A Framework for Quality-Adaptive Media Streaming: Encode Once — Stream Anywhere

Charles Krasic
B.Math., University of Waterloo, 1992
M.Math., University of Waterloo, 1996

A dissertation submitted to the faculty of the
OGI School of Science & Engineering
at Oregon Health & Science University
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

February 2004

The dissertation "A Framework for Quality-Adaptive Media Streaming" by Charles Krasic has been examined and approved by the following Examination Committee:

Jonathan Walpole
Professor
OGI School of Science and Engineering
Thesis Research Adviser

Wu-chi Feng
Associate Professor
OGI School of Science and Engineering

Mark Jones
Associate Professor
OGI School of Science and Engineering

Thomas Plagemann
Professor
University of Oslo

Dedication

To my parents, for their love and support always.

Acknowledgements

There are many colleagues and collaborators whose influence has helped me to form the research ideas in this dissertation, and to them I am deeply grateful. I was very fortunate to have Jonathan Walpole as my adviser. His tolerance, guidance, and strong support over the years have been instrumental to the successful completion of this work. He was a generous mentor outside of work also, where I was the benefactor of his great expertise and experience as a competitive cyclist and skier. I know I was not alone in this. Much like his research ideas, Jon's approach to his many hobbies are well known as being contagious to many of the people around him. His balance between profession and lifestyle remains one of my primary inspirations for pursuing an academic career.

Through my many years at OGI, many faculty have been generous with their time discussing research issues large and small, in collaborations, or simply in collegial gatherings. When I started at OGI, Calton Pu and Crispin Cowan were central to the vibrant group culture that deepened my interest in research. It was a real treat to visit and work with Charles Consel and the Compose group at IRISA. Dylan McNammee and David Steere further motivated my interest in systems in general. The members of PACSOFT, especially John Launchbury, Tim Sheard and Mark Jones were extremely supportive of my interest, and my somewhat cranky attitudes, in matters of programming languages. Finally, I can only say that OGI and the SySL are extremely lucky to have Wu-chi Feng and Wu-chang Feng on board. I am particularly grateful to Wu-chi who helped me refine my work, and provided a direct contribution to this thesis in the form of the simulation software that I used to compare my work to previous approaches.

Over the years at OGI, I worked with many great students, staff members, visitors, and interns. My dissertation stands squarely on my predecessors in DSRG, Shanwei Cen and Richard Staehli. I also would like to thank Mark Jeffereys, Erik Walthinsen, Josh Gruenberg, Perry Wagle, Dan

Revel, Anne-Francoise Le Meur, Jim Snow, Francis Chang, and Mike Shea, Luca Abeni, Kang-Li and Jie Huang. I couldn't have asked for a better office mate than Ashvin Goel.

Contents

List of Tables

List of Figures

Abstract

A Framework for Quality-Adaptive Media Streaming:
Encode Once — Stream Anywhere

Charles Krasic

Supervising Professor: Jonathan Walpole

This dissertation presents a general design strategy for streaming media applications in best effort computing and networking environments. Our target application scenario is video streaming using commodity computers and the Internet. In this scenario, where resource reservations and admission control mechanisms are generally not available, effective streaming should be able to adapt to variations in bandwidth in a responsive and graceful manner. The design strategy we propose is based on a single simple idea, adaptation by priority data dropping, or *priority drop* for short. We evaluate the efficacy of priority drop in the video and networking domains.

For video, we show how common compression formats can be extended to support priority drop, thereby becoming *streaming friendly*. In particular, we demonstrate that priority-drop video allows adaptation over a wide range of rates and with fine granularity, and that the adaptation is tailorable through declarative adaptation-policy specifications. Our main technical contribution is to show how to express adaptation policies and how to do *priority-mapping*, an automatic translation from adaptation policies to priority assignments on the basic units of video.

In the networking component of this thesis, we present two versions of *Priority-Progress Streaming*, a real-time best-effort streaming protocol. The basic version does classic unicast streaming for video on demand style streaming applications. The extended version supports efficient broadcast style streaming, through a multi-rate multicast overlay.

We have implemented a prototype video streaming system that combines priority-drop video, priority mapping, and the Priority-Progress Streaming proctocols. The system demonstrates the following advantages of our approach: a) it maintains timeliness of the stream in the face of rate fluctuations in the network, b) it utilizes available bandwidth fully thereby maximizing the average video quality, c) it starts video display quickly after the user initiates the stream, and d) it limits the number of quality changes that occur. In summary, we will show that priority-drop is very effective: a single video source can be streamed across a wide range of network bandwidths, and on networks saturated with competing traffic, all the while maintaining real-time performance and gracefully adapting quality.

Chapter 1

Introduction

Audio and video are increasingly important on the Internet. In fact, with the sustained trends to lower costs of computing and storage, greater deployment of broadband access, and development of file sharing applications such as Napster and its successors, there has been a significant surge in interest and usage of the Internet for transport of audio and video [77]. An eventual convergence toward Internet distribution of audio and video seems likely, although a number of serious technical and social challenges remain. Streaming is one technology component that is sure to play a significant role as this convergence unfolds.

The motivation for streaming is to provide the same instant access to continuous media that the web gives to text and images. However, there are a number of technical problems that must be addressed for streaming to reach its full potential in applications for communication and entertainment. In this chapter, we describe the basic technical problems and briefly overview the common approaches used to address them.

1.1 A basic overview of the streaming problem

The elementary problems for continuous media applications in general are the resource limitations of the fundamental resources: processors, storage, and network. For streaming applications, the network resource is perhaps the primary concern because of all the resource types, wide-area bandwidth costs are the most expensive and the slowest to improve. Hence, one of the primary research challenges in video delivery is reducing the bandwidth costs. At a very high level, there are two general ways to reduce the resource costs of video delivery over networks: one is to improve the representation of video data, through compression, the other is to make the network

distribution mechanism more efficient, both through basic improvements in network technology and through video specific distribution techniques.

There has been a great deal of research into video compression, and video compression is now commonly available. Most notably, current techniques can achieve very high compression, with ratios as high as two orders of magnitude being quite typical. On the distribution side, there have been steady improvements in the speed and cost of basic networking technologies, such as link types, switches, routers, etc. At a higher level, techniques such as caching and multicasting have been explored to exploit various forms of locality of access to video content.

Apart from the basic desire to reduce overall transmission cost, there is a very important secondary problem for streaming, which is dealing with the consequences of variable video and network rates. Intuitively, the basic job of any streaming mechanism is to deliver video across the network with the proper timing, so that it is displayed at the receiver at the proper rate and without interruption. For these timing requirements to be met, it follows that the volume of video data transmitted—as determined by the video's bitrate requirements—must not exceed the available bandwidth in the network. As it turns out, both the video and the network rates are highly variable over time. The question of how variable, and whether variations can be accurately predicted, is in fact the subject of considerable research. However, the reasons for variation in the video and networks rates are quite straightforward.

Video bitrates are bursty due to the use of video compression, which as we mentioned above is motivated by the desire to reduce cost. As with any data compression, the goal of video compression is to identify and eliminate redundancy. Video content is variable virtually by definition, because from one video to the next, there are random choices of camera angle, patterns of movement, changes of scene, etc. Predictive coding methods, where compression encodes some frames (predictive frames) as the difference relative to others (reference frames) leads to variation too. For example, as the similarity between a predictive frame and its reference increases, the compression ratio improves, but the variation increases since a larger percentage of the information is carried in the reference frame. In the limit, all the information comes from the reference frame and the predictive frame encodes zero changes. Thus, it follows that more efficient video compression naturally leads to bitrate profiles that more closely track the inherent variabilities (the entropy) of the source content.

Network rates are volatile because of the elementary properties of the Internet architecture. On the Internet, the available bandwidth varies over time because of the best effort sharing model on which the Internet service model is based. The Internet protocols in general do not perform any kind of admission control or attempt to provide service guarantees. The success of the Internet is evidence of its cost effectiveness in providing wide area connectivity, which, it could be argued, has a fairly direct connection to its best effort model. We take the position in this dissertation that the best effort nature of the Internet will not change fundamentally in the foreseeable future, so that variable bandwidth must be assumed to be part of the streaming problem.

