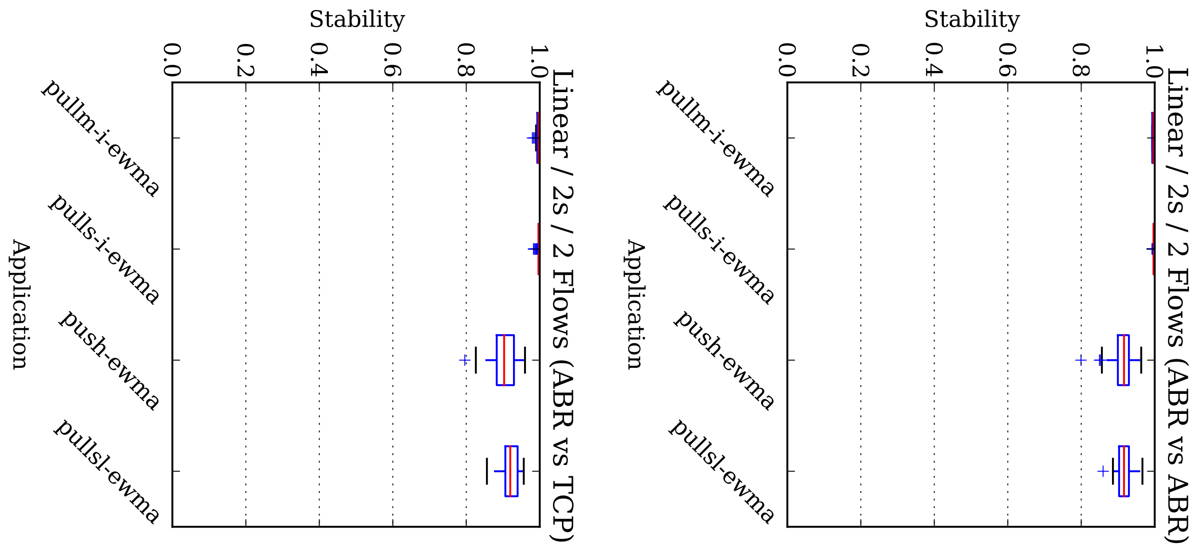
(b) EWMA

(c) HM20

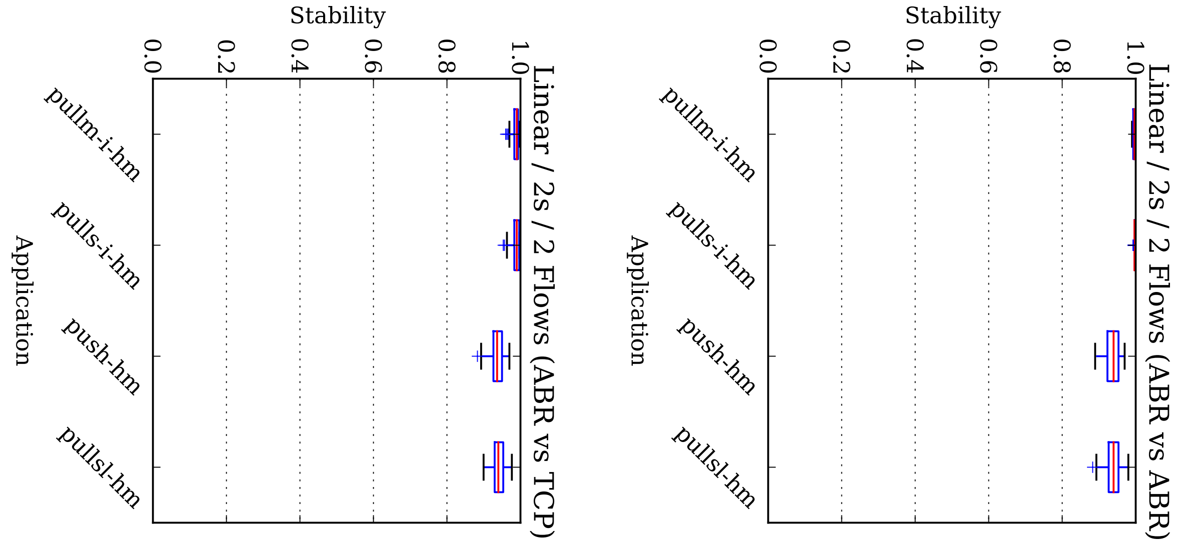
Figure 6.3: Bandwidth Estimation - Fairness (4 Flows)



(a) Previous Chunk

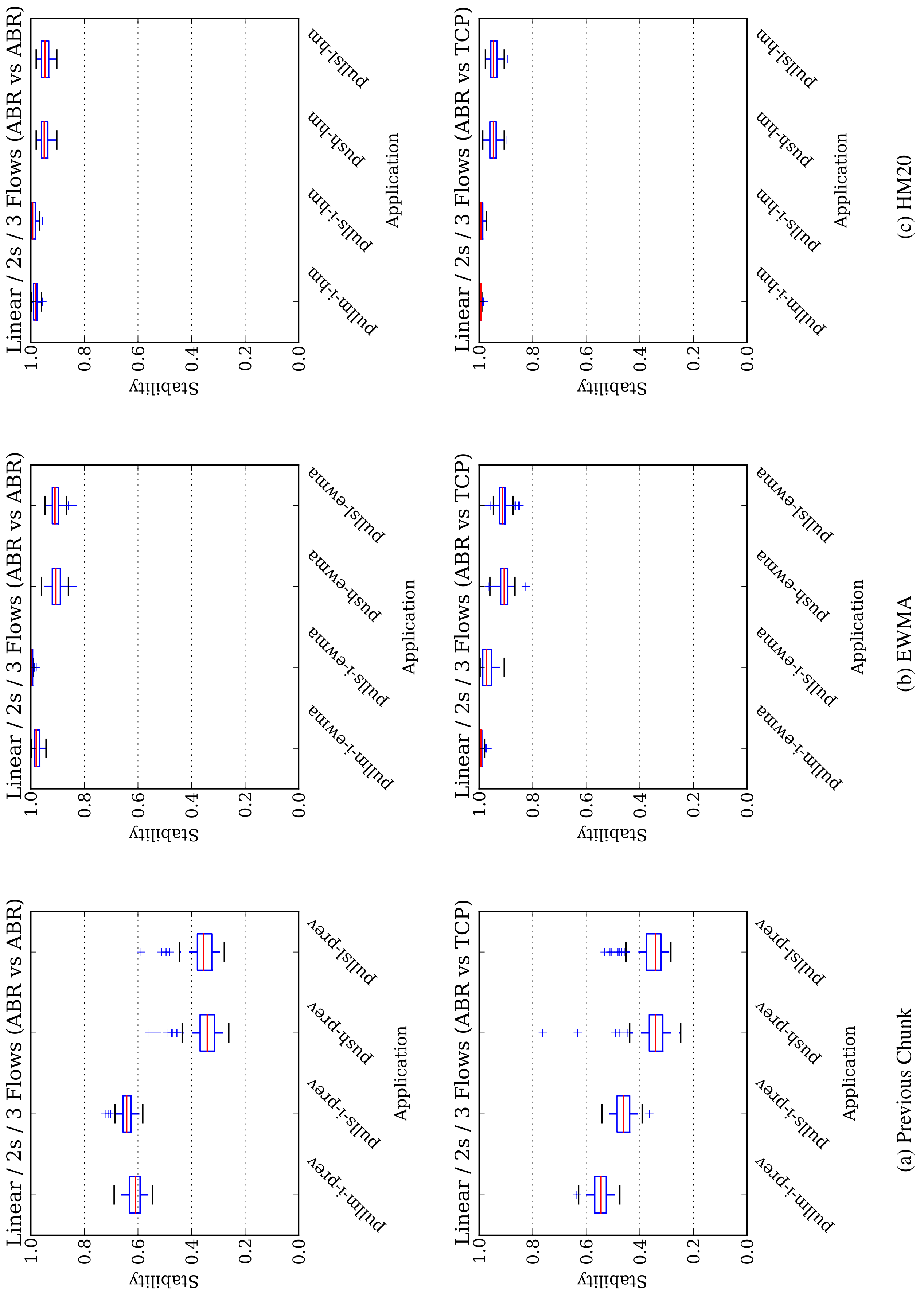


(b) EWMA



(c) HM20

Figure 6.4: Bandwidth Estimation - Stability (2 Flows)

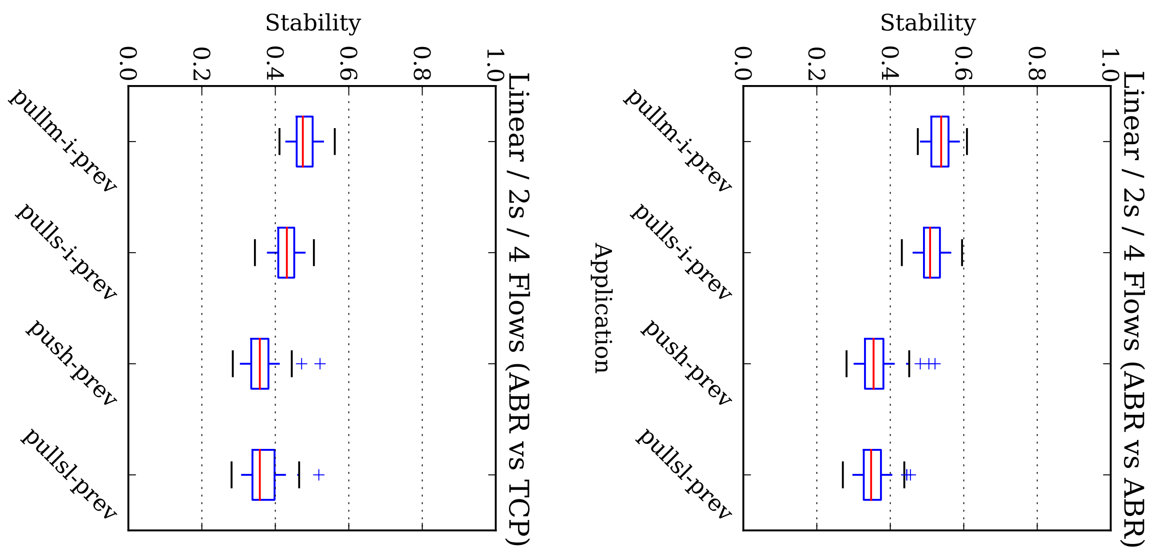


(a) Previous Chunk

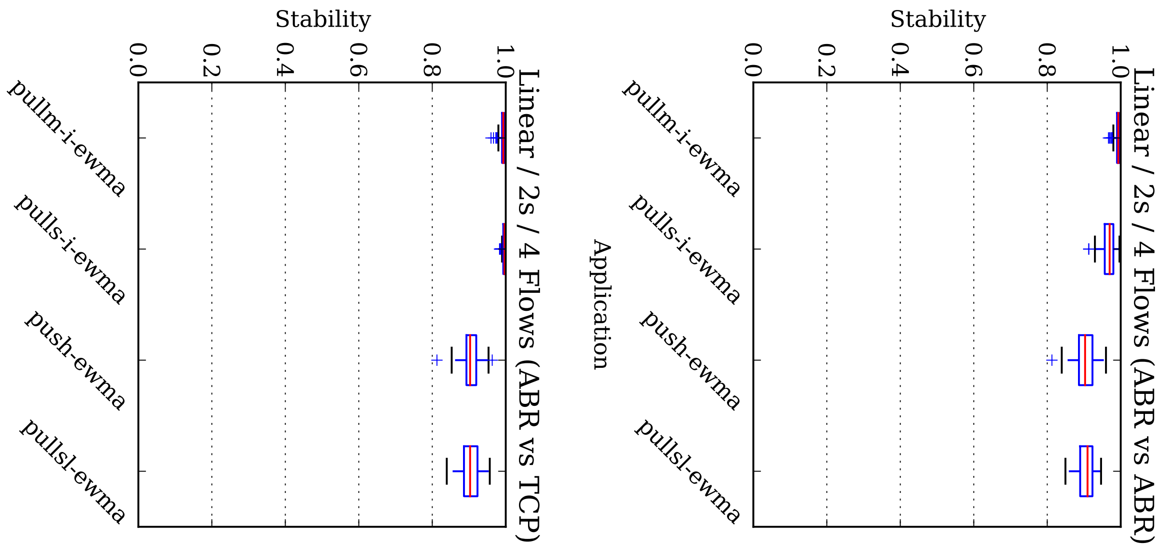
(b) EWMA

(c) HM20

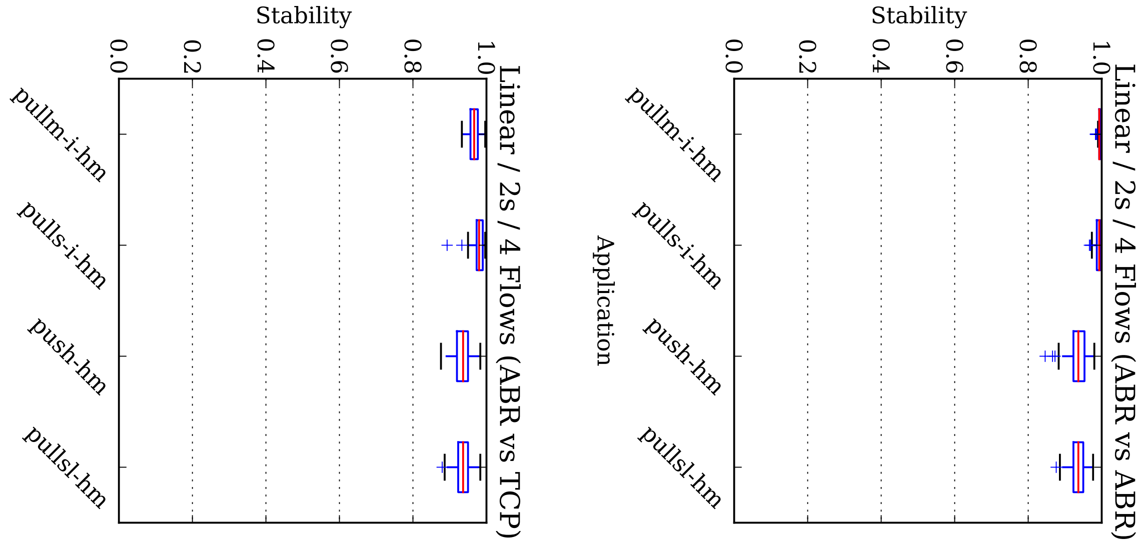
Figure 6.5: Bandwidth Estimation - Stability (3 Flows)