Given that video and network rates are fickle, the simple observation made earlier that the video rate should be kept below the available bandwidth is non trivial to achieve. To address this problem, *quality-adaptive* approaches to streaming have been developed. The basic idea of all of these approaches is to adjust the compression ratio of the video adaptively, so that the timeliness of video playout is maintained. Adjusting the compression ratio is possible because of the lossy nature of the common video compression formats. Unlike other data types (such as normal text), lossy compression methods are preferable for video because they yield major gains in compression efficiency, in exchange for a minor reduction in video fidelity. The amount of fidelity lost, the distortion, is generally a tradeoff against the amount of compression. In quality-adaptive streaming, this rate-distortion tradeoff is manipulated for the purpose of rate-matching the video to the network bandwidth.

The overall goals for quality-adaptive streaming are to make quality-rate adjustments so that streaming is effective, yet efficient and scalable. Achieving these goals requires addressing a number of sub-problems, which we describe in the following section.

1.2 Quality-adaptive streaming requirements

The description above included the essential properties of quality-adaptive streaming. We now expand upon the sub-components of the problem, and on how they relate to the overall goals of effectiveness, efficiency, and scalability. As a group, these problems constitute an interesting systems challenge, because they span boundaries of several distinct research domains, such as video compression, networking, and real-time computing.

1.2.1 Scalable video compression

Video representation plays a central role in the set of problems for adaptive streaming. As mentioned earlier, the resource requirements for representing video necessitate the use of video compression. The basic goal of compression is to reduce the number of bits required to represent a given video, or equivalently to reduce the bitrate of the video over time. The conventional compression problem might be framed in terms of maximizing video quality for a target bitrate (or minimizing the bitrate for a target quality). In practice, the quality-rate goals must be tempered against the computational requirements of compression and decompression, so that real-time display is feasible. As CPUs have generally improved, so too has the raw rate-distortion efficiency of video compression technology.

In adaptive streaming, we assume that the network is a best effort resource, hence there is no way to select a unique target rate ahead of time. Instead, the strategy of adaptive streaming is to adjust the video rate according to network conditions. Thus, the requirements for video compression must be re-phrased, so that the goal is to support some range of possible rates, which is commonly referred to as *scalable compression.* The aim of scalable compression can be viewed as a direct generalization of the aim of conventional compression, where the more generalized version is to maximize the quality across a range of target rates (or minimize the bitrates across a range of quality levels). In practice, the range of adaptation will be both limited in its span and it will be discrete rather than continuous. Hence, adaptive streaming has additional new goals, which are to maximize the range of supported rates and quality levels, and to provide the finest granularity of realizable points within that range. The greater the range and the finer the granularity, the more freedom there will be in rate-matching the video with the network bandwidth.

1.2.2 Finding the best mix of video adaptations

Video quality is multi-dimensional and consequently, there are several ways to adapt the quality-rate tradeoff. Smoothness of motion, spatial detail, spatial size, and accuracy of color, are just a few of the aspects of video quality that can be adjusted to alter the rate requirements. However, the best mix of adaptations will be content, task, and user specific. Finding the best mix possible is important, but it is predicated on the capability to influence the mix in the first place. Therefore,

given that there is no single best way to adapt video in general, or even a given video, it would be preferable to support a range of adaptation mixes, and to make mixed adaption decisions in a policy driven manner. On the one hand this has implications for the video representation, which must provide some way to effect adaptations of various independent dimensions of video quality, while still meeting the basic goals of compression efficiency. On the other, the network transport needs to be able to interact with the video representation to effect the policies, without unduly compromising modularity or efficiency.

1.2.3 Micro-level streaming problems: effectiveness

In this section, we turn from how video can facilitate adaptive streaming toward the actual problems of the delivery process itself. We have described the goals of quality-adaptive streaming to be effective, efficient and scalable. In this section, we expand upon the notion of what it means for adaptive streaming to be effective in the microscopic sense (from the perspective of an individual user). We propose the following four criterea to categorize the sub-problems in adaptive streaming: robustness, utilization, latency, and consistency.

Robustness

The overall goal of streaming is to deliver the video across the network such that the receiver sees a continuous playout at the correct rate and without interruptions. Here we are concerned with the timing implications of the rate-matching process: is the chosen video rate low enough that video data arrives on time for proper display? Depending on the streaming approach, we also may be concerned with what happens when some of the data is damaged or lost entirely in transmission. We define the degree to which streaming can avoid interruptions in the face of network conditions as its *robustness*. Due to the best-effort nature of the network, and the volatility of rates that result, robustness is a principal aspect of the effectiveness of a streaming approach.

Utilization

Although avoiding streaming failures is important, the quality of the video also matters a great deal. It is certainly possible to achieve a very robust solution if we ignore the resulting quality. A higher bitrate generally translates to higher video quality, therefore we observe an opposite

pressure to robustness, namely utilization: is the chosen video rate high enough to make full use of available bandwidth, hence, is the user experiencing the highest quality video possible? This aspect of streaming effectiveness is particularly relevant looking forward to the future, when we can expect a general trend toward better infrastructure (processors, storage, and networking). Rigid streaming approaches are in a sense forced to sacrifice utilization in the future, in favor of robustness in the present (the time video is first made available). There's a certain irony to the situation, since infrastructure improvements may seem less desirable if they don't yield noticeable increases in the user experience.

Latency

Video delivery can take on numerous forms, so it is worth emphasizing what distinguishes streaming from other approaches, particularly downloading. We describe streaming as being fundamentally about providing the same level of instant access to audio and video as the web does for text and images. For the web, document downloads are largely sufficient. Documents can generally broken down into pages which are small enough that their download times are acceptable to users, which is to say that users still spend the majority their time reading the documents, rather than waiting for downloads. If we take the same approach to continuous media, the wait time for downloads will certainly exceed the tolerance threshold needed to maintain the illusion of instant access. It might be argued that instant access is not important, and it would be difficult to formally prove that it is. However, we can refer to the history of the Internet for anecdotal evidence. Before the web, the Internet consisted mainly of download mechanisms, such as e-mail, ftp, and Usenet news. Although these mechanisms still exist, it seems clear that the instant, interactive, access of the web was a major boost to the utility of the Internet, and has generally increased our overall access to information.

The main technical premise of streaming is to eliminate the wait of downloads by making the transmission and display processes happen at the same time (as opposed to waiting for the entire transmission to complete before display can commence). In practice, streaming mechanisms will still be subject to delays due to bandwidth restrictions, propagation delays, buffers, etc. These delays can vary considerably from one streaming approach to the next, so we view the latency of a streaming mechanism as an important evaluation criteria. It is also important to point out that

latency can be measured in different ways, some of which may or may not reflect what the user experiences. Therefore, we define two user-centric notions of latency as the following:

Navigation latency: The delay between when a user initiates some navigation action and when they see the corresponding result, for example the delay between when a user chooses to start play and when the video starts to play.

Communication latency: The continuous measurement of the delay between when video enters the streaming process at the sender to the time the corresponding video is displayed.

It may not be obvious, but these two definitions may have different values for the same system (later chapters will explain fully). We make the distinction between them because their importance to the user will vary according to the type of video application. For instance, in video on demand, the navigation latency will be apparent, but once playing, the communication time is not perceptible. On the other hand, in a video phone application, the communication latency is critical to maintaining the natural flow of human conversations.

Consistency

Our argument for a quality-adaptive approach rests on the assumptions that the video rates and network rates are inescapably variable over time, and impractical to treat completely in advance. Thus, we argue that quality adjustments are necessary as part of the streaming process in light of the above robustness, utilization and latency criteria for effective streaming. However, we also recognize that quality changes can be distracting to the user. We define the *consistency* of adaptive streaming in relation to the frequency and magnitude of quality changes which the user may perceive (fewer and smaller changes mean better consistency). In due consideration of the other criteria, we also put forth that greater consistency will lead to a better user experience.

Summarizing Effectiveness

We have framed the effectiveness of streaming in relation to aspects that would be apparent to the individual user, through a definition of four sub-components to streaming effectiveness: robustness, utilization, latency, and consistency. On a number of levels, some obvious and some subtle,

the metrics associated with these sub-components are in conflict with each other, so a high level of overall effectiveness will be more about striking a good balance among the parts, rather finding a perfect solution to any one. In the next section, we turn to the macroscopic goals (network wide) for streaming: efficiency and scalability.

1.2.4 Macro-level streaming problems: efficiency and scalability

Video applications are potentially a major threat to the current stability of the Internet. The basic fact that video has much higher bandwidth requirements per user than other types of traffic means that it has greater potential to take an unfair share of bandwidth away from existing traffic, and generally congest the network. This is a major concern in the Internet, where resource sharing is largely a voluntary and co-operative activity.

Despite their greedy nature, video applications must behave as good network citizens. The majority of current Internet traffic is TCP based, therefore streaming transports need to be *TCP friendly* [80]. Intuitively, TCP friendly means that a flow consumes bandwidth in the same way that TCP would. Since TCP includes congestion control, this means that the flow will back off its transmission rate in times of congestion. Just as the current TCP applications, video must employ congestion control to ensure that the network can avoid congestion collapse and that video traffic shares fairly with other classes of traffic.

Scalable distribution

Another part of the macro-level streaming problem is finding ways to take advantage of locality of reference among the users of video. We call this the *scalable distribution* part of the streaming problem. Two major approaches to scalable distribution are content distribution networks (CDNs) and multicast. At a very high level, their common goal is to eliminate redundant network traffic due to the fact that different users are accessing the same content. CDNs and multicast are distinguishable by the form of locality they treat. CDNs are more about spatial locality and Multicast is more about temporal locality. The basic idea in a CDN is to employ persistent storage replication in the network to lighten the load on wide-area links. In CDNs, the emphasis is on exploiting spatial locality through the use of this persistent storage, involving techniques such as proxy caching. The locality is spatial in the sense that the CDN will try to service users from

caches that are physically the closest (in network distance) to the users. In multicast, the emphasis is more on temporal locality, in the sense of users accessing the same content at the same time (like a broadcast). Multicast is lighter weight than a CDN, because it doesn't require persistent storage in the network. CDNs and multicast do have common elements: their goal is to increase the number of video users that the network can support, and they both require some form of in-network assistance (e.g., caching and multicast forwarding).

Efficiency concerns

Scalable video distribution mechanisms transform the role of the network from a simple forwarder toward a more active participant. Consequently, there is a need to ensure that quality-adaptation mechanisms are efficient enough for in-network implementation. In particular, network nodes in CDNs and multicast may need to implement quality-adaptations in the network, due to the fact that they partition the end-to-end delivery path (in time or space) between the original source and the receiver. To preserve network utilization, these in network devices will need to be able to sustain their quality-adaptation duties at full line rates, which could easily reach Gigabit levels at interior points of the network. To support scalable distribution, quality-adaptive video delivery should strive for computationally simple adaptation mechanisms.

1.3 Overview of our approach: the Priority-Progress framework

In the previous section, we enumerated the main problems and some of the sub-problems for video streaming in the Internet. In this section, we give an overview of our approach to streaming, and the manner in which it addresses the above problems.

Our overall approach is named *Priority-Progress*, because of the connection between its central theme of informed data dropping and the two essential data attributes necessary to implement it in time-sensitive applications: a priority and a timestamp. This dissertation presents a complete framework for quality-adaptive streaming through the Priority-Progress approach. The framework treats the problems of video representation, adaptive streaming, and scalable distribution described in the previous section.

The key components of the Priority-Progress framework are outlined in the following subsections.

1.3.1 SPEG: Scalable MPEG

Although video coding is not the main subject of this dissertation, the availability of scalable media-compression formats is a principal assumption of the Priority-Progress approach. This dissertation describes one example of such a format that we have developed, called SPEG (Scalable MPEG). A single SPEG video file supports video adaptation in multiple quality dimensions. Moreover, the space of adaptation is over a very wide range with fine granularity. SPEG demonstrates the benefits of scalable compression in eliminating the "one target rate" constraint of conventional compression.

1.3.2 Priority Mapping

Adaptive streaming needs a policy driven way to choose from the many possible ways to mix video adapatations. The policy can reflect content, device, author and user requirements. The policy specification should be as declarative as possible to avoid exposing unnecessarily the complexities of the underlying streaming mechanisms. In our approach, policy specifications take the form of utility functions. The utility functions express the prefered mix of adaptations across the range of acceptable quality levels. We present a *Mapper* that accepts these utility functions and automatically prioritizes the units of a video stream such that priority order dropping of the data results in the specified mix of adaptations.

1.3.3 Priority-Progress Streaming

During transmission, we want to adapt video according to the rate decisions of a TCP-friendly congestion control. In this way, TCP friendliness is assured. We present an algorithm for quality-adaptive transmission called Priority-Progress Streaming (PPS). The basic idea in PPS is to send high priority data before low, but to stream successfully we need to manage timing and priorities simultaneously. The PPS algorithm defines how this is done, in a manner that aims to maximize both robustness and utilization. Furthermore, it has an important component called *window-scaling* that

provides a way to balance two conflicting objectives of adaptive streaming: low navigation latency and high consistency.

1.3.4 Priority-Progress Multicast

We extend Priority-Progress beyond unicast Video on Demand (VoD) to broadcast. By restricting the service model to synchronized viewing of a finite number of channels, a distribution network can scale to a potentially unlimited number of viewers. Individual link and node stress is bounded by the number of channels, not the number of viewers. We simulate broadcast via multicast, in our case Priority-Progress Multicast (PPM). In PPM, we construct a multicast overlay from a tree whose edges are independent PPS unicasts. PPM is a TCP-friendly multi-rate multicast. PPM nodes in the interior of the tree perform a very simple dropping algorithm, which can be implemented very efficiently in commodity hardware. TCP friendliness and multi-rate adaptation greatly simplify the problems of scalable distribution. In addition to multicast style broadcasts, PPM could also be a building block for video CDNs and Peer to Peer (P2P) VoD in a way that is resilient to problems of "flash" crowds.

1.4 Software Prototype

We have developed a software prototype to support the claims of this dissertation. Our software prototype is called QStream (short for the Quasar Streaming System). QStream is a complete video streaming system that includes a streaming video server, a streaming player, a multicast proxy, and a remote monitor[1]. The QStream prototype includes implementations of the SPEG, the Priority-Mapper, and the PPS and PPM algorithms. We have made the source code for QStream publicly available under the terms of the GNU Public License (GPL). The QStream prototype serves as the basis for qualitative and quantitative experimental evaluation of the thesis ideas.

[1]The remote monitor provides real-time visualization of an extensive set of streaming statistics through a software oscilloscope.

1.5 Thesis Statement

This dissertation presents a software framework for media streaming over the Internet based around novel adaptation techniques. The distinguishing characteristics of our framework are related by an overall goal of supporting an "encode once, stream anywhere" model of media streaming. The specific hypotheses of this dissertation are as follows: a) streaming video can be made scalable with fine granularity over a wide range of rates; b) tailorable adaptation policies can be used to control the mixture of adaptations to best meet content, task, and user specific requirements; c) this kind of video leads to an enhanced user experience when streaming takes place over typical Internet links; d) the video can be streamed over networks in a TCP-friendly way making it easier to deploy in the real world; and e) TCP-friendly video streaming can be applied efficiently to multicast delivery, enabling large scale video broadcast distribution.

Today's streaming systems are not adaptive enough to cope with the unpredictable quality of service in the Internet. The basic best-effort architecture of the Internet is unlikely to change, so streaming systems must adopt adaptation that can address the volatility of Internet quality of service (QoS). In this dissertation we will argue the reasons for inadequacy of current approaches. We summarize the state of the art as follows.

Today's streaming techniques are unreliable and the quality is generally poor. In practice, streaming is usually restricted to small video clips, on the order of a minute or less. Of course, the advantages of streaming for this scenario are small, since the additional startup delay for a download may not be very large. For longer duration content, where the benefits of streaming over download should manifest, the current systems show their fragility with respect to transient surges in network activity. In order to reduce the occurrence of failures with longer duration content, the typical approach is to set a very conservative target rate for the video, significantly lower than the average available network bandwidth. This under-utilization of the network resource is highly ironic given the vast amount of research effort spent on improving the compression efficiency of video codecs.

We argue that Internet video streaming must employ better adaptation to provide acceptable quality and robustness. In this dissertation, we present a uniform approach to adaptation called Priority-Progress that has the goal of supporting an "encode once, stream anywhere" level of

simplicity in Internet streaming systems. The central idea is to employ data dropping, uniformly and in a fashion informed by priority and timing information in the data, as a means of adaptation to best effort network conditions. We show the efficacy of this idea through a complete strategy for adaptive streaming, and its implementation in a working prototype system, the scope of which spans from the video encoding to the network streaming protocol.