(a) Previous Chunk

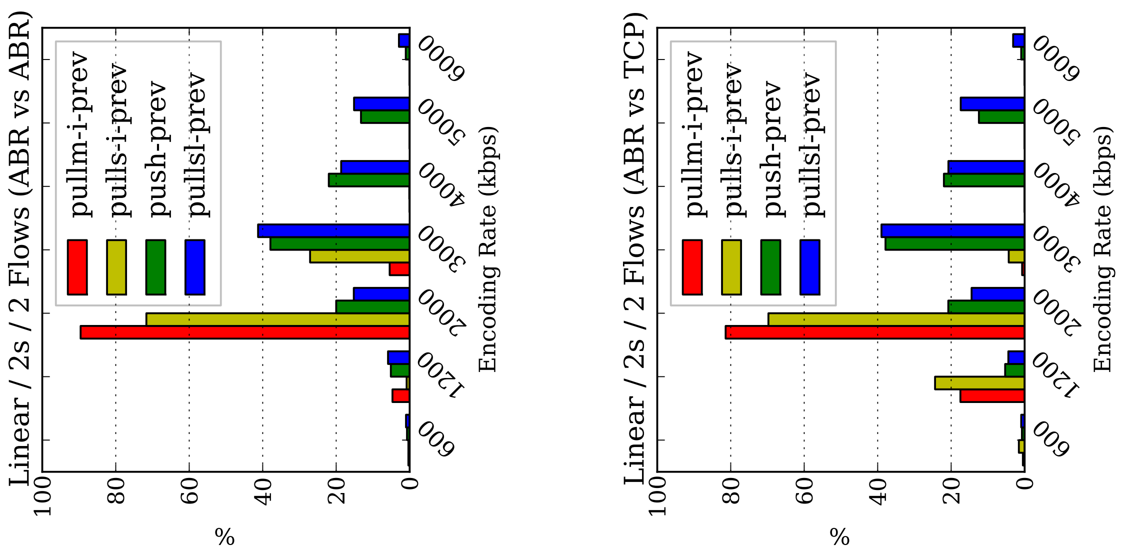


(b) EWMA

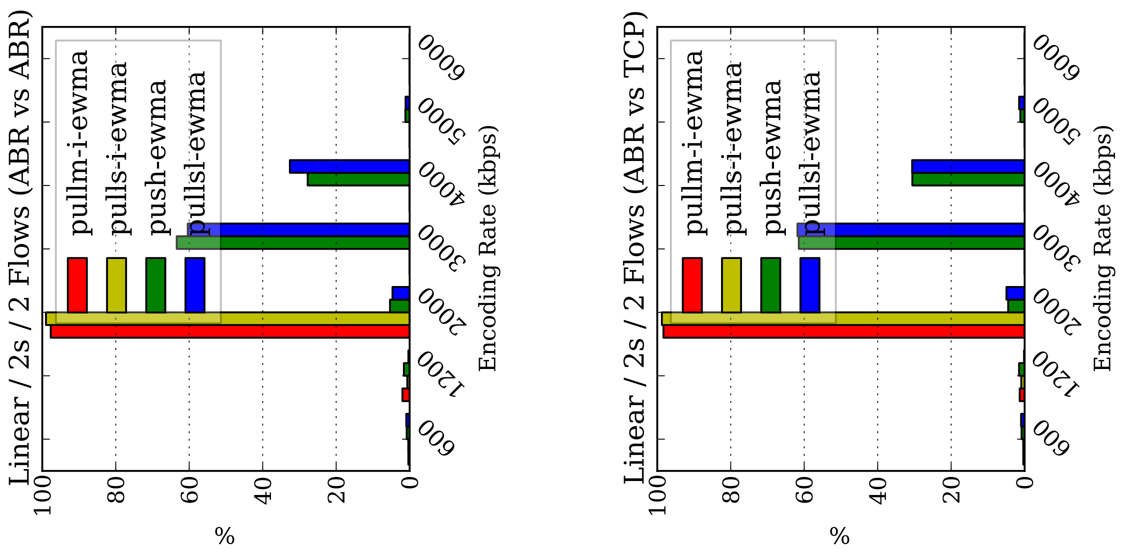


(c) HM20

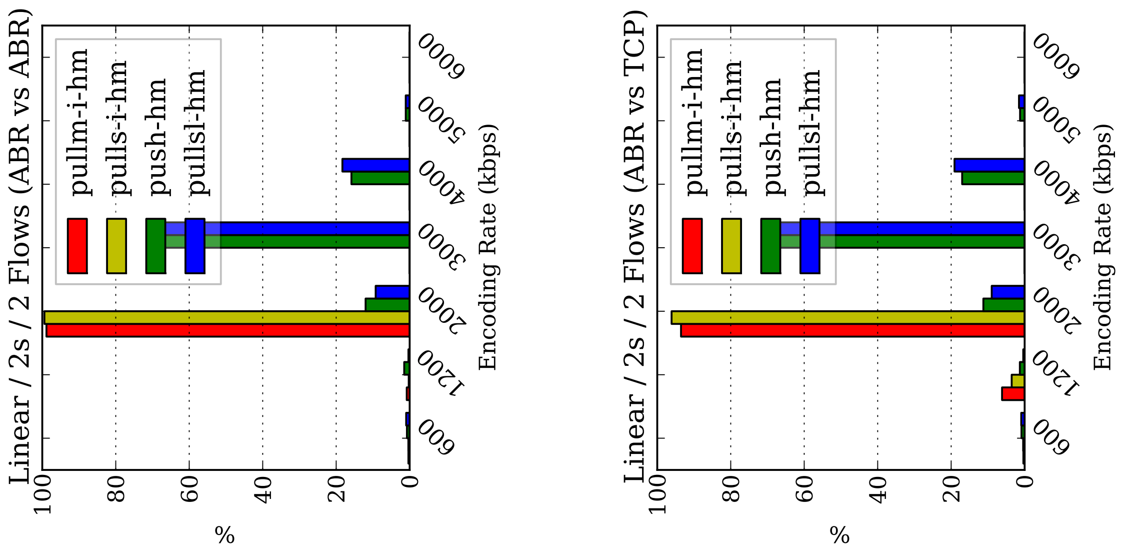
Figure 6.6: Bandwidth Estimation - Stability (4 Flows)



(a) Previous Chunk

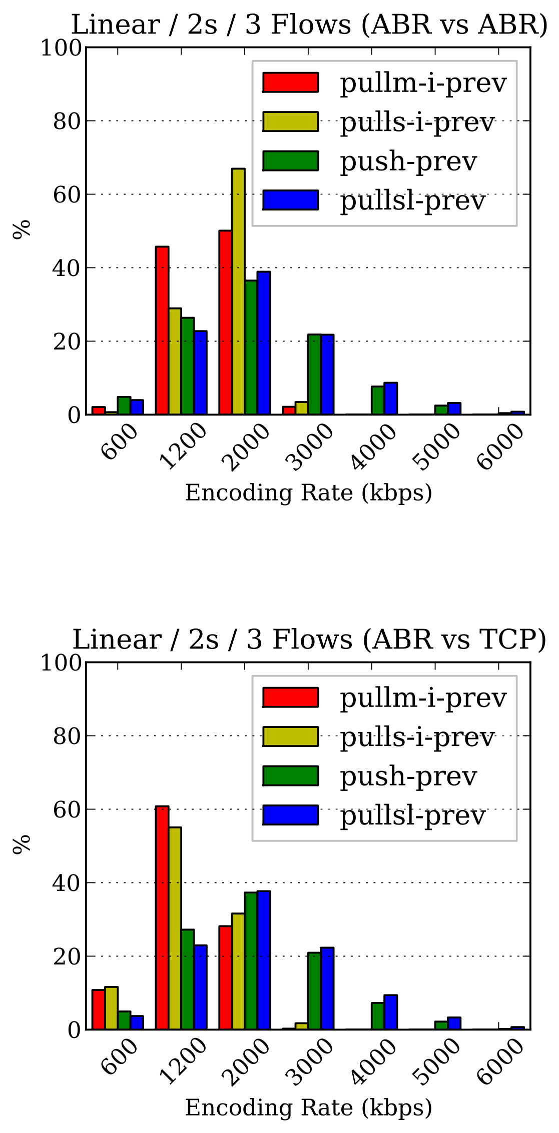


(b) EWMA

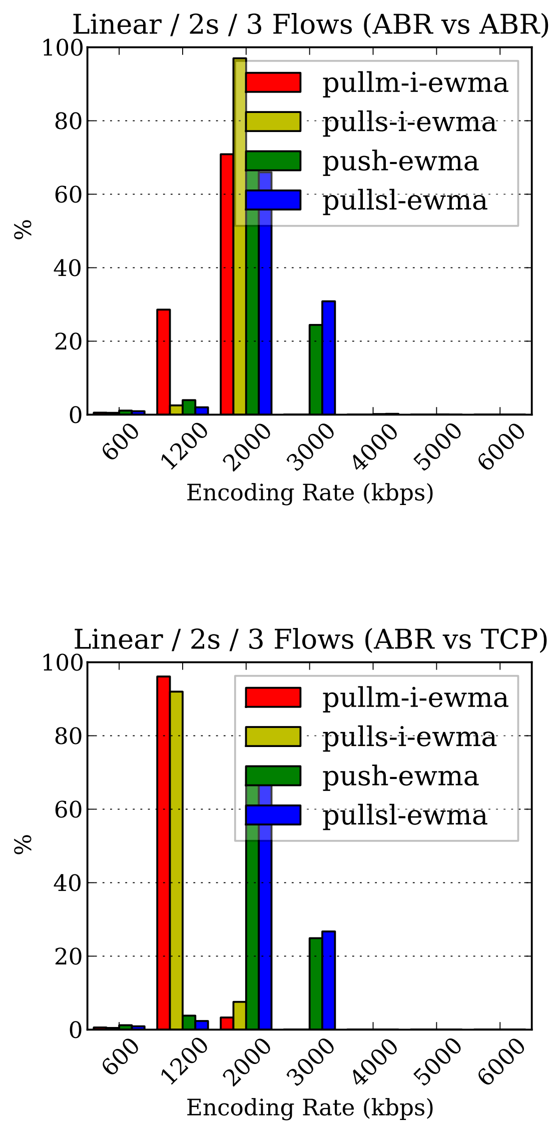


(c) HM20

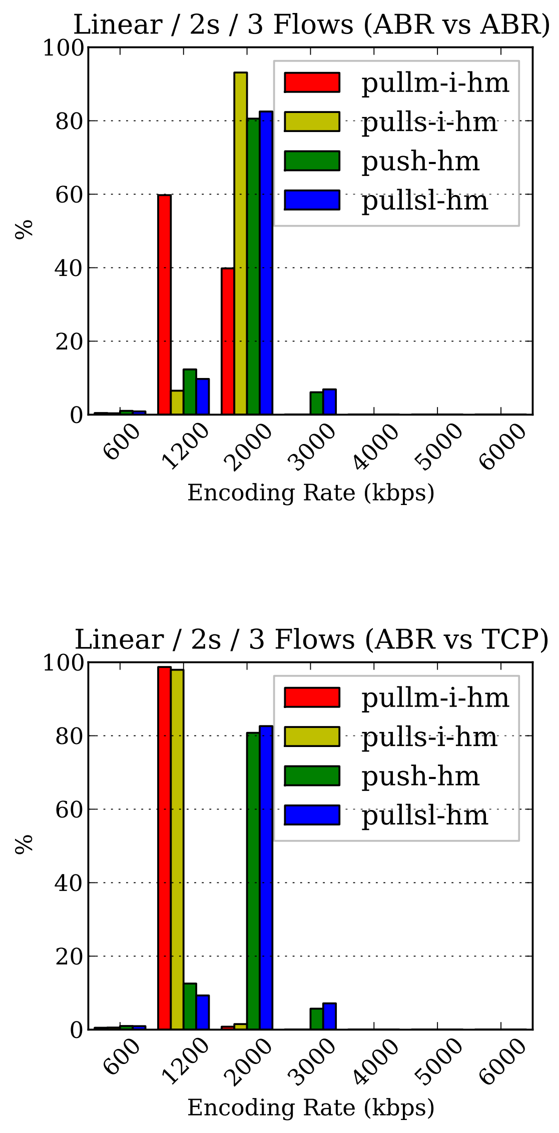
Figure 6.7: Bandwidth Estimation - Encoding Rate Distributions (2 Flows)



(a) Previous Chunk

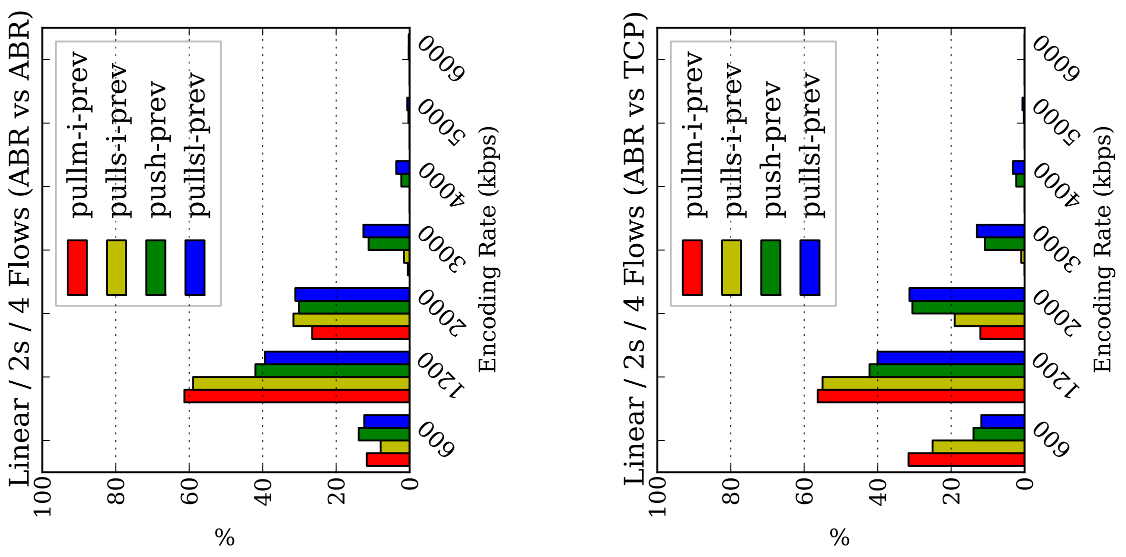


(b) EWMA

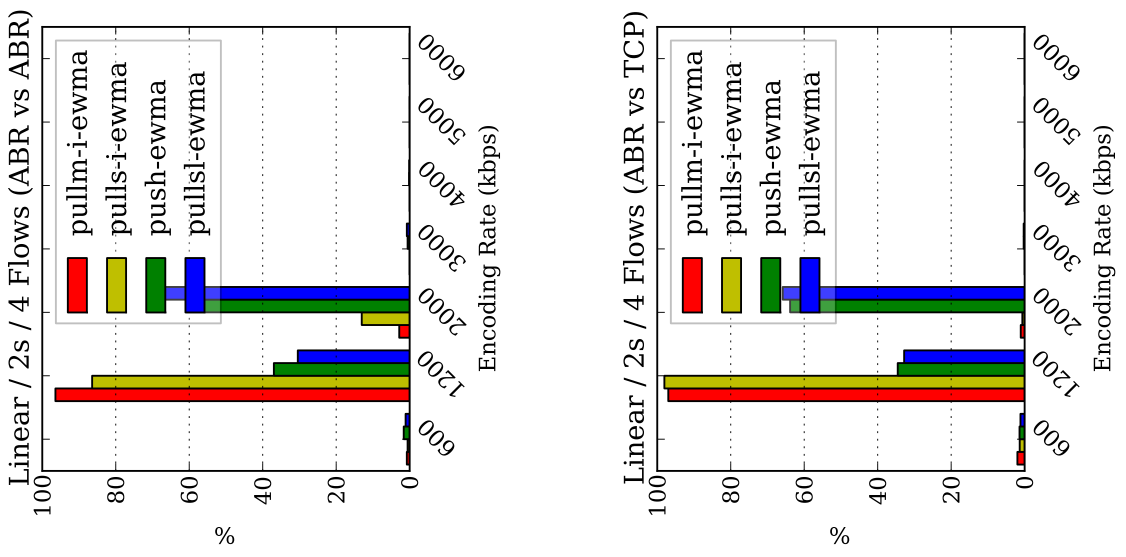


(c) HM20

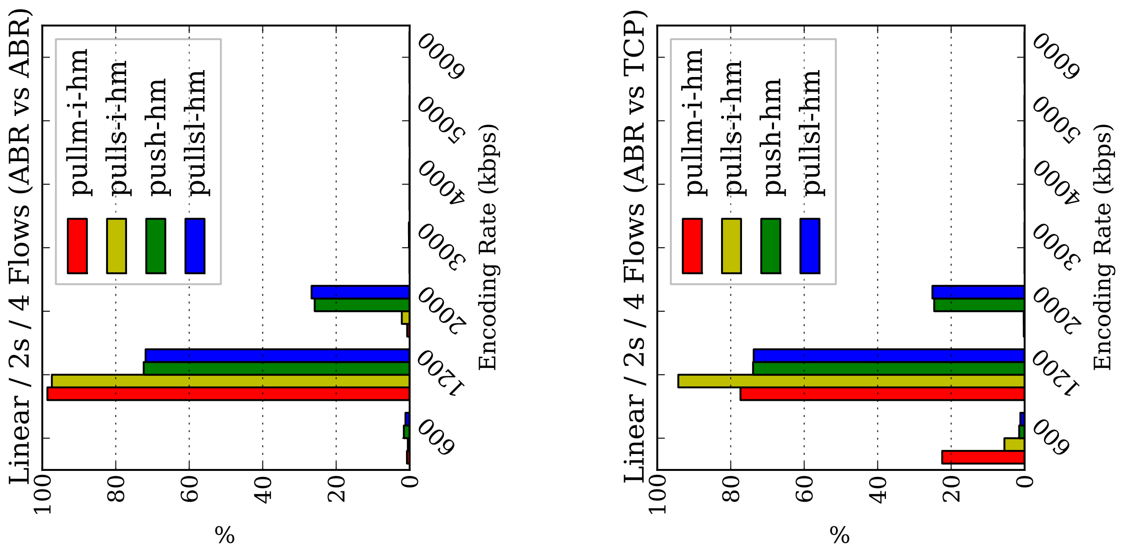
Figure 6.8: Bandwidth Estimation - Encoding Rate Distributions (3 Flows)