1.6 Summary of Contributions

To summarize, the main contributions of this dissertation are as follows:

- SPEG and the Priority Mapper: We show how through proper framing and prioritization, video need only be encoded once, yet it can support a wide range of bitrates with fine granularity. Moreover, the mix of adaptations within the range is explicitly controllable so that user, content, task and device specific requirements can be optimally addressed.

- Priority-Progress Streaming (PPS): We develop an adaptive streaming protocol that achieves several important objectives, namely robustness, high utilization, consistent quality over time, and low navigation latency.

- Priority-Progress Multicast (PPM): We extend PPS to multicast distribution through an overlay approach. PPM supports multi-rate, quality-adaptive, multicast distribution, in a completely TCP friendly manner. To our knowledge, prior approaches have only been able to a subset of these characteristics simultaneously.

- QStream prototype: the above conceptual contributions are made concrete in a comprehensive prototype implementation. This prototype is the basis for our experimental evaluation of our framework. The prototype itself also constitutes interesting contributions toward programming for time-sensitive applications, such as the use of reactive programming, and the use of remote, real-time, visualization techniques.

1.7 Dissertation outline

The remainder of this dissertation is organized as follows. In the next chapter we revisit the problems presented in this chapter and discuss them in relation to the related work. In Chapter 3, we describe SPEG video and priority mapping. In Chapter 4 we give an overview of the PPS protocol. We give an overview of our multicast protocol, PPM, in Chapter 5. Chapter 6 presents the complete algorithms for unicast and multicast streaming as implemented in the prototype software, QStream. Chapter 7 presents an experimental evaluation of unicast (PPS) and multicast (PPM) streaming. Chapter 8 presents our conclusions and suggests problems that require future work.

Chapter 2

Background and Related Work

The basic idea of Internet streaming is a relatively old one, the first systems for streaming audio and video over the Internet have existed since the early to mid 1990's [8, 60, 10, 59]. However, the impact of streaming so far seems to be well short of its potential. In this chapter we present a brief history of streaming, describe the current state of the art, and point out relationships between the contributions in this dissertation and the previous work.

2.1 On the limited impact of streaming

Since the time that Internet streaming technology first became available, many popular Internet sites have incorporated streaming content. News sites such as `cnn.com` and `nytimes.com` regularly provide news clips in streaming formats. Entertainment oriented sites such as `iFilm.com` and `AtomFilms.com` feature movie trailers and some short films in streaming formats. Music videos are also available via sites like `mtv.com`. It is difficult to obtain exact estimates of how many Internet users utilize streaming content, but anecdotal evidence suggests that most users at least have a streaming player installed on their computer, and this could be taken as an indication of the mainstream status of streaming. On the other hand, it would seem that relatively few people actually use streaming on a regular basis, certainly less than the other basic Internet technologies, such as e-mail, the web, instant messaging, and more recently peer to peer (P2P) file sharing systems [1]. One might interpret the limited success of streaming as a general indication that people are not as interested in using the Internet for audio and video, but this interpretation can be quickly dismissed. One only has to consider the developments related to Napster and the subsequent P2P

[1] Again, this is difficult to verify in the literature, but anecdotal evidence is strong.

systems for strong evidence to the contrary. In one recent traffic study [73], it was shown that video and audio account for the first and second largest portions of Internet traffic. Individuals (and businesses) are clearly very interested in using the Internet to deliver audio and video. We believe there is a large gap between the current and potential popularity of streaming and that this gap is due to a mixture of technology, business, and social-legal issues. Although the focus of this dissertation is entirely on technology aspects, we must acknowledge that resolutions to the other non-technological issues are essential to enable many streaming applications. Nevertheless, the work in this dissertation is motivated by the belief that current streaming technology could improve significantly, and that such improvements can eventually contribute to more wide spread use of streaming. We now turn to the history of streaming so far, and to the current state of the art.

2.1.1 Streaming in practice

Very soon after the first streaming systems appeared and were described in the literature [10, 59], there was rapid development and competition among commercial software platforms for video streaming. This competition led fairly quickly to a division (which remains today) into three popular platforms: Microsoft's Windows Media, Real Networks' RealSystem [13], and Apple's QuickTime. These three platforms consitute the basis for the majority of streaming on the Internet. There are also some systems developed by academic projects, such as the Quasar project at OGI [10, 87] and the MASH project at Berkeley [59], but those systems did not develop large user communities.

To various degrees, the popular streaming systems adhere to a suite of Internet standards related to streaming, such as RTP [74], RSTP [75], SIP [34], and SMiL [33]. Despite the standardization efforts, the three commercial systems were still largely proprietary. It is especially worth noting that the core issue we address in this dissertation—how to adapt to variable resources—are almost entirely outside the scope of the Internet standards. The Real-Time Protocol (RTP) protocol, perhaps due to its rather general name, is often confused with a complete solution to streaming. In fact, RTP and its various sub-profiles are mainly limited to specifying syntax for timing and other information, and they do not specify at all how to manage timing of the streaming process, nor do they provide any algorithms for adapting to available resources. We argue that Internet streaming is unlike most other Real-Time applications, because of the best effort nature

of the services provided by the Internet infrastructure. Classic real-time design generally involves worst-case analysis and testing of system timeliness properties, and provisioning sufficient resources for the worst case. However, neither video nor the Internet are well suited to worst-case analysis, because the rate properties of both (video and available bandwidth) are known to be extremely bursty [29, 91, 14]. For video, the burstiness is due to the use of video compression. The burstiness of available bandwidth on the Internet is due to its best effort model, and to the absence of end-to-end resource provisioning services. We will now discuss why these properties are unlikely to change, either for video or for the Internet, and hence motivate the subsequent discussion of adaptive streaming approaches.

2.1.2 Video compresssion and variable bitrates

When television was developed, it used entirely analog electronics. The advent of computer technology led to the development of digital representations of video. Raw (uncompressed) video has extremely high rate requirements, especially when the video is represented digitally. For example, a television grade signal in digital form is typically coded to about 125 megabits per second (Mbps)[2]. When digital video equipment was first developed, only highly specialized and expensive devices could deal with such high resource requirements. Soon after, when video compression techniques were introduced, it became possible to reduce the rate requirements of video dramatically. Initially, the goals of using compression were tied to making digital video usable with more affordable equipment. For example, the development of the MPEG-1 standard was closely tied to the advent of consumer CD-ROM devices. Hence, MPEG-1 was often associated with a target video rate of 1.5Mbps which fit nicely within the capabilities of early generations of CD-ROM devices[41]. Since the time that MPEG-1 was developed, there has been significant progress in improving video compression [35, 42]. However, the progress in hardware has been even greater, and it seems likely that general purpose hardware will continue to improve more quickly than compression. Thus it is worth considering whether the expected advances in hardware will eliminate the need for video compression altogether. For the forseable future, we believe that the need

[2] 125 Mbps is based on 720 x 480 pixel frames with 12 bits per pixel (YUV formats using 6 bytes for 4 pixels) and 30 frames per second.

for compression will persist because compression technology already can reduce the rate requirements of video by between one and two orders of magnitude. This reduction in rate requirements translates to cost reductions of the same proportions for storage and network bandwidth resources. Thus, as long as there are non-negligible costs for storage and network bandwidth, the incentive to utilize compression will remain strong. However, the space and bandwidth benefits of compression do not come without some disadvantages.

One of the primary disadvantages of video compression is that compressed video has rate requirements that are highly variable over time. Statistical studies have shown that compressed video is bursty over the full range of time scales [29]. While the rate reduction from compression aids streaming, it is also true that compression hinders streaming because it adds the requirement of dealing with significant rate changes over time. To be clear, these changes are not a deficiency of video compression, they just reflect the fact that the entropy in video will naturally change as the content varies betweeen frames and from one scene to the next. Some video encoders offer purported support for Constant Bit Rate (CBR) coding, but they do so at the expense of compression efficiency[3]. As a more efficient alternative, various buffering techniques have been developed to smooth out these rate changes for network transmission. Feng [25] has compared the performance of a selection of the proposed smoothing techniques. However, all of the approaches examined in Feng's study assumed that there was some fixed level of available bandwidth, which would be true if it were possible to provision network bandwidth in advance. While some special purpose networks (such as networks dedicated solely to video broadcast) may have that property, the general-purpose Internet does not. To cope with network bandwidth changes, adaptive streaming is necessary. In this dissertation we will present an integrated approach that deals with video rate and network bandwidth variations simultaneously.

Before we turn our attention to adaptive streaming, we first review two considerable efforts towards augmenting the basic services of the Internet: *Multicast* and *Quality of Service* (QoS). Both of these efforts share better support for video delivery as a principal goal. Multicast concerns how to improve the scalablility of Internet based delivery. QoS concerns the goals of provisioned resources and predictable service.

[3]Most CBR coders involve some amount of zero padding of the video data to smooth the rate.

2.1.3 Multicast

Multicast is a transmission technique that aims to support efficient, one-to-many data transmissions such as video broadcasts. Multicast is especially attractive for multimedia data such as video because the high bitrates of video and very large numbers of receivers can combine to make the cost of a unicast based approach prohibative.

Multicast works by organizing the transmission to a group of receivers into a tree structure, where the data is replicated at interior branch points of the tree. The effect of the tree is to limit the amount of stress placed on any single point in the distribution. By stress, we mean the the number of copies of the packets of a given flow to traverse a given node or link. An ideal multicast tree will limit link stress to exactly 1 per active session, and the node stress will be the degree of each node (number of directly connected edges). In a unicast scenario, the node and link stress at the source of the distribution will be the same as the total number of destination receivers. With multicast, the lower maximum stress means that the resource demands on nodes and links in multicast can be spread more evenly throughout the network, hence multicast enables sharing the available resources better with other traffic than unicast, which can easily overwhelm capacity of links and nodes close to the source.

IP Multicast (RiP)

IP multicast [18] was proposed to incorporate multicasting as a basic service primitive in the Internet. Intended as an interim solution, the MBone [20] was developed to allow IP multicast traffic to cross regions of the Internet without native IP multicast support. For various reasons, which include a mixture of technical and economic issues, full deployment of IP multicast has yet to materialize [19]. Some of the technical reasons include problems with inter-domain routing protocols, problems with management of the multicast address space, and the lack of congestion control in multicast transports. The economic reasons include the pervasiveness of asymmetric "policy" routing in the Internet, where Internet Service Providers (ISPs) configure routing policy within their own domain so as to cause foreign packets to exit as soon as possible, rather than

taking the shortest route to the destination[4]. For this reason, routes on the Internet are often asymmetric taking different paths in each direction between two end hosts (because the end hosts use different service providers). The asymmetric aspect of policy routing can conflict with the standard mechanisms used to establish routing for multicast trees (e.g., reverse-path routing). Some ISP may even perceive an economic disincentive to accept "foreign" multicast packets at all, due to the interaction with service level agreements (SLAs) between ISPs. SLAs are usually enforced based on packet accounting mechanisms at border points between ISPs. However, multicast packets may replicate at points other than the border, thus bypassing the SLA accounting. Network operators are thus reluctant to allow IP multicast within their networks. After more than a decade of development, IP multicast service is unavailable to most end users of the Internet. Due to the slow deployment of IP Multicast, recent research is revisiting some of the assumptions of the IP multicast design.

Multicast Revisited

In the wake of IP multicast's shortcomings, there have been many recent multicast proposals that move away from the notion of multicast as an IP primitive and move toward application level approaches (overlays). Unlike the original multicast overlay—the MBone—these new proposals also explore altering some of the basic assumptions of what the multicast service model should be.

Although multicast can potentially generalize to many kinds of data distribution, there has always been a fairly close association between multicast and video delivery. As mentioned in Section 2.1.2, the high bitrates associated with video make it desirable to find ways to reduce bandwidth costs. Multicast can be thought of as a kind of compression technique in so much as it can reduce the overall network requirements to distribute a video to a group of users. Also, like video compression, we see that multicast can bring additional complexities, which can only be justified if the savings from using multicast are large enough. For the purpose of this dissertation, we classify the challenges in multicast into two main categories: tree management and data forwarding.

[4]A *foreign packet* is one who's source address does not match any of the ISP's customers.

In multicast, one of the basic issues is how to establish and maintain the topology of multicast distribution trees. For instance, in order to simplify the address space and routing issues relative to IP multicast (as well as certain security related issues), many of the recent overlay approaches assume a strict, single-source model, as opposed to the more general many-to-many model supported by IP multicast [94]. Another common direction has been to combine multicast with ideas from the many recent peer to peer (P2P) approaches, which employ various strategies with the common theme of achieving scalable and self-organizing routing [11, 45, 81]. One of the key problem areas is how to find multicast topologies in an overlay that deliver the best performance. A related question is what metric or combination of metrics should be used as in selecting and judging topologies. The fact that overlays often only maintain partial routing information—which can obscure the true underlying network topology—is a significant complication because edges that are distinct in the overlay topology may in fact have common physical links in the real network. However, many of the recent systems are intentionally designed to accept the possiblity of slightly suboptimal topologies (due to only partial knowlege) in exchange for much better scalability properties.

Much of the work in multicast is on the routing side, and does not treat the data forwarding issue. The default multicast forwarding function is relatively trivial: it simply duplicates and forwards data as necessary from the incoming edge to the outgoing edges in a best effort fashion. One of the goals of the IP multicast architecture is to keep the forwarding function as simple as possible, preferably with minimal overhead relative to unicast IP forwarding. To maintain this simplicity, purely end-host managed mechanisms had to be used to add other features to the multicast service model, such as reliability [95, 28], and congestion control [85, 69, 7]. Although the desire to minimize complexity inside the network is well intentioned and has been a long-proven strength of the Internet architecture [72], the purely end-host based schemes for reliability and congestion control were complex and, in some cases, it was difficult to judge if they achieved the desired effect. For instance, congestion control mechanisms that rely on joining and leaving multicast groups work on different time scales than TCP's congestion control, so it is unclear how fairly they share. Given the trend toward overlay multicast instead of IP multicast, it stands to reason that an overlay might be designed with more advanced data forwarding functionality, moving some of the complexity back into the network in the hope of realizing a better overall

solution.

In this dissertation, we explore an overlay based approach for multi-rate adaptive multicast of video, which treats the data-forwarding side of multicast. Our approach can be combined with recent topology management techniques mentioned above to produce a fully adaptive multicast solution. As with the unicast components of this dissertation, our focus is on adaptive delivery, which is motivated largely by our expectation that resource provisioning will be unavailable to most users of the Internet for at least the near future and perhaps well beyond that. Section 2.2.6 will discuss the related work on adaptive multicast.

2.1.4 Internet QoS

From the early days of the Internet, it was recognized that the Internet's best-effort architecture was at odds with the goals of time sensitive applications such as audio and video [12]. Often, the design principles of telecommunications networks used for traditional voice traffic (telephone) were held up as examples for how digital networks can provide provisioned resources. For some time, there was a general belief that there would be some kind of unification between the architectures of data networks typified by the Internet, and the more traditional telecommunications networks. Many expected ATM technology to achieve this unification, bringing with it true support for Quality of Service (QoS) in the Internet. In networking circles, the QoS term is often used to denote support for various forms of resource reservation. Good support for video was an often touted payoff for the transition to ATM and QoS support. A core design decision in ATM was that the use of small fixed size packets (cells in ATM terminology), which allowed ATM to provide various degrees of predictable service, and better switching latency properties. However, a significant transition of the Internet towards ATM did not materialize and seems completely improbable now, for reasons described well elsewhere by Kalmanek [46].

In addition to the activity surrounding ATM, there were efforts to add QoS support directly to the existing Internet protocols. The approaches included two broad IETF programs: Integrated Services (IntServ) and Differentiated Services (DiffServ). At a high level, the two differed in their granularity. IntServ was more ambitious, and aimed to augment the Internet with mechanisms that would provide *per flow* reservations, mainly via the RSVP protocol [92]. RSVP did not succeed in achieving significant deployment. The resistance to RSVP was due to scalability

problems (overall it was too heavy weight) and its high overheads [15]. DiffServ was an attempt at a lighter weight solution, with coarser granularity of provisioning [6]. Unlike IntServ, which was aimed at end-user per flow reservations, DiffServ was aimed at aggregate reservations mainly for use between enterprises and service providers. However, by the time DiffServ arrived, service providers had already decided that simply over provisioning the core of the Internet was a simpler and cost effective enough solution to many of the problems addressed by DiffServ. As described by Crowcroft et al. [15], DiffServ was "too little, too late".