(a) Previous Chunk

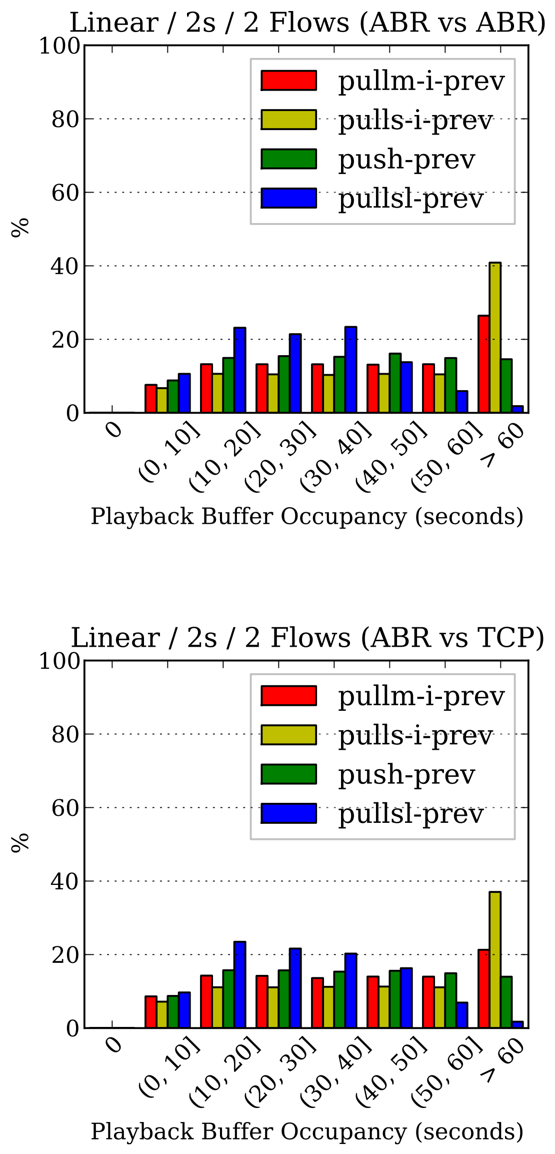


(b) EWMA

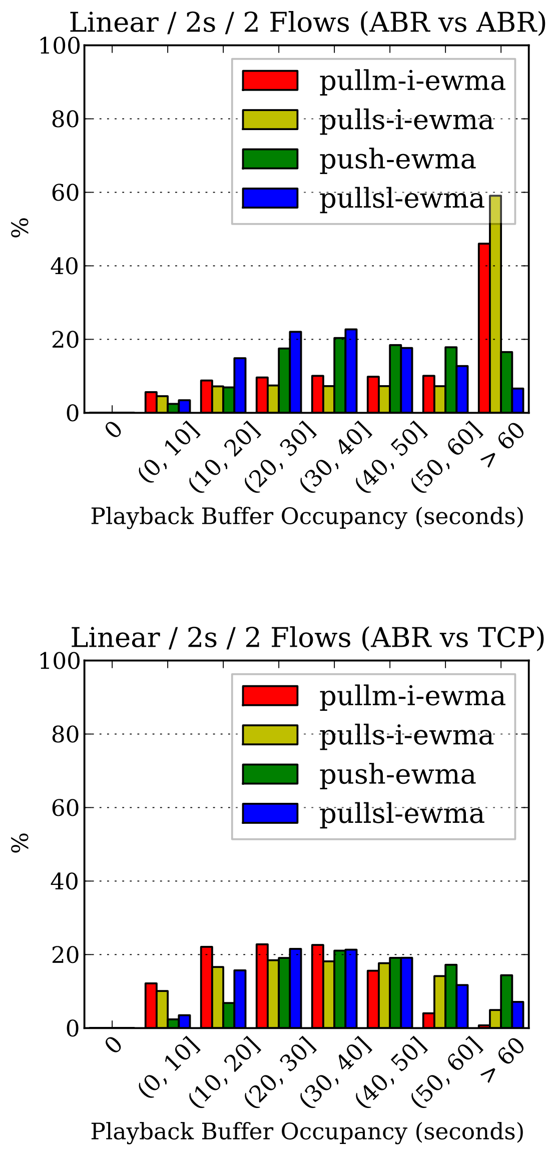


(c) HM20

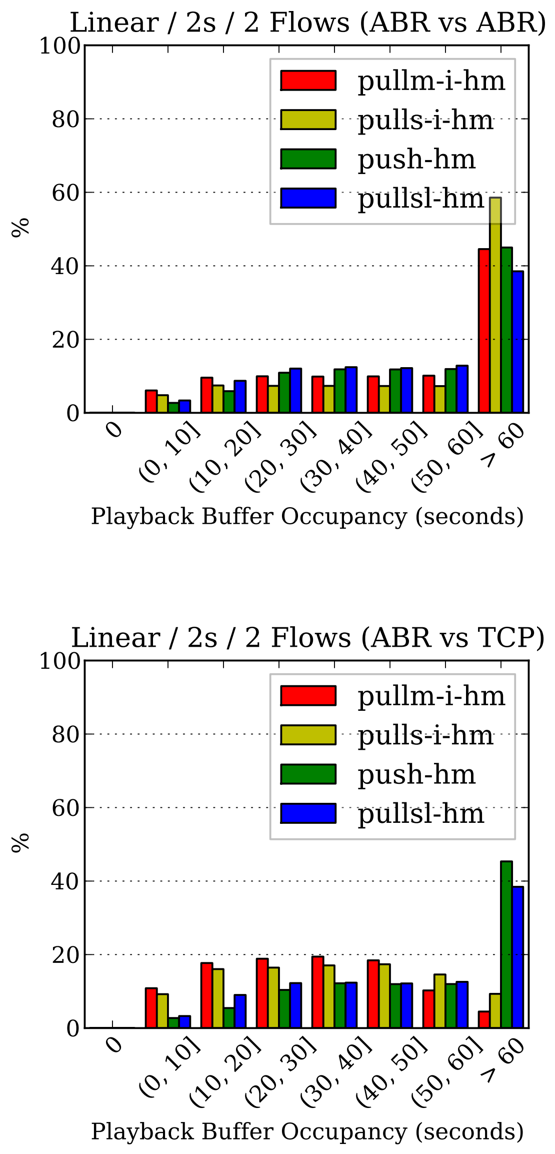
Figure 6.9: Bandwidth Estimation - Encoding Rate Distributions (4 Flows)



(a) Previous Chunk



(b) EWMA



(c) HM20

Figure 6.10: Bandwidth Estimation - Playback Buffer Occupancy Distributions (2 Flows)

Chapter 7

Request Scheduling Strategies

In the previous chapter I looked at different approaches to bandwidth estimation, and showed that applying an averaging function over a series of recent transfer rates, in order to estimate bandwidth, results in a much more stable system than simply using the most recent transfer rate. I will now investigate another important component of some ABR systems, which is the request scheduling strategy used in client pull systems. I will describe three separate strategies for scheduling requests to the server, and investigate their effects on the behaviour of client pull ABR flows. I will also compare my results with those from another recent study that looked at the same three strategies [1].

7.1 Details

In a client pull based system, after each chunk has finished downloading, the client must send a request to the server for the next chunk at the desired encoding rate. Server push and pull selective applications gain some advantage from being able send each chunk after the previous without any delay, thus maintaining a high throughput on the TCP connection, it would also be difficult for a server to know when a client might want to wait before receiving the next chunk or how long it should wait. But for client pull applications more options are available, and it makes sense to give careful consideration to the request scheduling strategy. An obvious approach is to simply schedule each request to happen as soon as possible, building up locally buffered content indefinitely in order to safeguard against future congestion and possible stalls, but this approach is far from optimal in most situations. If a client leaves prematurely then buffered content will be discarded and bandwidth has been wasted, and there is also the risk of downloading chunks at an unnecessarily low rate when there is no rush and conditions may well improve later. Taking these types of concern into account, other approaches revolve around the idea of scheduling requests to maintain a target volume of buffered content. This approach could also be implemented with a server push based approach by having the server estimate the client's playback state given the encoding rates and number of chunks it has sent. However, this is more complicated to implement and there is scope for problems arising if the estimates deviates significantly from the true playback state, which is why I only consider scheduling strategies for client pull applications.

On top of the basic **immediate** request scheduling strategy, I first consider a **periodic** request scheduling strategy that tries to avoid over buffering by maintaining a target playback buffer occupancy. This works by first selecting a target duration of playback buffer occupancy, then scheduling requests to be sent when the current buffer occupancy falls just below that target. When the chunk has finished downloading, the buffer occupancy jumps up beyond the target occupancy again and the next request is scheduled. Eventually the system should fall into a repetitive pattern with buffer occupancy hovering above and below the target occupancy. The formula used to calculate the request scheduling delay in this case is shown in Equation 7.1, where C is the current buffer duration and T is the target duration.

$$Delay = \begin{cases} 0, & \text{if } C < T. \\ C - T, & \text{otherwise.} \end{cases} \tag{7.1}$$

The periodic request scheduling strategy avoids over buffering, but has also been shown to introduce synchronisation issues that lead to unfair allocation of encoding rates, depending on the relative starting times of competing ABR flows. The authors of [1] demonstrate this effect and refer to it as the application starting time bias. Long delays between requests mean applications can see a biased view of the network while other applications are idle, thus overestimating their share of the available bandwidth, and the periodic timing of transfers can result in a pattern of repetition where one application consistently overestimates bandwidth while another competing application consistently underestimates. In this situation the relative starting time for each flow determines how they are biased for the rest of the stream. Synchronisation issues are not uncommon in computer networking applications, see [39] for example, and a common solution is to introduce a small element of random jitter to remove the repetitive behaviour. In [1], introducing a small random offset to the target buffer occupancy for each chunk is found to be enough for the application starting time bias and synchronisation problems. For the **random periodic** request scheduling strategy, a similar formula to the one in Equation 7.1 is used to calculate the request scheduling delay after each chunk has been downloaded. In Equation 7.2, R is a small randomly chosen offset calculated after each chunk has finished downloading.

$$Delay = \begin{cases} 0, & \text{if } C < (T + R). \\ C - (T + R), & \text{otherwise.} \end{cases} \tag{7.2}$$

My implementations of periodic and random periodic request scheduling both use a 30 second target buffer, as suggested in [1], with the value of R chosen randomly to lie between 0 and 5 seconds. The value for R used in [1] is not given, but the authors suggest that only a small offset is required.

Combining three different request scheduling strategies with both pull multiple and pull single applications, all using previous chunk bandwidth estimation, gives the following six applications tested in this chapter: **pullm-i-prev**, **pullm-p-prev**, **pullm-rp-prev**, **pulls-i-prev**, **pulls-p-prev**, and **pulls-rp-prev**. In this notation, 'i', 'p', and 'rp' each stand for 'immediate', 'periodic', and 'random periodic'. As before, applications are simulated competing against both ABR flows and long lived TCP flows, with varying amounts of cross traffic, and using the original set of 7 linear encoding rates **(600, 1200, 2000, 3000, 4000, 5000, 6000 kbps)** and **2 second** chunk duration. The analysis of these simulations is given in the following four sections by fairness plots, stability plots, encoding rate distributions, and playback buffer occupancy distributions.

7.2 Fairness

Figures 7.1 and 7.2, starting on page 89, show fairness plots for all six applications competing against both ABR flows and long lived TCP flows. Figure 7.1a shows graphs with three pull multiple applications competing against increasing numbers of ABR cross flows, using immediate, periodic, and random periodic request scheduling. Figure 7.1b shows the same for three pull single applications, and likewise for Figures 7.2a and 7.2b with long lived TCP cross flows.

Plots in Figure 7.1 show little difference between applications using immediate, periodic, and random periodic request scheduling, for both pull multiple and pull single. Experiments in [1] suggest both that periodic scheduling results in unfair allocation of resources between competing ABR flows, and that random periodic scheduling solves the problem. This means there seems to be a conflict between the two sets of results, which I will discuss further in Section 7.6.

The plots in Figure 7.2, however, where applications compete against long lived TCP flows, show lower Jain's index values for applications using periodic and random periodic request scheduling. This is expected given the way fairness is calculated. Jain's index is applied to the total number of bytes sent by each competing flow during the time in which the main ABR flow is active on the network. Increasing the length of idle periods between transfers for the ABR flow, during which long lived TCP flows continue to send data, will clearly result in more discrepancy between the ABR flow and competing long lived TCP flows.

7.3 Stability

Figures 7.3 and 7.4, starting on page 91, show stability plots in the same format as the fairness plots discussed in the previous section. Plots in Figure 7.3a show pull multiple applications using the three different request scheduling strategies, competing against increasing numbers of ABR flows, and likewise with Figure 7.3b for pull single applications. Figures 7.4a and 7.4b show the same for applications competing against long lived TCP flows.

Neither sets of plots show any consistent effect on stability between the immediate, periodic, and random periodic request scheduling strategies. The top left plot in Figure 7.3 appears to show some increase in stability when the two periodic strategies are used, and the top left plot in figure 7.4 appears to show a decrease. However, I have discussed in previous chapters how stability measurements can be thwarted by poor efficiency for client pull applications using a short chunk duration, and there is no noticeable difference between the three approaches in any scenario involving three or four competing flows. Periodic and random periodic request scheduling were never postulated to improve stability, and these results also show that they do not seem to have any obvious negative effect either.

7.4 Encoding Rate Distributions

Figures 7.5 and 7.6, starting on page 93, show encoding rate distributions for all six applications again competing in different scenarios, with individual plots laid out in the same format as the previous two sections. This time, each plot contains three distributions, one for each variant off either pull multiple or pull single, using a different request scheduling strategy. Each distribution shows the distribution of encoding rates chosen over 100 repetitions of that simulation.

The red distributions in each scenario follow the same pattern that has been observed in previous chapters for client pull applications using a short chunk duration. Distributions average below the fair share value for an 8Mbps bottleneck, and those in the top row of each figure show less variance than those below where there is more competing traffic and more rates become available at the low end of the spectrum. Introducing periodic and random periodic request scheduling, whose distributions are shown in yellow and green, does not have any noticeable or consistent effect. This is not surprising, given that plots in the previous two sections have also shown little difference between the three strategies, but playback buffer occupancy distributions in the following section will highlight the real reason for wanting to adopt the periodic approach.

7.5 Playback Buffer Occupancy Distributions

Figures 7.7 and 7.8, starting on page 95 show playback buffer occupancy distributions for all six applications in the same set of scenarios as have been shown throughout this chapter. None of the distributions in either Figure have any bars in the equals 0 state. This is good and shows that adaptation is working, applications are avoiding stalling by using lower rates when it would not be possible for each application to stream at the highest possible rate. However, there is a significant noticeable difference between applications using immediate request scheduling and those using periodic or random periodic request scheduling. The red distributions, for applications using immediate request scheduling, show a roughly equal amount of time spent in each of the first six states after equals 0, with a slight peak in the over 60 seconds state. This is the pattern that would be expected for an application that continues to build up its playback buffer indefinitely at a steady rate, while spending roughly the same amount of time in each state as it builds up buffer, then again as it falls to 0 when all content has been downloaded.

But yellow and green distributions follow a distinctly different pattern in every plot. Instead of continuing to build up buffer indefinitely, they peak either in the (30s, 40s] or (40s, 50s] states, with very little time spent beyond them. In other words, applications using periodic and random periodic request scheduling strategies spend most of their time with a playback buffer occupancy of somewhere between 30 and 50 seconds, thus avoiding any of the problems associated with over buffering. The difference between where distributions for the two strategies tend to peak can be accounted for by the extra random offset that is added to the target each time a request delay is calculated with the random periodic strategy.

7.6 Competing ABR Flows

In Section 7.2, I found that periodic scheduling did not result in any unfairness between competing ABR flows in my simulations, which is at odds with findings from [1]. In that study, the authors find that synchronisation issues with periodic scheduling lead to an application starting time bias resulting in unfair allocation of encoding rates between competing ABR flows, and also that random periodic scheduling remedies the problem. My first thought in trying to understand this conflict was that differences between analysis techniques used in my work and theirs may be the cause. I measure fairness by considering the difference in bytes transferred by competing flows, over the entire course of both flows, whereas in [1], fairness is measured over time by comparing the difference in encoding rates used by each competing flow at discrete intervals over the course of the flows. Considering the difference between encoding rates versus number bytes sent should not be a significant problem, since if one application is consistently getting higher encoding rates then it will also send more bytes and the two indexes should agree. But it could be that while my simulations are unfair at certain points over the lifetime of a stream, this averages out for the entire stream so that my measure of fairness does not highlight the discrepancies. However, after examining time series graphs from my own simulations, it became clear that this was not the problem, and that my simulations of competing ABR flows using periodic request scheduling really were fair for most of the duration of the streams, with no significant or long lived periods of unfairness. This can be seen in Figure 7.11, on page 99, which shows chunk encoding rates and transfer durations, along with playback buffer occupancy over time, for a single repetition of two competing ABR flows using my original parameters and periodic request scheduling. Periodic scheduling kicks in just as the buffer occupancy begins to level out, but both flows settle at around 2000 kbps for most of their duration.

To investigate the issue further, I then set about listing the differences between parameters used in both sets of studies, so that I could run more simulations matching their parameters in an attempt to replicate their results better. The differences that I was able to pin down and remove included the encoding rates used, the bottleneck bandwidth, and bandwidth estimation techniques. In the experiments in question from [1], 3 ABR flows compete over a **3**

mbps bottleneck link, using **2 second chunks** encoded at 8 different rates ranging from 350 kbps to 2750 kbps (Assuming they are evenly spaced, I estimate these to be **350, 690, 1030, 1370, 1710, 2050, 2390, and 2750 kbps**). On top of this they use harmonic mean bandwidth estimation looking at the 20 most recently observed transfer rates. Figure 7.9, on page 97, shows fairness results for simulations I ran using the parameters matching those used in [1] as closely as possible. Looking at the two graphs in the middle row with three ABR flows competing, as was done in [1], it is immediately obvious that something interesting is happening. The boxplots for applications using periodic and random periodic request scheduling are showing much more variation than before, with the average fairness index lying well below that of applications using immediate request scheduling. There is also a slight hint of improvement for the pull multiple application using random periodic request scheduling, in the middle row of Figure 7.9a, but nothing drastic or conclusive, and the same cannot be said of the corresponding application in Figure 7.9b. To backup these results, Figure 7.12 shows the same thing as Figure 7.11, but for 3 competing ABR flows, using parameters matching those used in [1] along with periodic request scheduling. Here, there is far more discrepancy amongst encoding rates used by the three competing flows.

I had originally thought that the use of previous chunk bandwidth estimation, rather than harmonic mean bandwidth estimation, might be why my original results for this chapter did not show any unfairness between competing ABR flows using periodic request scheduling. If the problem concerns falling into bad patterns of over estimation and under estimation of available bandwidth, then it could have been that harmonic mean bandwidth estimation was to blame, since once the system finds itself in that situation then using smoothed bandwidth estimates will make it more difficult to escape with just one or two good estimates. However, I also ran simulations using the same parameters as my original ones, but using harmonic mean bandwidth estimation instead, and found no difference in the results. Instead, the findings represented in Figure 7.9 suggest that the problem is sensitive to the combination of bottleneck bandwidth, available encoding rates, and number of competing flows. To gain an intuition for why this might be the case, consider Figures 7.13 and 7.14, starting on page 101. These show zoomed-in versions of Figures 7.11 and 7.12, and give a clearer picture of what is happening for individual chunks. The location of the bars on the x-axis shows the times at which chunks were being transferred, with the gaps between showing periods where that flow was idle. In Figure 7.14 the idle periods are much more apparent, and often comparable in duration to the chunk transfers themselves.