After more than a decade of significant effort, QoS support (especially on the "last mile" to end-users) is almost non-existant. However, the research in the area remains active, and there are new approaches (e.g. Subramaniam describes an overlay approach to QoS support [79]) that may eventually prove more successful. Nevertheless, the lessons from the QoS efforts so far tell us that it would be unwise to design video streaming to depend entirely on QoS support. Instead, we should consider video streaming and Internet QoS as separate and complementary technologies. Video streaming should benefit from QoS support if and when it is available, but it should deal gracefully with the best effort infrastructure that seems likely to be with us for a long time to come.

2.2 Adaptive Streaming

Given the apparent barriers to including resource provisioning in the basic Internet service model, our position is that an adaptive approach is essential to deal with the diversity and volatilty of the various resources involved in Internet based streaming.

There has indeed been a great deal of work in the literature related to adaptive video streaming, which spans several distinct domains, including video compression, real-time systems, and networking (unicast and multicast). A spectrum of adaptive strategies have been proposed to deal with the consequences of the Internet's best effort service model, which we summarize below. The surveys of [93] and [84] are also good references.

In the remainder of this section we expand on the basic performance issues for quality-adaptive streaming in light of some of the approaches proposed in the literature and in terms of commercial streaming systems. The related work on quality-adaptive streaming relevant to this dissertation falls into six categories: single rate adaptation, multi-version techniques, online-scaling, scalable

compression, adaptive unicast streaming, and adaptive multicast streaming. We will discuss each of these areas below.

One of the main issues that spans each of these areas is the granularity of adaptation. Ideally, a quality-adaptive streaming system will select video quality to match the average available network bandwidth. In practice, adaptation tends to be limited to discrete steps, and consequently the rate match is only approximate. A system that supports steps with finer-granularity generally results in a better match, which manifests itself in higher quality and better reliability of streaming. The type of video compression, especially whether the compression is scalable or not, is a major factor influencing the granularity of quality-adaptive streaming. Because many of the compression formats in common use are not explicitly scalable [41, 40, 42, 43], the target rate is a required parameter for encoding. These formats do not provide explicit support for adapting rate after encoding. Frame dropping is a well known work-around, and is probably the most popular video adaptation mechanism, having been used since the first quality-adaptive Internet streaming systems appeared [10]. In addition to frame dropping, there are other important ways to adapt video, which will be discussed in the following sections. When multiple adaptations are combined, it can help to increase the granularity of the space of adaptations.

2.2.1 Single Rate Adapation

In practice, one of the most commonly used strategies is *one-time adaptation*, where the user chooses between a small set of predetermined rates before streaming begins. Once started, streaming is fixed at this single-rate regardless of competing traffic. One-time adaptation has two basic problems: it is prone to yield lower quality than necessary when more bandwidth is available, and it is prone to complete failure when less bandwidth is available (requiring play to pause while client side buffers are re-filled). Both problems become more probable as the duration of streaming increases. From a user's perspective, the result of these difficiencies are that streaming for long periods has poor quality and is unreliable. Apple's Quicktime uses one-time adaptation and, in addition, it adjusts the amount of client-side buffering based on measured rate volatility during startup [83]. Consequently, startup time while initial buffering is established can be quite high—on the order of tens of seconds. Windows Media and RealSystem based systems are often configured in this mode also, even though they offer advanced mechanisms as options.

2.2.2 Multi-version Techniques

Multi-version streaming systems store a single video at a range of pre-selected bitrates, with the goal of switching adaptively between the versions according to available bandwidth. Multi-version adaptation benefits from being able to use conventional, non-scalable codecs, which are more commonly available. Also, conventional codecs presently achieve higher compression efficiency than scaleable codecs. The main drawback to multi-version adaptation is that, to keep encoding and storage overheads reasonable, it requires the selected bitrates divide the range of rates with coarse granularity. This leads to quantization effects in the adaptation controllers, which hinders their ability to attain utilization and robustness. A poor controller has similar problems to the one-time adaptation strategy, it can under utilize average bandwidth, or it can be unreliable in times of congestion. It is worth noting that, to the user of the video, the under utilization of bandwidth can negate the compression efficiency advantages of using conventional codecs. The coarse division of rates also makes it difficult to offer control over the kinds of adapation that occur, so as to best match specific requirements. Despite these limitations, multi-version adapatation is the most widely deployed technique for stored content, as it is used in Windows Media IntelliStream [5] and Real's SureStream [13].

2.2.3 Online scaling

Online-scaling techniques, which include live encoding, transcoding, and data-rate shaping (DRS), allow changing the target rate parameter of the encoder or transcoder on the fly [44, 96]. Transcoding and DRS can have significantly lower computational complexity than live encoding. The main advantage of online scaling is very fine granularity. However, even DRS (the most efficient of online-scaling methods) is very computationally intensive relative to non-adaptive streaming, or to adaptive streaming through frame dropping or multi-coding[5]. While online scaling allows very fine-grained adaptation, the computational time required to recode limits scalability quite severely. Online scaling is best suited to applications which only require support for just a single receiver or perhaps for a small number of receivers. However, the computation cost of online scaling in

[5]DRS either requires a full encoding step, or a transcoding step, which decompresses all the video blocks sufficiently to modify them, and then re-codes the result.

servers and edge devices make it an ill-advised choice for supporting large numbers of independently adaptable streams.

2.2.4 Scalable compression

Scalable video coding technologies focus on creating compression formats that allow adaptation of the rate-distortion relationship without explicitly re-coding (e.g. MPEG-2 scalability, MPEG-4 FGS). In contrast to online scaling, scalable compression aims to support low-complexity adaptation that will scale to large numbers of streams. Scalable compression schemes explicitly support multiple quality levels, exposing two or more layers in the encoded video. The layers are progressive: the higher layers depend on the lower layers, and the higher layers are used to refine quality. The various scalable compression approaches differ in terms of granularity, ranging from very coarse, as in the work in Layered Multicast [61] and MPEG-2 Scalability [35], to very fine, such as in recent work in MPEG-4 and H.26L Fine Granularity Scalability [55, 37]. With the current state of the art in scalable video compression, there remains a compression efficiency penalty, in that video quality from scaleable compression is lower compared to the results of non-scalable compression at the same rate, but this penalty is getting smaller [37]. Fine granularity scalability through layering makes it possible to begin streaming without even knowing the target rate by sending lower layers before higher layers and truncating higher layers if time runs out. The contrast between this approach and online-scaling, where the quality adaptation must commit to a target rate *before* encoded data is ready to transmit, is worth noting. In exchange for the small efficiency penalty (see Section 3.4), scalable compression offers a significant boost in freedom for the design of adaptive streaming mechanisms. Scalable compression techniques are complementary to the work we describe in this dissertation. We expect advances in scalable compression can be easily and directly incorporated into our framework.

While the three categories above are concerned mainly with video representation and coding, the next two categories concern *adaptive streaming*, that is, the mechanics of actual network delivery. We discuss adaptive streaming with respect to unicast scenarios first, and then we extend our view to adaptive multicast.

2.2.5 Adaptive Unicast Streaming

A principal concern with streaming is the potential impact of video traffic on existing Internet traffic. Many research projects have studied quality adaptive streaming in relationship with *TCP-friendly* congestion control [93, 66, 24, 22, 44, 78, 50]. A common idea among them is to let the transport protocol and its congestion control dictate the appropriate sending rate. The main differences are in the details of deciding what to send and what to drop, and what information are used to inform these control decisions. For example, Rejaie *et al* describe their algorithms for optimal streaming [66], where optimal means minimal client-side buffering, and thus a minimal associated contribution to end-to-end latency. The role of their algorithm is to control adding and removing quality layers, where the control decisions are based on a rate-driven feedback control. The design of their control is based on analysis of additive-increase multiplicative-decrease (AIMD) congestion control[6] and an assumption of *a priori* knowledge of video rate requirements [66]. *Feamster et al* extend this work to more general congestion control mechanisms [22]. Recently, Dai et al. [16] have presented an approach that proposes to integrate scaleable compression (MPEG-4 FGS) with adaptive streaming, using alternative congestion controls based on Kelly controllers.

In contrast to these systems that explicitly attempt to match rates, *Feng et al* describe an adaptive streaming algorithm that uses a sliding window over video frames, sending data from low to high quality, in best effort fashion [24]. Feng's algorithm gains simplicity because it does not attempt to minimize client-side buffering absolutely, and it has the advantage of working without direct assumptions about the design of the underlying congestion control. Kang et al. [47] propose a priority-driven adaptation, but assuming fixed bandwidth channels. Prior to the work in this dissertation, the question of how to link scalable video encoding and tailorable adaptation policies to TCP-friendly streaming was mainly an open one.

The framework presented in this dissertation uses scalable compression and TCP. One of the contributions of our approach is to demonstrate the benefits of using the priority-timestamp packet as the basic unit of media abstraction, as opposed to video frames or layers in a stream. Through priority mapping, we extend scalable video compression to support tailorable adaptation so that compromises made in quality better reflect the influence of specific content, viewing devices, and

[6] TCP's congestion control uses an instance of AIMD after it reaches steady state.

user preferences. Our Priority-Progress Streaming (PPS) algorithm extends TCP-friendly adaptive streaming to support direct control over quality compromises in streaming, such as relative preferences between different dimensions of video quality, latency tolerances, and limits on the number of quality changes, while preserving the goals of high utilization and video quality.

2.2.6 Adaptive Multicast Streaming

The same basic principles that motivate an adaptive approach for unicast streaming apply to multicast as well. To summarize, the bitrate of video is variable because of the use of compression, and the available bandwidth is bursty because of the diverse makeup of the network and its best-effort service model. Thus, an adaptive approach is needed to scale the video to match network conditions. Moreover, where unicast streaming is concerned with adapting video to the highest rate appropriate to a single receiver, in multicast there are multiple receivers, each of which may have a unique appropriate rate. In contrast to unicast streaming, work on adaptive multicast has been mainly confined to research, it is not yet supported by any of the popular commercial streaming systems. In the remainder of this section, we summarize some of the related work on adaptive multicast from the literature. For more details, Liu *et al.* have done a nice survey of the area [56]. We remark that we restrict our view to multicast streaming, noting that there has been considerable work on multicast downloads as well.

Solutions for adaptive multicast can be either single rate or multi-rate. In both approaches, one of the primary goals is to share the network fairly with other traffic, that is, the adaptation should serve a congestion control function like that of TCP. In the single rate approach, the adaptation matches the rate of the entire tree to that of the slowest link. Obviously, this penalizes some receivers in the tree, but it may be simple to implement. In the multi-rate approach, the goal is to find the best rate for each receiver, in a way that avoids penalizing faster receivers in the presence of other slower recievers in the tree .

Receiver Driven Layered Multicast (RLM) was an early proposal for adaptive media streaming over IP multicast for continuous media such as video [61]. The basic approach was to have the media partitioned into layers that were associated with individual multicast groups. The layers in the groups are progressive in that a base layer is used to carry the minimum quality version, and enhancement layers each refine the quality (presuming each of the lower layers are also present). The

layered multicast approach is necessarily coarse grained—typically the layer sizes are distributed exponentially, for example, each higher layer is double the size of the previous lower layer. The coarse granularity is necessary to limit the number of multicast groups, and to limit the number of join and leave operations, each of which can take significant time to complete. These join and leave operations are the basis for adaptation in RLM, RLM receivers increase and decrease their data rate by joining and leaving multicast groups. The receiver oriented nature of RLM is driven by the goal of scalability. One of the main advantages (at least at the time of its design) of the RLM approach is that it builds entirely upon the existing features of IP multicast service. This advantage is somewhat moot now, as deployment of IP multicast hasn't reached significant numbers of users (see Section 2.1.3). A major disadvantage of RLM is its coarse granularity of adaptation, which among other things means that RLM is unable to share the network fairly with TCP traffic.

Extensions to the RLM approach to try and address the issue of TCP friendliness for the single rate case have been proposed by Vicisano et. al [85] and later extended by Rizzo [69]. A number of studies have also looked at approaches to multi-rate congestion control [86, 71, 48, 53]. The priority drop approach we take in the multicast component of this dissertation is quite a departure from those works, which generally assume a layered source stream. In these layered approaches, adaptation is co-ordinated by recievers or the sender. In our approach, the internal nodes of the multicast tree participate in the priority dropping decisions. Also, the TCP friendliness of our approach is very easy to understand, since we use TCP.

Multi-rate adaptive multicast has also been one of the main target problems for a variety of proposals for *Active Networks*. The basic idea of Active Networks was to add various forms of extensibility to routers. Most of the literature in Active Networks concerns the infrastructure support needed to provide extensibility. However, some of the work took a higher level view towards target applications such as video [96, 49, 88, 36]. Yeadon's PhD work [96] proposed various ways to adapt video in an active service framework. Our inspiration to develop SPEG came mainly from Yeadon's work. Yeadon's filters were also our inspiration to investigate an alternate approach to simplify the operations required of network nodes, because the computational complexity of his filters is a serious impediment to scalability. In some sense, Yeadon's filters are similar to Amir's earlier work on application level video gateways [1] in that they both do transcoding in the network. The approach in this dissertation is to restrict in-network operations to priority dropping, so

that our multicast data forwarding has much more modest computational demands.

Chapter 3

Streaming Friendly Video

In this chapter, we describe how we extend a common type of video compression to support graceful quality adaptation (via priority data dropping). In our approach, video can adapt across multiple quality dimensions simultaneously and the mix of adaptations is tailorable, meaning that the adaptations can be controlled through an explicitly specified policy. These features are made possible by a component of our system called the *Priority Mapper* (hereafter referred to simply as the Mapper) that automatically converts specified adaptation policies into appropriate priority assignments. The policies express relative quality preferences, which may be content, user, device, and task specific. The Mapper assigned priorities are such that priority-order data dropping will cause video quality to degrade in the least important aspects first, according to the specified policy.

The architecture of our adapative streaming framework divides the adaptive streaming process into several distinct stages. The common point of reference between the stages is priority-drop based adaptation. In this chapter, we describe the two stages that constitute the video preparation phase of our architecture: video encoding and priority mapping. The adaptive streaming stage will be described separately in Chapters 4 through 7. By treating video encoding and priority mapping as separate stages, we allow the possibility that there may be multiple priority assignments for the same video, which might be used to tailor the mix of adaptations to diverse usage scenarios. For example, the mix of adaptations for a user with a small screen on a mobile device may differ from a user with a workstation or even a home theater. Taking another example, the adaptation mix may need to change temporarily whenever the viewer decides to do a slow motion replay. Our separation between encoding and priority mapping makes it easy to tailor the adaptations as necessary without incurring the cost of multiple encodings.

Our video preparation stages—encoding and priority mapping—can be done before (offline)

or during (online) the actual streaming process. Because, in this dissertation, we focus mainly on streaming of stored content (video on demand), we chose to do the encoding offline, and the priority mapping online. It is also possible to do the priority-mapping offline, perhaps several times to form a canned set of adaptation mixes. This would have less flexibility than mapping online, but it would decrease the overall workload of the streaming server. We have not actually implemented offline mapping, because the computational cost of online mapping hasn't actually been significant in practice. On the encoding side, we have done an online version in the context of a live streaming variant of our system (from a webcam source) [38]. Whatever the chosen staging is, the result of the preparation stages is a generic encapsulation of the video data, where priority and timestamp information capture the essential information necessary for the streaming process to effect graceful adaptation. The general techniques we have taken for video could potentially be applied to other streaming data types (e.g., measurements from environmental sensors), with appropriate encoding and priority-mapping analogues, while our Priority-Progress network protocols should continue to work without changes.

3.1 Scalable Video

The video format used in our streaming system is called SPEG (Scaleable MPEG). Although scaleable compression is an important component of our framework, it is not the primary focus of this dissertation. SPEG adds a fairly basic level of spatial detail scaleabilty to MPEG, just enough to demonstrate the basic properties we expect from more optimized scaleable codecs. In particular, we are interested in the ideas that a scalable coding can be adapted gracefully across a very wide range of rates and quality levels, and that more than one dimension of video quality is adaptable. At the time this work began, there were few items in the literature to describe such encodings, and no publicly available software implemenations that we knew of[1]. More recently, other researchers have developed high performance scaleable compression with similar properties for current standards such as MPEG-4 [55, 37]. In the body of this chapter, we give a basic review of video coding along with the details of how SPEG adds spatial scalability. Readers familiar with MPEG and FGS [35, 55] may choose to skip this material. We will conclude the discussion

[1] At the time of writing, ours is still the only publically available implementation that we are aware of.