To understand the importance of this, refer back to the explanation, given in Section 7.1, as to why periodic scheduling is thought to cause problems in the first place. The problem has to do with one application seeing a biased view of the network while other applications are idle, thus overestimating the available bandwidth and selecting a higher rate than it should. The higher encoding rate chunks take longer to transfer, so that competing flows do not see this biased view of the network, and instead are likely to underestimate bandwidth since transfers are shorter for chunks with lower encoding rates and therefore find it more difficult to reach a high sending rate. It follows that when idle periods are more prominent, then the problem will be more likely to manifest. Now consider what determines the length of idle periods between chunks when periodic request scheduling is used. When chunks are encoded at a rate just below the bandwidth seen by that flow, transfers will take almost the same length of time as the chunk duration. If C is the current playback buffer occupancy, and a request is sent when C equals 30 seconds, a 10 second chunk might complete downloading by the time C has fallen to 21 seconds. C will then jump by 10 seconds to 31 seconds, and the next request is scheduled to take place after 1 second. On the other hand, if the encoding rate is well below the bandwidth seen by a flow, then transfers will be relatively quick. This time, by the time the chunk has finished downloading C might only have fallen to 25 seconds, so that it will then become 35 seconds and the next request is scheduled to happen after 5 seconds.

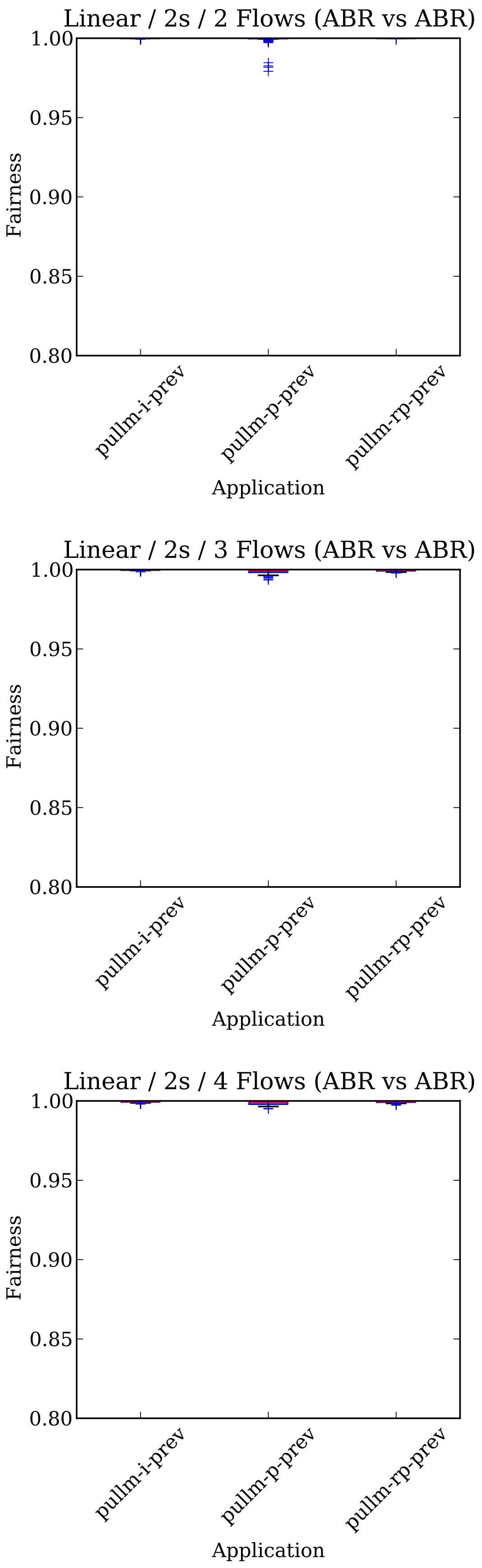
In other words, when the encoding rate is close to the available bandwidth the idle periods are shorter, and vice versa. Since idle periods are the cause of fairness issues, it follows that those issues could well be sensitive to the relationship between encoding rates, available bandwidth, and number of competing flows. Figure 7.9 supports this theory, since the problems only appear noticeably with one combination of those parameters. Going one step further, Figure 7.10, on page 98, shows that tweaking the bottleneck bandwidth from 3 mbps to 4 mbps is enough to change the results so that the fairness issues between competing ABR flows no longer transpire. Graphs in the middle row of Figure 7.10, which shows simulations using a 4 mbps bottleneck link, show very little difference in fairness between competing ABR flows for applications using periodic scheduling and those using immediate scheduling, unlike the corresponding graphs from Figure 7.9, which show simulations using a 3 mbps bottleneck link. This supports the idea that the problem is sensitive to certain parameters, and introduces scope for further work to investigate why that is the case.

7.7 Discussion and Summary

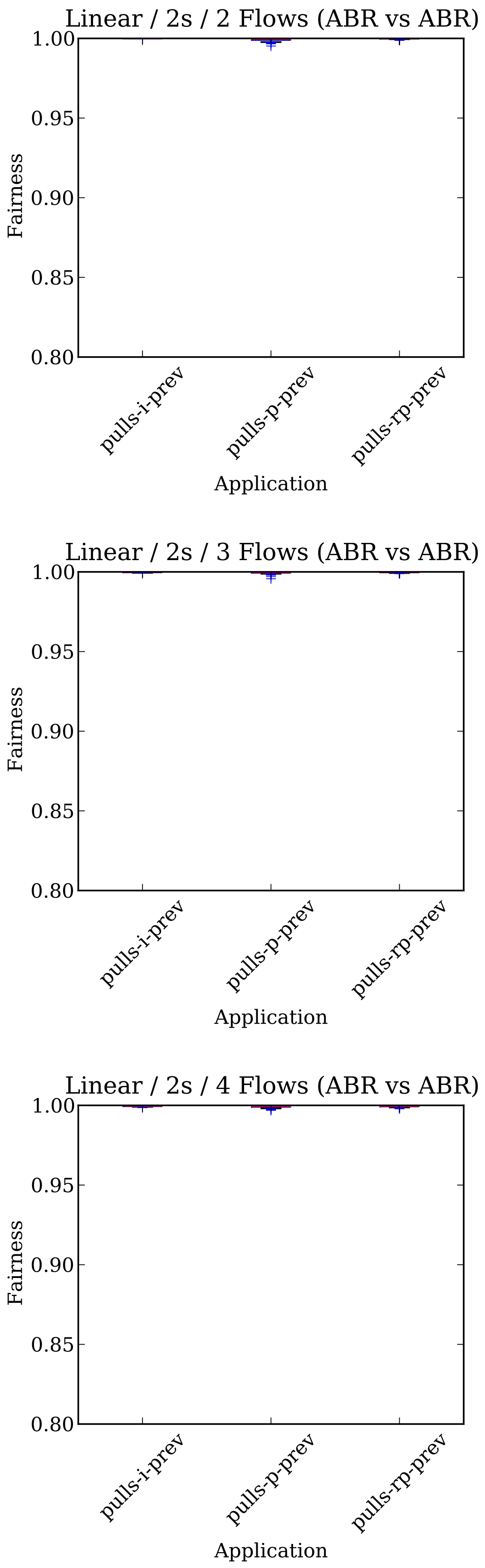
In this chapter I have looked at the effects of introducing different request scheduling strategies to client pull applications. Greedily downloading each chunk immediately after the previous one has finished can lead to over buffering, which risks wasting bandwidth and downloading chunks at a low encoding rate prematurely. For client pull applications, other options are available. One approach, referred to as periodic request scheduling, tries to avoid over buffering by timing requests to maintain a target duration of playback buffer. This is done by selecting a target playback buffer occupancy, then timing chunk requests to occur when occupancy has fallen just below that target. My simulations show that this works as intended. Applications build up their buffer occupancy initially, then hover around the 30 second mark, which is the target duration used, and never have more than 50 seconds worth of content buffered. On top of this, they still manage to avoid ever having any playback interruptions. Fairness plots also revealed a hit on performance when applications using periodic request scheduling compete against long lived TCP flows, but this was expected, given the way fairness is calculated, with longer delays between chunk transfers and continuous cross traffic.

My original results, however, did not show any adverse effect on fairness when two or more ABR flows compete using periodic request scheduling, which contradicts experiments described in [1]. The authors of that study hypothesised that using periodic request scheduling would lead to unfair bitrate allocation between competing ABR flows, due to the introduction of a starting time bias that affects each application's bandwidth estimates, and also gave evidence through experiments that this was the case. They also showed that a common solution to this type of problem, which is to introduce a small random offset to the target buffer duration for each round, referred to as periodic request scheduling, is an effective remedy. After trying to pin down the differences between my experiments and theirs, and running more simulations to try to match what they had done more closely, I was able to show that periodic request scheduling could lead to unfairness between competing ABR flows in some situations. According to my results, this seems to be sensitive to certain parameters, in particular the relationship between encoding rates, available bandwidth, and number of competing flows. I was not, however, able to conclusively show any positive effect of using random periodic request scheduling. There is scope for further work here to understand why I was not able to do this, and why the original problem only seems to manifest itself in some situations.

Where over buffering is to be avoided in client pull based applications, a periodic request scheduling approach is a good solution. However, developers should be aware of other problems that can be introduced by this technique, and take careful steps to be sure to avoid those problems. Scheduling strategies could also feasibly be considered for server push based applications, though this is less straightforward and I do not study the effects of such an approach in this work. In the next chapter, I bring the dissertation to a conclusion with a summary of my results and discussion of areas where my work has highlighted scope for further research.

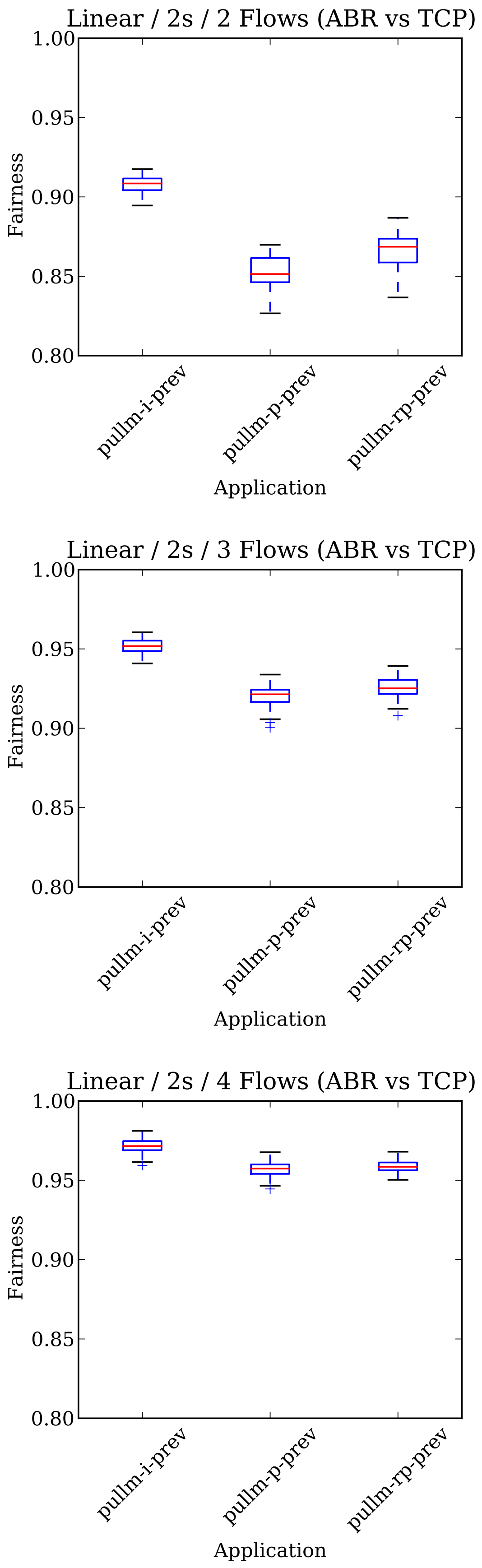


(a) Pull Multiple

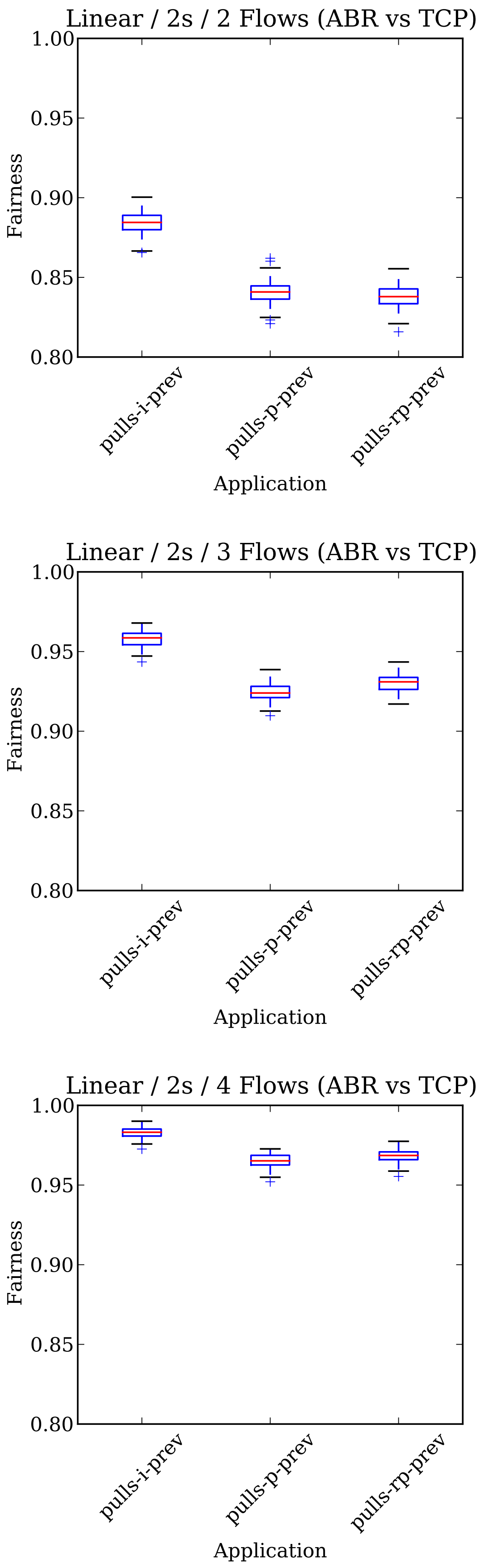


(b) Pull Single

Figure 7.1: Request Scheduling - Fairness (ABR vs ABR)

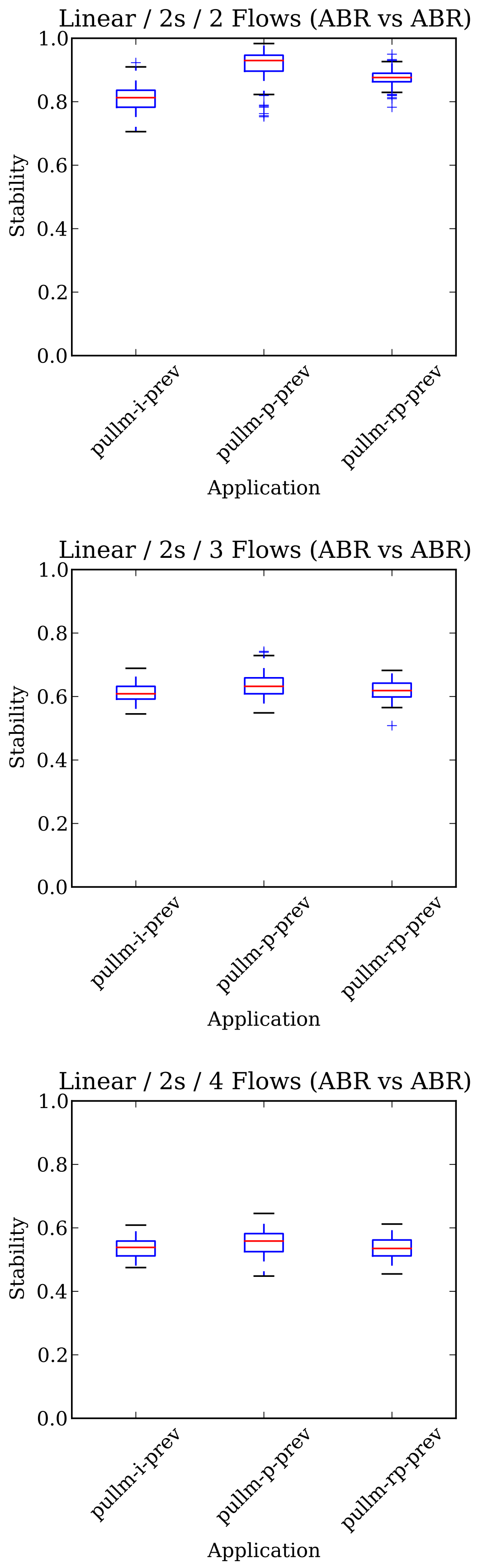


(a) Pull Multiple

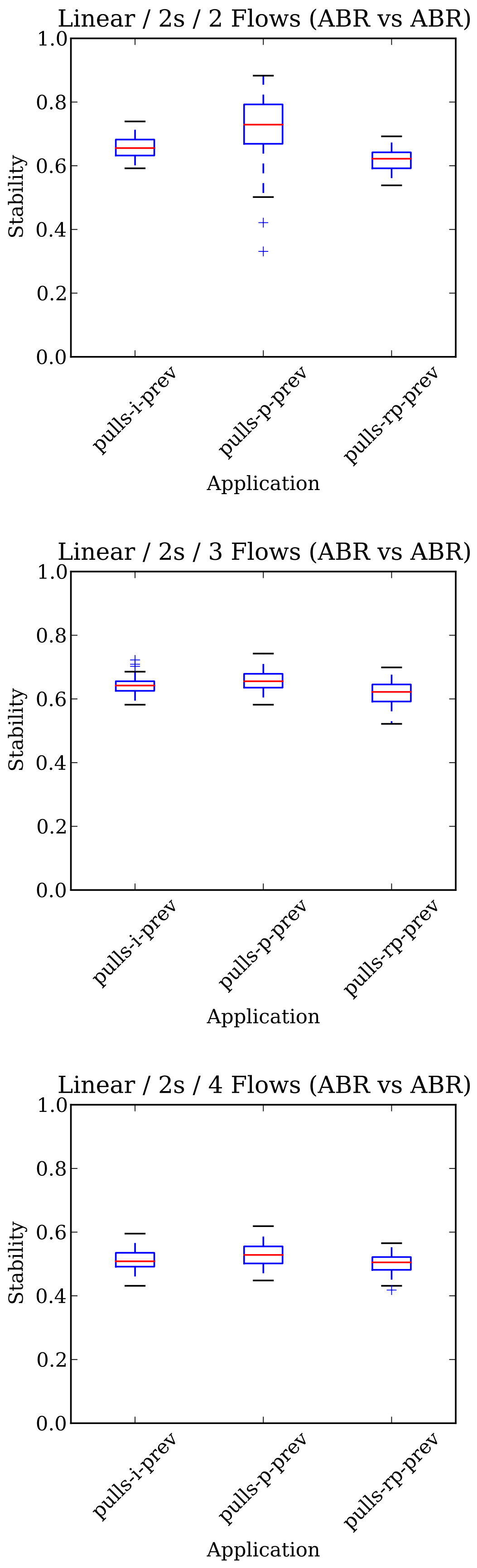


(b) Pull Single

Figure 7.2: Request Scheduling - Fairness (ABR vs TCP)

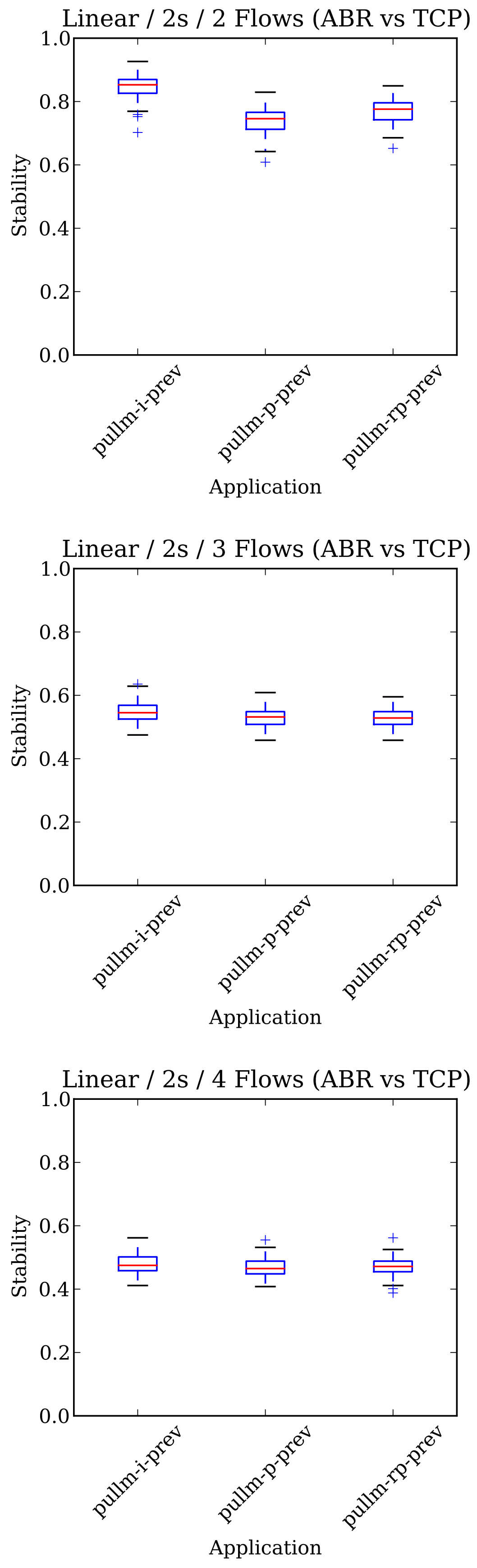


(a) Pull Multiple

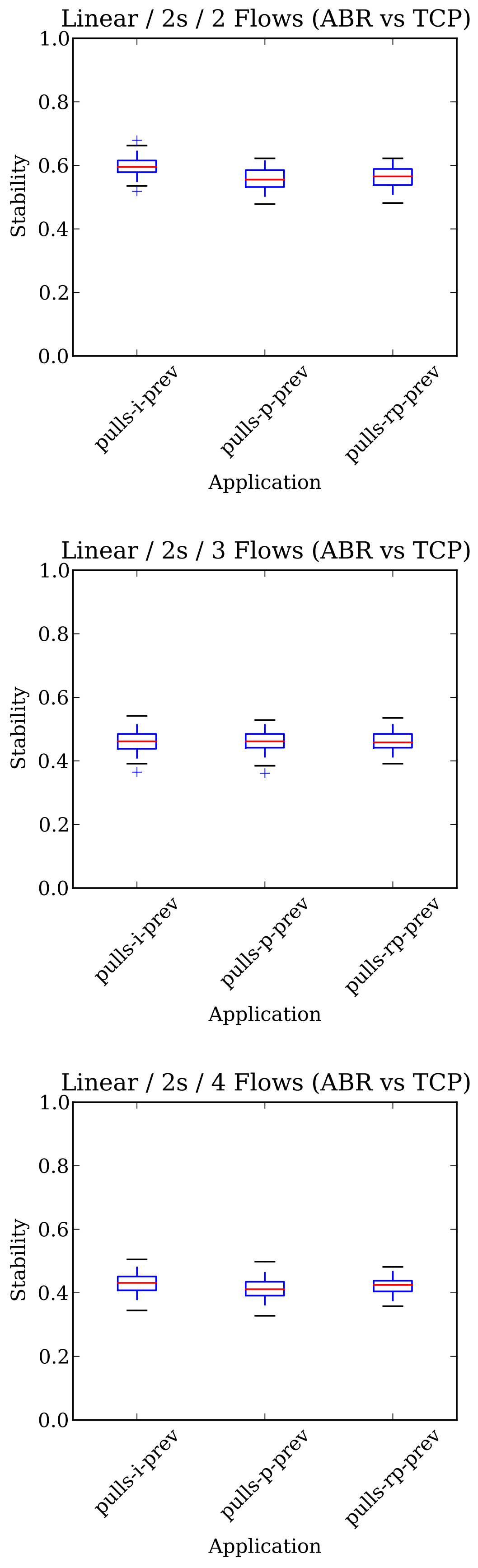


(b) Pull Single

Figure 7.3: Request Scheduling - Stability (ABR vs ABR)

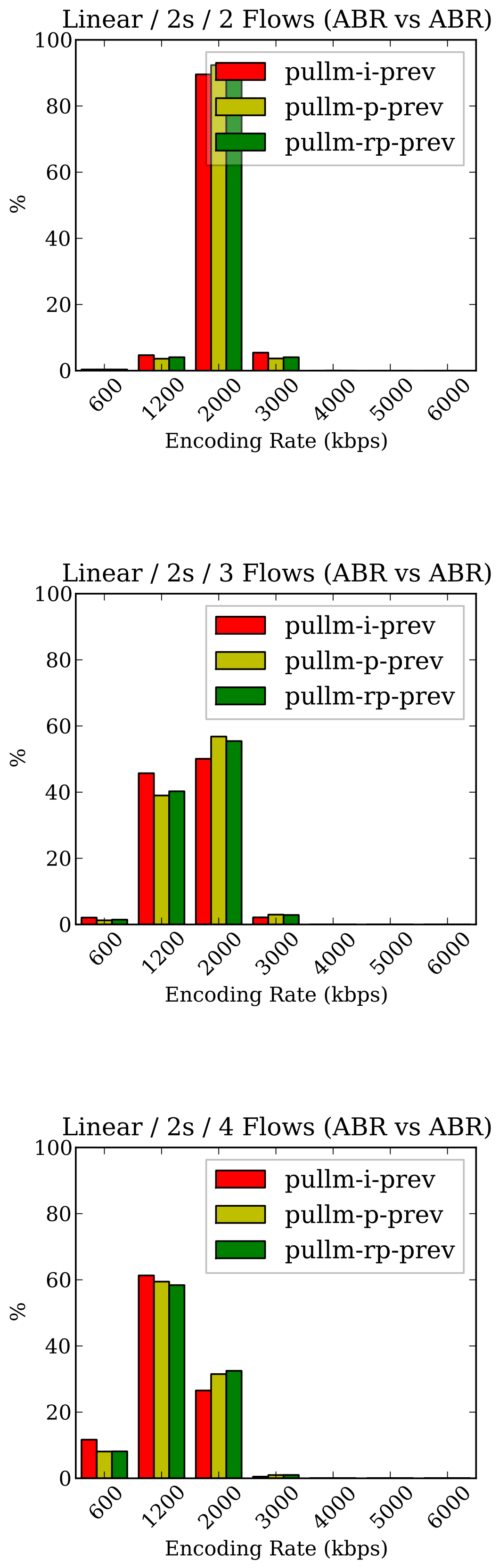


(a) Pull Multiple

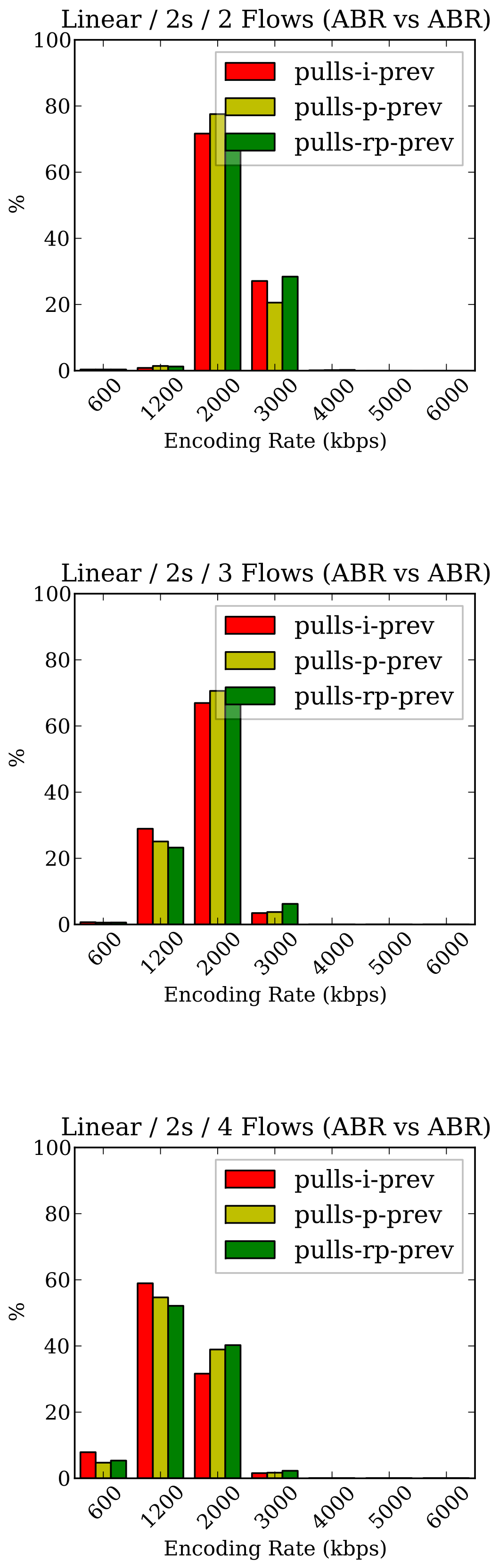


(b) Pull Single

Figure 7.4: Request Scheduling - Stability (ABR vs TCP)

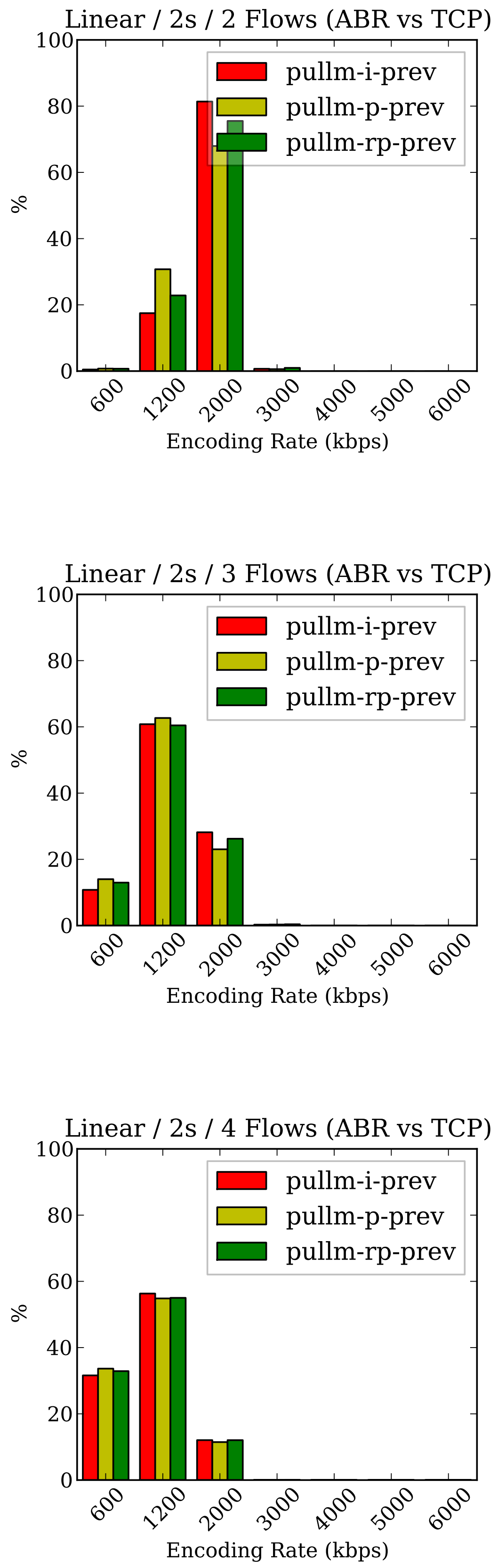


(a) Pull Multiple

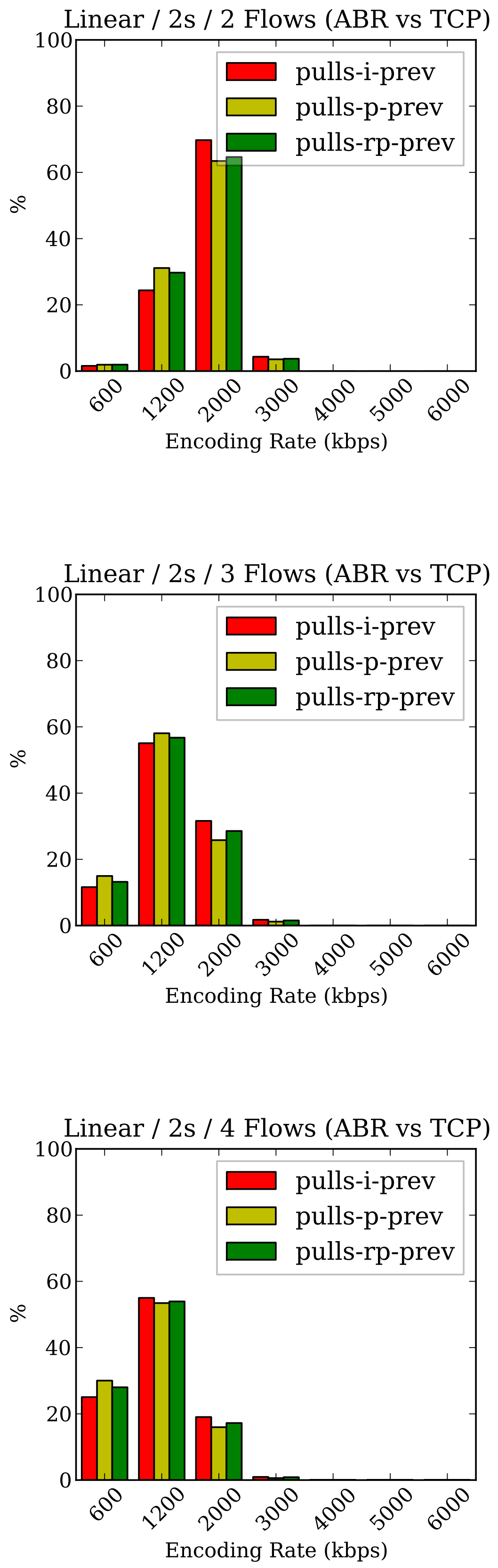


(b) Pull Single

Figure 7.5: Request Scheduling - Encoding Rate Distributions (ABR vs ABR)

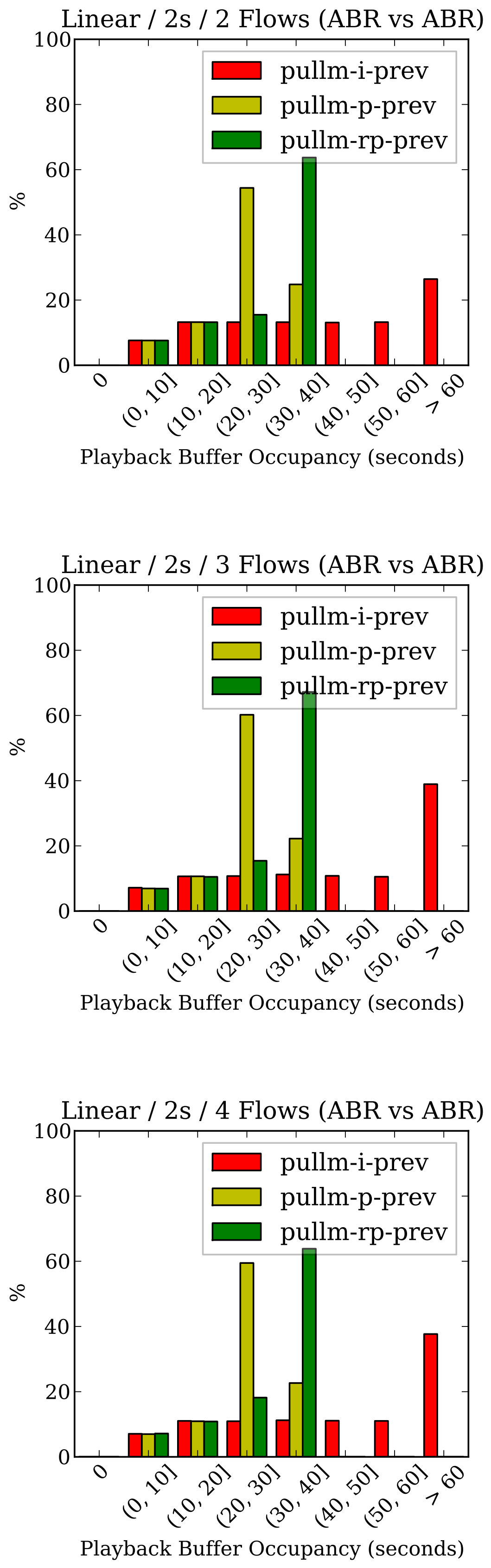


(a) Pull Multiple

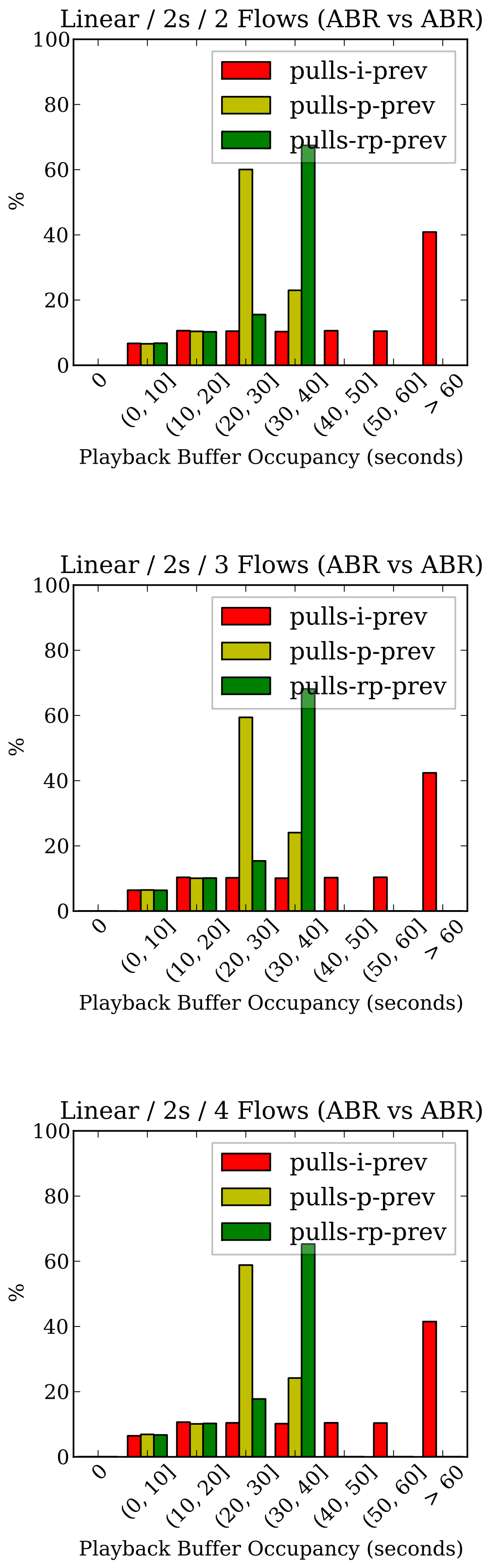


(b) Pull Single

Figure 7.6: Request Scheduling - Encoding Rate Distributions (ABR vs TCP)

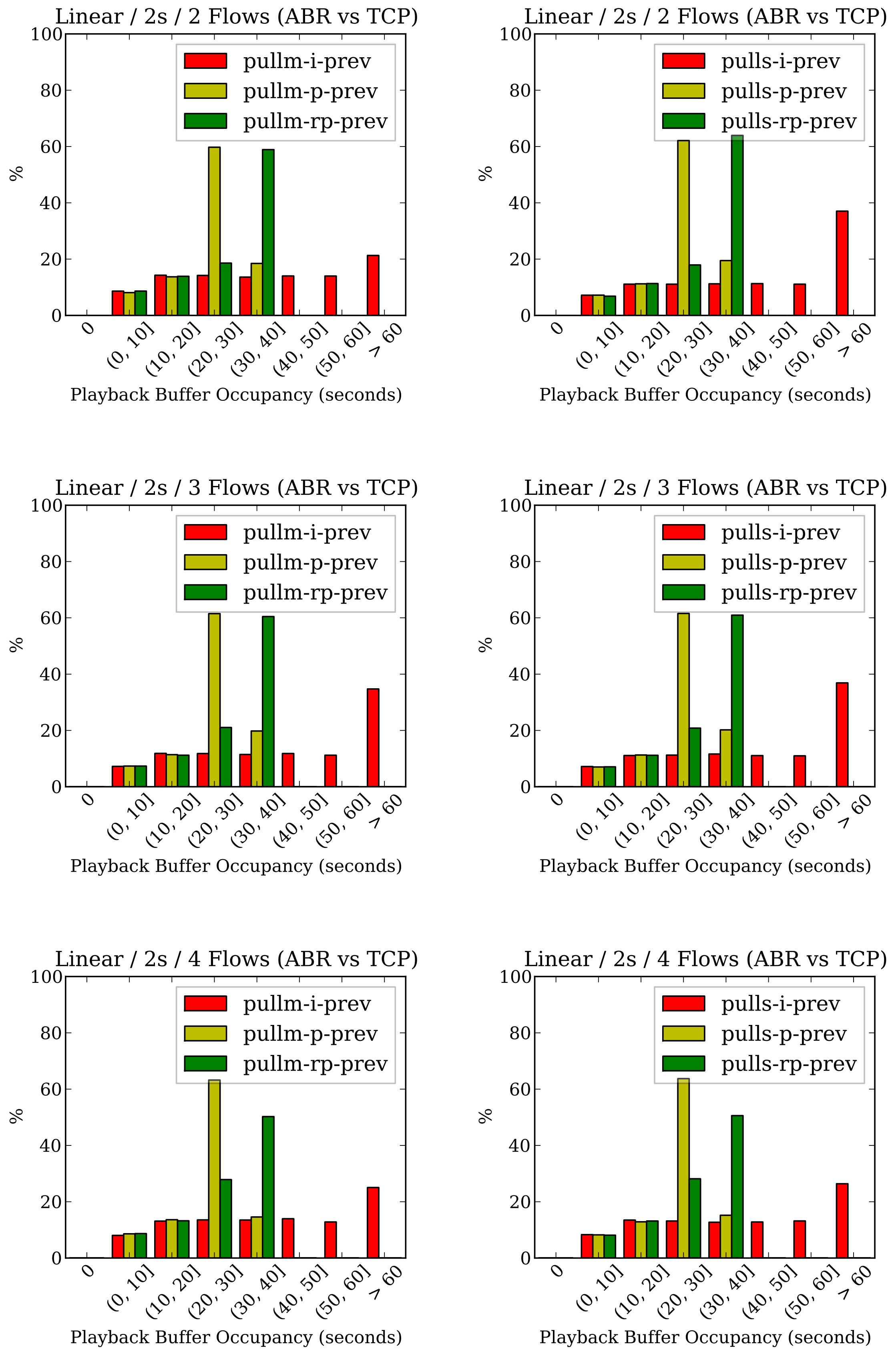


(a) Pull Multiple



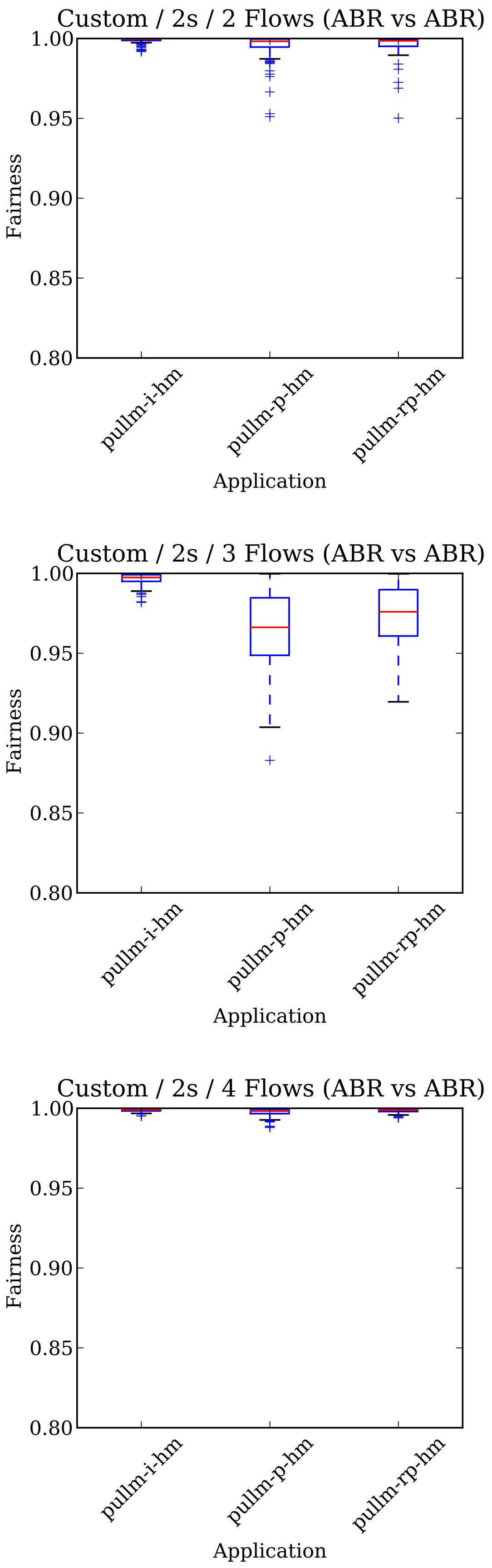
(b) Pull Single

Figure 7.7: Request Scheduling - Playback Buffer Occupancy Distributions (ABR vs ABR)

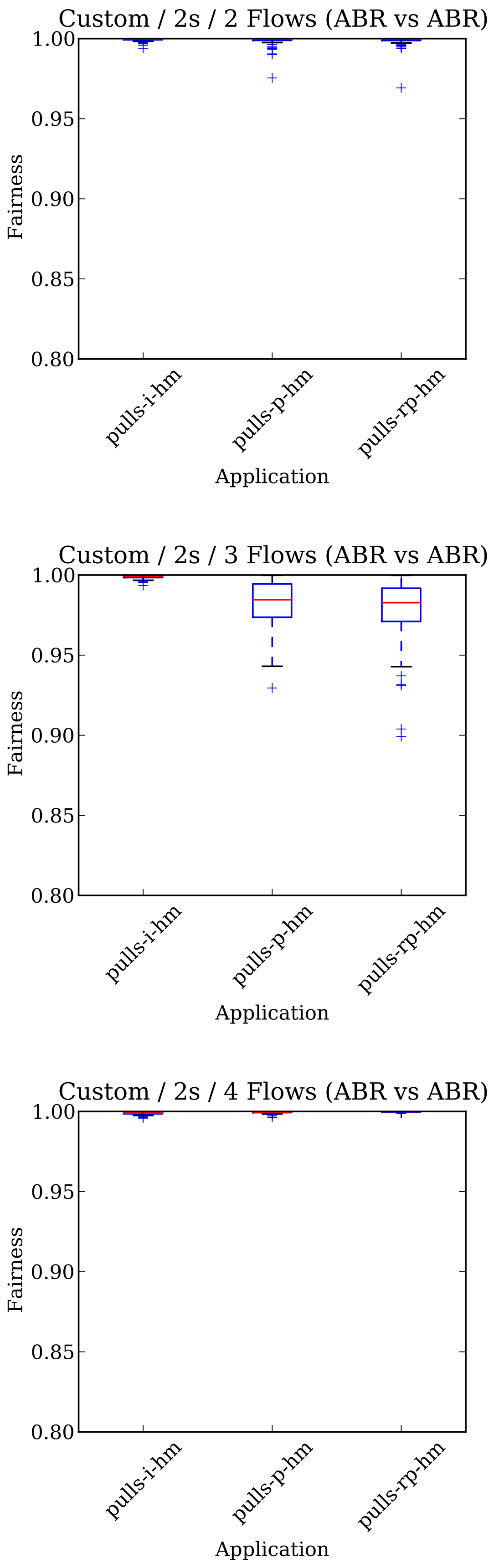


(a) Pull Multiple (b) Pull Single

Figure 7.8: Request Scheduling - Playback Buffer Occupancy Distributions (ABR vs TCP)

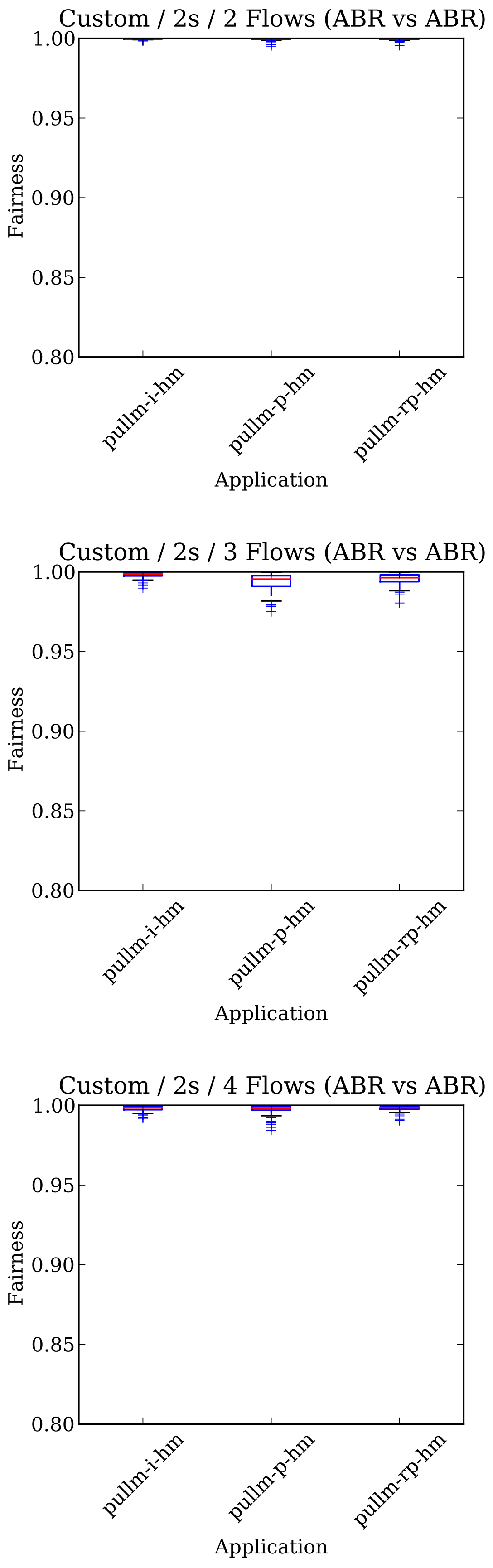


(a) Pull Multiple

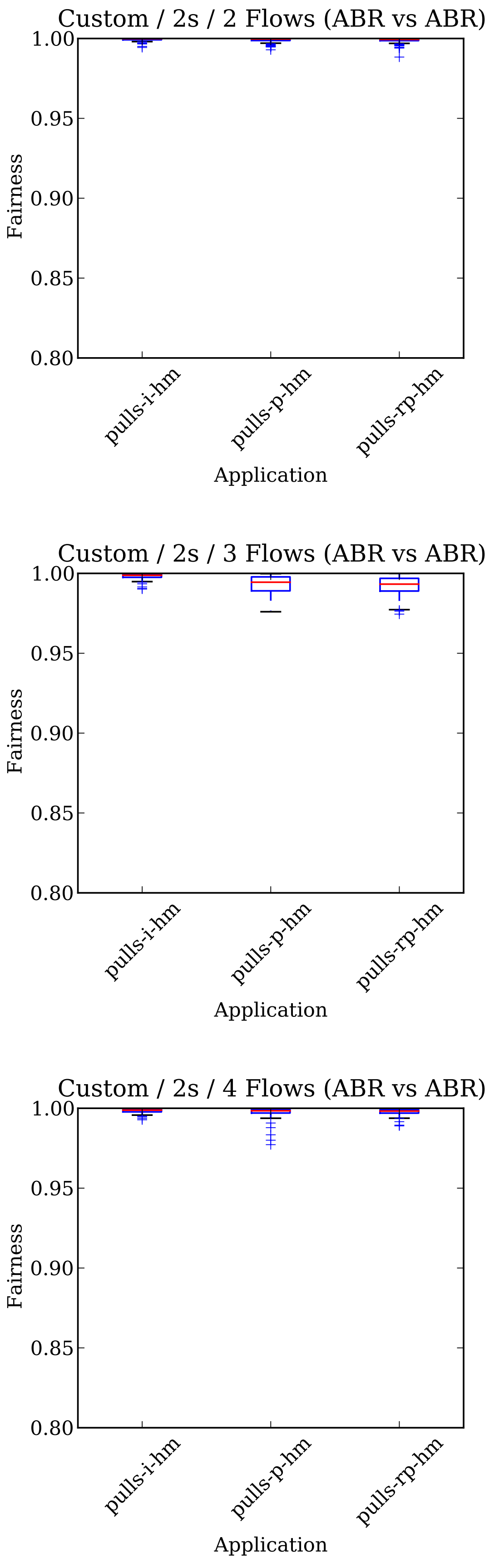


(b) Pull Single

Figure 7.9: Request Scheduling - Fairness (ABR vs ABR, parameters matching [1])

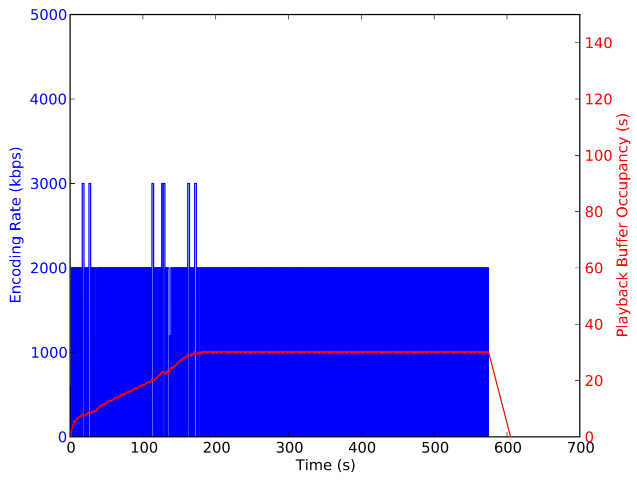


(a) Pull Multiple

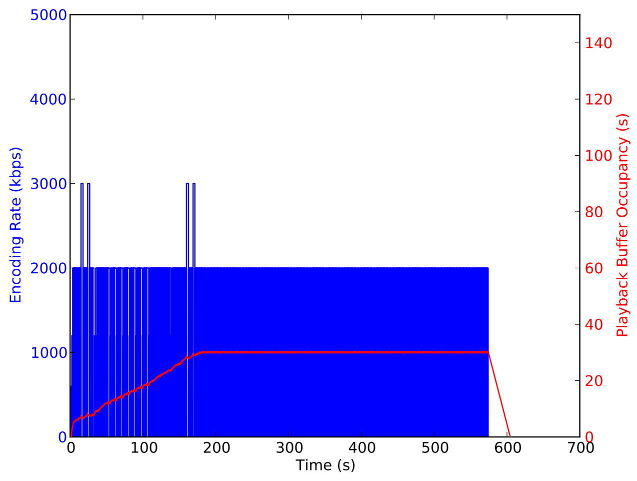


(b) Pull Single

Figure 7.10: Request Scheduling - Fairness (ABR vs ABR, parameters matching [1] with 4 mbps bottleneck)

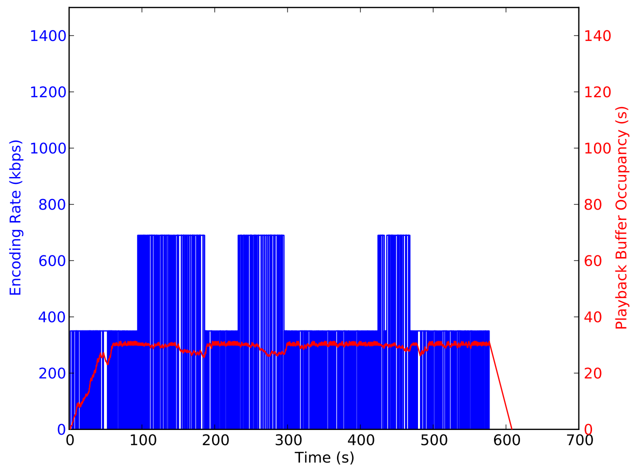


(a) ABR client 1

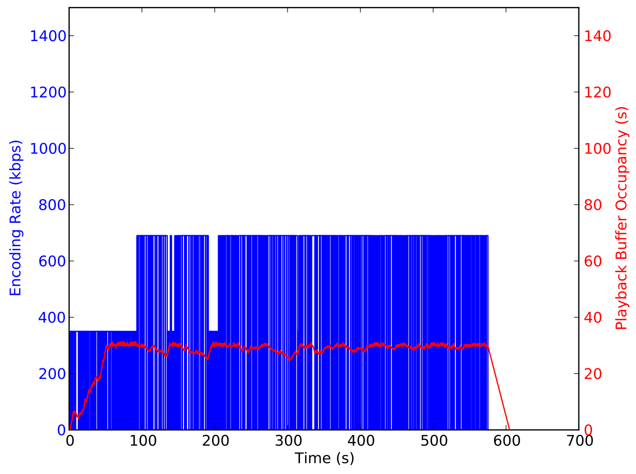


(b) ABR client 2

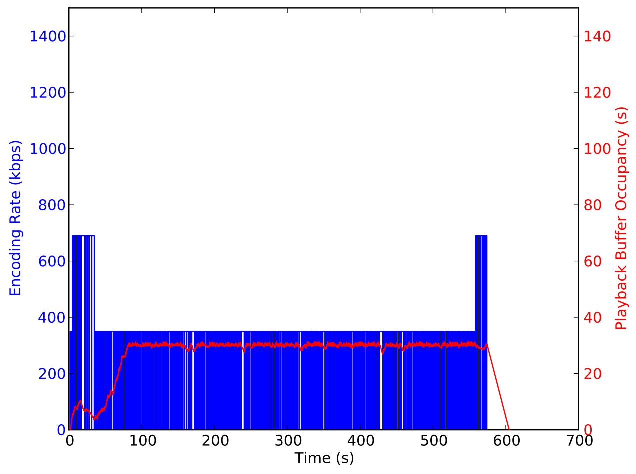
Figure 7.11: Encoding rates and playback buffer occupancy over time (original parameters, pull multiple, 2 ABR flows, 0 - 700s).



(a) ABR client 1

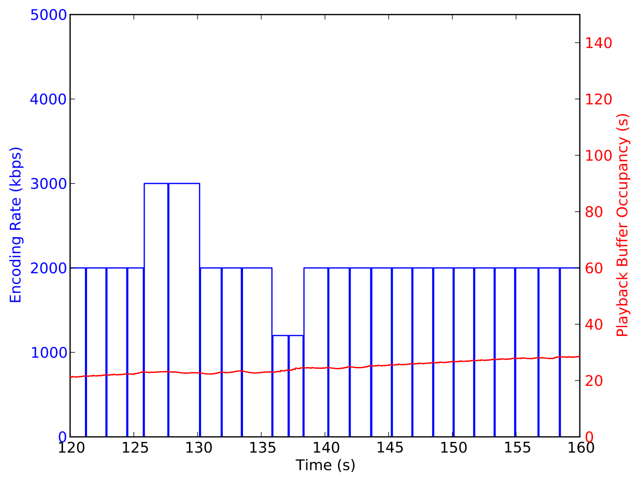


(b) ABR client 2

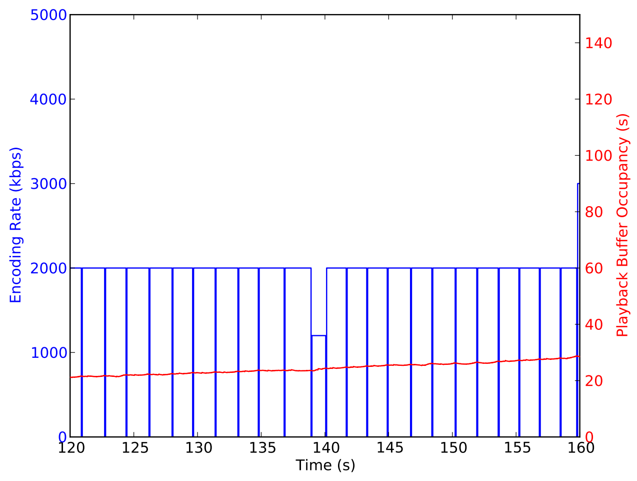


(c) ABR client 3

Figure 7.12: Encoding rates and playback buffer occupancy over time (parameters matching [1], pull multiple, 3 ABR flows, 0 - 700s).

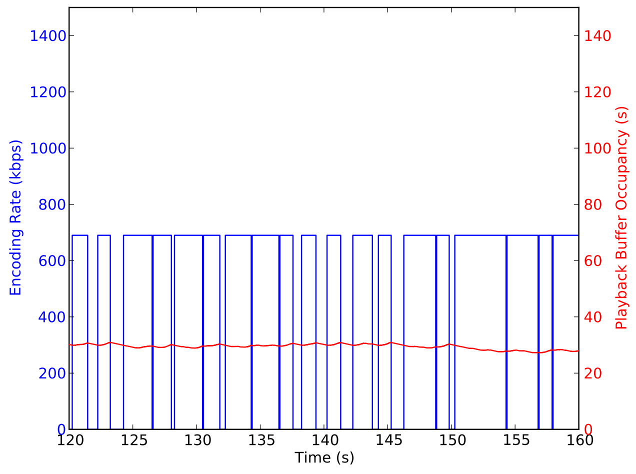


(a) ABR client 1

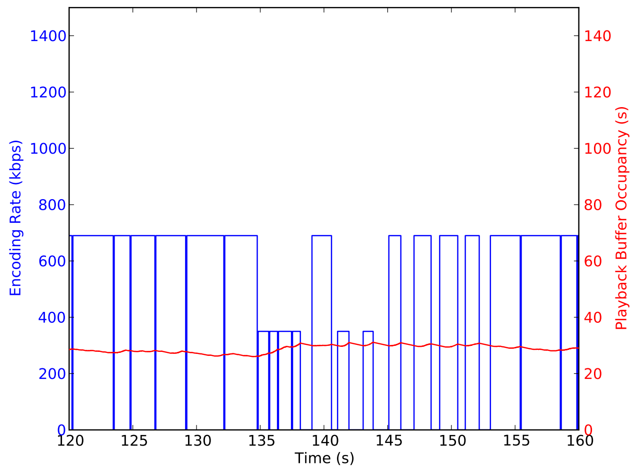


(b) ABR client 2

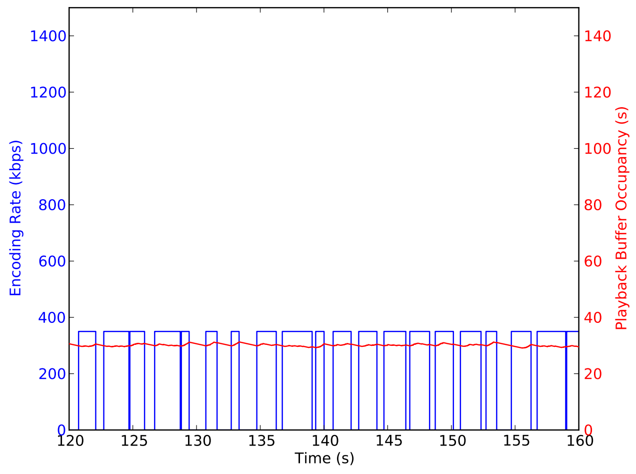
Figure 7.13: Encoding rates and playback buffer occupancy over time (original parameters, pull multiple, 2 ABR flows, 120 - 160s).



(a) ABR client 1



(b) ABR client 2



(c) ABR client 3

Figure 7.14: Encoding rates and playback buffer occupancy over time (parameters matching [1], pull multiple, 3 ABR flows, 120 - 160s).

Chapter 8

Conclusion

In this chapter, I present a summary of my findings and discuss their relevance to developers. I then suggest areas where there is scope for future research and continuation of my work, as well as areas where my methods could be improved. In the final section I end the dissertation with some concluding remarks and a summary of my contributions.

8.1 Summary of Findings

The main findings of my research fall broadly into three sets of related results. The first set of results are concerned with efficiency problems with client pull applications and how they can be tackled. These are summarised in the following list:

- The on-off nature of client pull flows results in poor bandwidth efficiency, and makes it difficult for those flows to achieve high throughput and encoding rates, especially when competing with other traffic.
- Server push applications, which maintain a continuous TCP flow between client and server, do not exhibit these problems and behave much like any other long lived TCP flow.
- Increasing the length of chunk duration largely mitigates the efficiency problems in client pull applications.
- Pull selective applications perform as well as server push applications, but scale like client pull applications.

In chapter 4 I observed efficiency problems for client pull applications with a 2 second chunk duration, using both multiple and single TCP connections. There was noticeable discrimination against client pull applications competing against continuous TCP flows, and client pull applications achieved lower average rates than server push and pull selective counterparts in the same scenarios. The inefficiency, highlighted and discussed in previous chapters, is caused by frequent idle periods at chunk boundaries and the on-off nature of client pull flows. The same behaviour is observed in [3], much of which is devoted to investigating the underlying cause of the problem. They also conclude that idle periods at chunk boundaries are at the root of the problem. Transfers for short chunks never have time to build up to and maintain a high throughput before the transfer is complete. When the next chunk starts, TCP's congestion window will have been reset, either because the connection was left idle or because a new connection was created. Meanwhile any competing flows will have continued to fill the queueing buffer at the bottleneck router so that the new flow is likely to experience increased packet loss, which only makes it more difficult to escape slow start and reach the steady state phase.

Two effective ways to counteract this problem are to increase the chunk duration, or to avoid using the client pull approach altogether. In chapter 5 I showed that with a 10 second chunk duration the difference between client pull and server push applications is much smaller. This is because the longer transfers have more time to increase throughput and reach the steady state. From chapter 4 it was also clear that the problem did not exist with server push or pull selective applications, since these maintain a continuous TCP flow between client and server with no idle periods at chunk boundaries. The downside of using server push is that it is a less scalable approach, however, the hybrid pull selective approach scales like client pull, since adaptation logic is handled by the client, but performs like server push on the network since it is able to maintain a continuous TCP flow.

The next set of results relate to encoding rate stability, and the difficulty of choosing a suitable long term encoding rate:

- ABR applications are liable to suffering from poor encoding rate stability, unless steps are taken to address the issue.
- Having fewer available encoding rates naturally makes for more stable streams, with fewer rate switches.
- Dampening bandwidth estimates by using an averaging function to filter out short term fluctuations can significantly improve encoding rate stability.

In my benchmark simulations, applications using all four transfer mechanisms exhibited poor encoding rate stability. This has also been observed in a number of recent studies [1, 3, 12]. Instability seems to arise from the difficulty of picking a suitable encoding rate, that can be sustained for a reasonable period of time, above the HTTP layer. At this level, TCP hides much of the complexity of what is really happening on the network. Observed rates fluctuate over short time spans and make it difficult for an application to accurately estimate the bandwidth it has available. In chapter 5 I observed that the problem was less severe when using the smaller set of exponentially spaced rates. Although it is not clear what difference the spacing makes, when there are fewer rates available the system is less likely to switch rate due to small throughput fluctuations. In chapter 6 I showed that filtering bandwidth estimates, by applying an averaging function to recently observed transfer rates, will also help significantly to improve encoding rate stability. Short term fluctuations are dampened and the bandwidth estimate won't change by much unless changes in network conditions persist. I demonstrated this using both the harmonic mean of 20 recent transfer rates and an exponentially weighted moving average of all transfer rates. Using an averaging function like this is not a new idea, and is also suggested in both [3] and [1]. In [1], the authors also show with their experiments that using the harmonic mean improves encoding rate stability.

In chapter 7 I investigated different approaches to request scheduling for client pull systems, and compared the obvious immediate scheduling approach with two periodic approaches intended to avoid the problems of over buffering. The two main outcomes of this chapter can be summarised as follows:

- Periodic request scheduling can be used to avoid over buffering and the problems associated with it.
- In some circumstances, periodic request scheduling can result in unfair allocation of resources between competing ABR flows.

Service providers may wish to avoid over buffering, since it can be both wasteful of bandwidth and achieve suboptimal results. If a user leaves the stream prematurely then over buffered content will have been wasted, and over buffering also runs the risk of downloading chunks hastily at a lower rate when conditions may improve later. To avoid over buffering, periodic request scheduling times requests for new chunks to be sent just as the buffer occupancy drops below a target duration. This allows the system to maintain a safety net of buffered content, which can absorb the affects of any sudden changes on the network, without having to buffer content indefinitely. My experiments described in Chapter 7 showed that periodic request scheduling can successfully avoid over buffering without introducing playback interruptions. However, my original results from this chapter were in conflict with the results of experiments from [1], which showed that periodic request scheduling can lead to fairness issues between competing ABR flows. After tweaking my parameters to match the ones in experiments from [1] more closely, I was able to reproduce their results, but changing the bottleneck bandwidth from 3 mbps to 4 mbps was enough to make the effect disappear again. This suggests that the problems of unfair allocation of resources between competing ABR flows using periodic request scheduling is sensitive to the relationship between certain parameters. In [1] the authors also showed that introducing a small element of random jitter to the periodic request scheduling strategy was enough to avoid the repetitive patterns and synchronisation effects that cause this problem. I was unable to reproduce that result conclusively, in part due to the fact that I could only produce the original problem with one set of parameters.

Some of my results are new contributions to the field and some help to strengthen conclusions that other studies have also reached. The comparison between server push and client pull approaches, for example, is a novel contribution, as is the evaluation of the pull selective approach. The effects of changing chunk duration and number of available encoding rates have not been widely studied,though some recent studies have touched on these less directly. For example, [3] considers chunk duration briefly at one point. On the other hand, my results highlighting instability, and those demonstrating that filtering bandwidth estimates can improve stability, are not new, but instead serve both to help validate my simulations and support the conclusions already reached by researchers. Similarly, my investigation of request scheduling parameters was not original work, but did help to support findings by researchers whilst also adding an extra dimension and showing that the picture is incomplete.

8.2 Recommendations to Developers

I have said from the beginning that my results are trying to shed light on some poorly understood tradeoffs in ABR streaming. The intended outcome was not necessarily to find the best way to implement an ABR system, but to help developers make more informed decisions for themselves when it comes to implementing an ABR system. In this section I will discuss the main tradeoffs that I have investigated, and suggest how I think developers can interpret my results usefully.

The first problem that developers will need to consider is which transfer mechanism to adopt, and how to address the inefficiency of the client pull approach if they take that option. Choosing between client pull and server push and pull selective will involve considering more than just how well they perform on the network. There are other fundamental tradeoffs involved in this decision, and developers must consider practical issues such as ease of implementation and deployment. Server push and pull selective applications do not suffer from the same inefficiency problems as client pull applications, and while server push does not scale very well, pull selective does scale well and offers an interesting option that is worth careful consideration. Both server push and pull selective, however, are more difficult to implement and less flexible than a client pull based system. Server push and pull selective each require a modified server, for example, and make it more difficult for a client to switch host easily. On top of this, it is more difficult to implement a periodic scheduling strategy with a server push based approach, since it requires having the server estimate the client's playback state. This makes it more difficult to avoid over buffering, which is discussed later in this section, with a server push based system. Other modifications that require playback state information, such as always selecting the lowest rate when there is less than a certain duration of content buffered locally for example, are also more difficult to implement with server push for the same reason.

If developers choose to use client pull, then a second option for tackling the inefficiency is to select the chunk duration carefully. In my experiments, changing the chunk duration from 2 seconds to 10 seconds was enough to make a significant difference. There is, however, a danger of making the chunk duration too long and hampering the system's ability to react and avoid congestion. It is not clear where the boundary lies here, and the answer will depend on things like content buffering and request scheduling strategies, but most commercial systems seem to use a chunk duration of between 2 and 10 seconds [1, 3].

The second important issue that developers should be aware of is encoding rate instability. This problem manifested with every application in my initial experiments, and has also been observed in a number of other studies recently. The first thing to consider here is bandwidth estimation and using an averaging function to dampen the estimate and filter out short term fluctuations. This was shown to work in my experiments, and has also been suggested or demonstrated to work in other studies. There is also a concern here that too much dampening will make it difficult for the application to learn quickly of any genuine changes in the network it needs to react to. Secondly, developers should consider carefully which encoding rates to make available for ABR streaming. This will depend on a number of factors, including the content itself and intended target device, but my simulations indicate that more is not better. Fewer encoding rates means fewer opportunities for rate switches, though providers should be careful to ensure that enough rates are made available to make adaptation effective. After deciding on the desired range of rates, I would suggest that just one or two in the middle should be enough to cover most scenarios. This will also means less space is taken up on the disk, and less time spent encoding each file that is uploaded to a site like YouTube, for example.

Finally, developers wishing to avoid over buffering will want to select a suitable request scheduling strategy, possibly in conjunction with a client pull based system. Again, this is not just an engineering decision, and will depend on the intended use case of the system. For example, for a movie streaming website such as Netflix, it can probably be assumed that if the user watches the first 5 minutes then they will continues to watch the rest of the movie in most cases. For a site like YouTube, on the other hand, the user's attention span is likely to be much lower and the provider may wish to consider using a periodic request scheduling strategy to avoid over buffering and wasting bandwidth. Where periodic request scheduling is used, developers should be aware of the potential for this to cause synchronisation problems and lead to unfairness between competing ABR flows. If problems like this do occur, then introducing a small element of random jitter to the scheduling strategy looks to be a simple counter measure.

Hopefully this will be a useful set of guidelines to help developers who are making these choices. In the next section I will discuss scope for future work and areas where my methods could be improved.

8.3 Scope for Future Work

This work has highlighted a number of areas where there is scope for further research. There is also scope to directly continue and build on my work in some areas, as well as to improve on the methods I used. I will start here by being critical of my own work, and suggesting things that I would change were I to continue with the simulation approach. The first thing I would do, were I to continue this work, is spend time improving the competition traffic models. First of all, more realistic competing traffic scenarios are needed in order make my results more relevant and valid. Currently I have two competing traffic models, long lived TCP flows and other ABR flows. Although these are a good starting point, they both represent fairly artificial situations, which are not typical of those found when users stream videos over the web in real life. A third traffic model could be constructed by analysing real world network traces, and using this would add a lot of credibility to my experiments by demonstrating that the results also appear in real life scenarios. On top of this, I would spend time improving the randomised background traffic model that I use in a similar way, and in particular pay more attention to how the parameters of this traffic affect the behaviour of simulations.

A second area where I could improve my results is by using more complicated scenarios, and spending more time inspecting time series graphs to better understand what is happening. For example, it would be useful to be able to set up experiments where a competing flow starts after an ABR flow has become established, then carefully analyse in detail what happens immediately before and after the new flow starts. I could also spend time looking at the relationship between queue occupancy on the bottleneck router, congestion window behaviour, and throughput. Finally, there is scope to extend my results by considering the effect of changing parameters on the network which have remained static in my work. In particular, I would want to make my results more robust by showing that they can be replicated with different end-to-end delays and bottleneck bandwidths. It would also be interesting to study how buffer bloat is affecting the behaviours I observe and compare simulations using under buffered and over buffered queues, or to play with different TCP models and tweak the initial congestion window. Since TCP's behaviour was responsible for both the stability and inefficiency problems that I observed, it would make sense to check if these can be solved simply by using a newer TCP model, or tweaking the parameters of the model currently in use. For example, increasing the initial congestion window from 1 packet to 10 packets, as discussed in [40], would benefit short transfers and may help client pull flows with a short chunk duration. Using simulation software makes it possible to study things such as TCP parameters, which may not be easy to change on a real network.

Areas where my work has introduced scope for further research include the behaviours of server push and pull selective systems, the cause of unfairness between competing ABR flows using periodic request scheduling, and the effects of media formatting parameters. Server push and pull selective systems have not been widely studied in other works, and their is definite scope to look at these systems more closely. For example, although it can be inferred that server push does not scale as well as client pull, due to the extra processing required on the server for each client, my simulations did not and were not expected to capture this. Ns-3 does not model application running times, and my simulations do not stress the servers with multiple concurrent clients. It would, however, be useful to be able to observe and quantify these differences. For a client pull system using a single long lived TCP connection, the server already keeps per client state, but it is difficult to say how much difference the extra state and server side calculations required for server push flows really makes. This could be investigated with a more sophisticated simulator, or by studying a real life system running on lab machines.

In Chapter 5 I showed how tweaking media formatting parameters can benefit the design of an ABR system, but that picture is still incomplete and there are unanswered questions remaining. It's not clear what is a safe limit for a chunk duration that doesn't endanger the system ability to react to congestion. It would also be useful to investigate further how the range and spacing of encoding rates affects a system's performance. Running experiments using intermediate chunk durations between 2 and 10 seconds would help to understand the effect at a finer level of granularity. Again in Chapter 5, although I suggested that using fewer rates resulted in more stable streams, it was unclear what effect if any the difference between linearly and exponentially spaced rates had. Repeating the experiments from this chapter using a set of four linearly spaced rates, for example, would show more clearly whether having fewer rates or having different spacing between rates was responsible for the effect on stability. In Chapter 6, I demonstrated that introducing an averaging function to bandwidth estimates can improve stability, but further work is needed to investigate the effects of both the α parameter in Equation 6.1, for EWMA, and the number of samples averaged in Equation 6.2, for harmonic mean. Finally, my results from Chapter 7 suggested that the problems with periodic request scheduling are sensitive to the relationship between certain parameters including bottleneck bandwidth, number of competing flows, and available encoding rates. There is scope to investigate this further to try to understand why that is the case and how common the problem is.

8.4 Conclusion

In this project I set out to investigate the effects of various parameters on ABR video streaming systems, in order to shed light on areas that are well understood and guide developers faced with making difficult decisions when implementing ABR systems. I used network simulation software to model and compare different application models, looking at the difference between client pull and server push approaches, the effect of media formatting parameter such as chunk duration and number of available encoding rates, and the implementation of various system components that implement ABR logic.

In the end I found that client pull systems can be inefficient and find it difficult to achieve their fair share of available bandwidth, due to the effect of frequent idle periods on TCP's behaviour. This result mirrors findings from a number of previous studies. I also found that two effective approaches to countering this problem are to use a longer chunk duration, or to avoid client pull altogether and implement a server push based system. Increasing the chunk duration from 2 seconds to 10 seconds gives the short transfers for each individual chunk a better chance to build up and maintain a high throughput in a client pull based system. Server push systems maintain a continuous TCP flow between client and server, so behave more efficiently than client pull systems, but the server must keep extra per client state and perform ABR calculations for each client, meaning that this approach is also less scalable than client pull. As a novel contribution of this work, I found that pull selective applications, which which are a hybrid of client pull and server push, putting the responsibility for ABR calculations back on the client whilst maintaining a constant TCP flow between client and server, offer another alternative to developers wishing to avoid the efficiency problems of client pull applications. Pull selective systems perform as well as server push and scale as well as client pull.

I also found that ABR systems in general are vulnerable to encoding rate instability, unless measures are taken to address the issue. Again, this is a problem that other studies have found, and arises from TCP behaviours that make it difficult to select and maintain a suitable encoding rate using application layer transfer rate estimates. Effective counter measures include carefully choosing what rates are available in order to make sure there are not too many, and using an averaging function on bandwidth estimates to filter out short term fluctuations. Finally, I also found that periodic request scheduling can avoid over buffering in client pull systems, but can also lead to unfair sharing of resources between competing ABR flows.

In this chapter I summarised these findings and their relevance to developers, before setting the stage for further research or a direct continuation of this work. My project has uncovered new results, validated and reproduced results from other studies, and highlighted areas where more work is needed to understand conclusively what is happening. Web-based, on-demand adaptive bitrate video streaming systems have become very popular with content providers in recent years, and do not seem to be going away for the foreseeable future. Hopefully the findings of this work will help developers to make better systems for people to use.

Bibliography

[1] J. Jiang, V. Sekar, and H. Zhang, “Improving fairness, efficiency, and stability in HTTP-based adaptive video streaming with FESTIVE,” in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, 2012.

[2] Cisco Visual Networking Index: Forecast and Methodology, 2012-2017. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360_ns827_Networking_Solutions_White_Paper.html. [accessed 21-December-2013].

[3] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari, “Confused, timid, and unstable: picking a video streaming rate is hard,” in *Proceedings of the 2012 ACM Conference on Internet Measurement*, 2012.

[4] S. Akhshabi, A. C. Begen, and C. Dovrolis, “An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP,” in *Proceedings of the 2nd Annual ACM Conference on Multimedia Systems*, 2011.

[5] Akhshabi, Anantakrishnan, Begen, and Dovrolis, “What happens when HTTP adaptive streaming players compete for bandwidth?” in *Proceedings of The 22nd ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2012.

[6] R. Houdaille and S. Gouache, “Shaping HTTP adaptive streams for a better user experience,” in *Proceedings of the 3rd Annual ACM Conference on Multimedia Systems*, 2012.

[7] BT Vision. http://www.btvision.bt.com/. [accessed 21-December-2013].

[8] Virgin Media’s TiVo service. http://store.virginmedia.com/digital-tv/set-top-boxes/tivo.html. [accessed 21-December-2013].

[9] B. Clouston and B. Moore, “RFC 2475 - An architecture for differentiated services,” http://www.ietf.org/rfc/rfc2475.txt.

[10] M. Allman, V. Paxson, and E. Blanton, "RFC 5681 - TCP Congestion Control," http://www.ietf.org/rfc/rfc5681.txt.

[11] T. Stockhammer, "Dynamic adaptive streaming over HTTP - standards and design principles," in *Proceedings of the 2nd Annual ACM Conference on Multimedia Systems*, 2011.

[12] G. Tian and Y. Liu, "Towards agile and smooth video adaptation in dynamic HTTP streaming," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, 2012.

[13] Netflix. http://www.netflix.com. [accessed 21-December-2013].

[14] Hulu. http://www.hulu.com. [accessed 21-December-2013].

[15] Vudu. http://www.vudu.com. [accessed 21-December-2013].

[16] YouTube. http://www.youtube.com. [accessed 21-December-2013].

[17] BBC iPlayer. http://www.bbc.co.uk/iplayer. [accessed 21-December-2013].

[18] Smooth Streaming Experience. http://www.test.org/doe/. [accessed 21-December-2013].

[19] Smooth Streaming Protocol. http://msdn.microsoft.com/en-us/library/ff469518.aspx. [accessed 21-December-2013].

[20] Apple Quicktime. http://www.apple.com/quicktime. [accessed 21-December-2013].

[21] Adobe HTTP Dynamic Streaming. http://www.adobe.com/products/hds-dynamic-streaming.html. [accessed 21-December-2013].

[22] V. Paxson and S. Floyd, "Why we don't know how to simulate the Internet," in *Proceedings of the 29th Conference on Winter Simulation*, 1997.

[23] M. Ammar, "Why We STILL Don't Know How to Simulate Networks," in *Proceedings of the 13th IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2005.

[24] ns-3 Network Simulator. http://www.nsnam.org. [accessed 21-December-2013].

[25] matplotlib Python graph plotting library. http://www.matplotlib.org. [accessed 21-December-2013].

[26] J. Mogul, L. Brakmo, D. E. Lowell, D. Subhraveti, and J. Moore, "Unveiling the Transport," *SIGCOMM Computer Communications Review*, vol. 34, 2004.

[27] D. D. Clark and D. L. Tennenhouse, “Architectural Considerations for a New Generation of Protocols,” *SIGCOMM Computer Communications Review*, vol. 20, 1990.

[28] ITEC - Dynamic Adaptive Streaming Over HTTP (Dataset). http://www-itec.uni-klu.ac.at/dash/?page_id=207. [accessed 21-December-2013].

[29] Ofcom - UK fixed-line broadband performance, November 2012. http://stakeholders.ofcom.org.uk/market-data-research/other/telecoms-research/broadband-speeds/broadband-speeds-nov2012. [accessed 21-December-2013].

[30] V. Jacobson and R. Braden, “RFC 1072 - TCP Extensions for Long-Delay Paths,” http://www.ietf.org/rfc/rfc1072.txt.

[31] Wikipedia article - Tail drop queue. https://en.wikipedia.org/wiki/Tail_drop. [accessed 21-December-2013].

[32] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow, “RFC 2018 - TCP Selective Acknowledgment Options,” http://www.ietf.org/rfc/rfc2018.txt.

[33] J. Nagle, “RFC 896 - Congestion Control in IP/TCP Internetworks,” http://www.ietf.org/rfc/rfc896.txt.

[34] S. Floyd, T. Henderson, and A. Gurtov, “RFC 3782 - The NewReno Modification to TCP’s Fast Recovery Algorithm,” http://www.ietf.org/rfc/rfc3782.txt.

[35] TCP Models in ns-3. http://www.nsnam.org/docs/release/3.11/models/html/tcp.html. [accessed 21-December-2013].

[36] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, “A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems,” Tech. Rep., 1984.

[37] M. Handley, J. Padhye, and S. Floyd, “RFC 2861 - TCP Congestion Window Validation,” http://www.ietf.org/rfc/rfc2861.txt.

[38] N. Cranley, P. Perry, and L. Murphy, “User perception of adapting video quality,” *International Journal of Human-Computer Studies*, vol. 64, 2006.

[39] Wikipedia article - TCP global synchronization. https://en.wikipedia.org/wiki/TCP_global_synchronization. [accessed 21-December-2013].

[40] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin, “An argument for increasing TCP’s initial congestion window,” *SIGCOMM Computer Communications Review*, vol. 40, 2010.