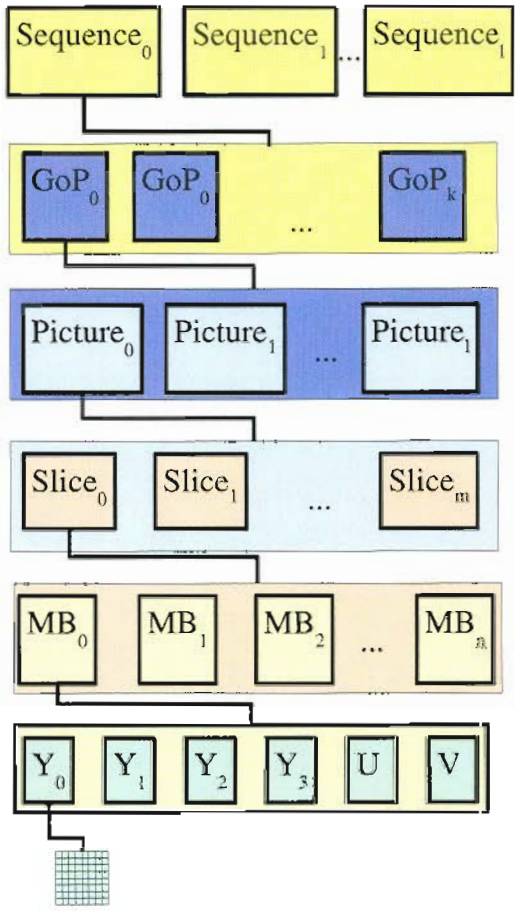


Figure 3.1: Typical Hierarchical Structure of Compressed Video

of SPEG with some observations of how we expect future scalable compression formats will be improved to better assist streaming delivery.

This dissertation does not expect the reader to be familiar with the mathematics of signal processing that underlies video coding. We attempt to provide explanation to give the intuition of the mathematics, although understanding the compression aspect is not critical.

Figure 3.1 shows the typical structure of compressed images in common video formats such as MPEG. A video consists of a sequence of groups of pictures (GOPs). A GOP is a sequence of frames[2]. A frame is decomposed into sub-units, such as MPEG slices. A slice consists of a

[2]Although we try to use the term "frame" consistently in this dissertation, it should be noted that the terms image, picture, and frame are synonymous

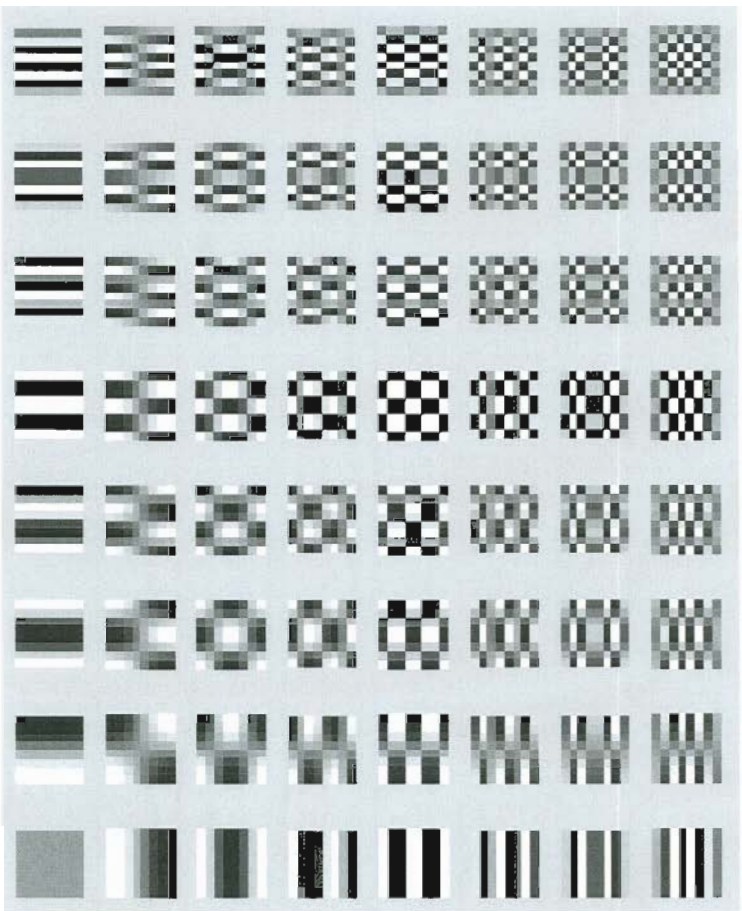
sequence of macro blocks, which in turn consists of fixed numbers of blocks with a block being a pixel array. Slices serve a combined purpose of allowing limited error recovery in the event of bit errors, and some ability to fine tune certain encoding parameters for specific regions of a frame. However, the error recovery role of slices in MPEG is very limited and is not intended to solve the kinds of problems that occur during network transmission. The purpose of the macroblock relates to the fact that it is common practice in video to represent color with lower fidelity than luminance (color subsampling). So typically, as in MPEG, a macroblock might contain four luminance (Y) blocks, and one block each for the two color components (U and V). Finally, a block is an array of pixel values. In MPEG-1 through 4, a single fixed 8x8 size is used for all blocks. Upcoming revisions to MPEG-4 will support a range of possible block sizes.

The foundation of MPEG compression is the treatment of the data in the blocks. MPEG transforms the pixel values using a staple technique from signal processing called a frequency transform. The original values are considered to represent values of a signal over time, where the position of the pixel value within the block determines the time point in the signal, and the value of the point represents the amplitude of the signal. The transformation produces a function over the frequency components of the signal that approximates the orginal signal. MPEG uses the Discrete Cosine Transform (DCT). The form of the function is a summation over cosines. The output of the transform is the set of coefficients for the cosine terms of the sum. The value of each coefficient represents how strongly the original signal contains the frequency of that corresponding term in the sum. Intuitively, a larger value for a higher frequency term means the values of corresponding pixels change quickly. Conversely, if the pixel values are similar to each other in the area of the block, then only the coefficients of lower frequency terms will have large values.

Figure 3.2(a) shows the formula of a cosine transform. The shape of the function as a whole is the same as the original sequence of pixel values. Figure 3.2(b) gives a visual representation of the two dimensional (horizontal and vertical) DCT basis, which can give some of the intuition for how they are combined to represent arbitrary signals. From left to right, the basis functions have increasing frequency in the horizonal direction, and from bottom to top, frequencies increase in the vertical direction. The bottom left is the zero frequency basis, often referred to as the DC component, because it has a constant value (i.e., by analogy to electronics with DC "direct current", as opposed to AC "alternating current"). The remaining components are referred to

$$
\begin{aligned}
F(u,v) &= \frac{1}{4}C(u)C(v)\sum_{x=0}^{7}\sum_{y=0}^{7} f(x,y)\cos((2x+1)u\frac{\pi}{16})\cos((2y+1)v\frac{\pi}{16}) \\
u,v,x,y &= 0,1,2,\ldots,7 \\
C(j) &= \begin{cases} \frac{1}{\sqrt{2}} & \text{if } j=0 \\ 1 & \text{if } j>0 \end{cases}
\end{aligned}
$$

(a) DCT equation used in MPEG-1



(b) Visual representation of DCT basis functions

Figure 3.2: The Discrete Cosine Transform (DCT)

as AC components. The DCT transform function takes a two dimensional array of pixel values and produces a set of weighting coefficients, which, when applied in a summation over these basis functions, results in the best approximation to the input pixel values. The DCT coefficients constitute a frequency domain representation of the signal (pixel data). Using the inverse DCT (IDCT), the transform can be reversed (applied to the coefficients) so as to produce original pixel values.

The compression benefit comes from the fact that, whereas the original pixel values may be distributed evenly (albeit randomly), the frequency domain version will tend to be concentrated in a few low frequency coefficients. The high frequency coefficients will tend to have small values, close to zero. Or rather more importantly, the human eye is more sensitive to the parts of the image that are represented by the low frequency coefficients. This is the crux of the compression gains for lossy video encoding compared to losseless techniques. It is possible to drop information from the higher frequency coefficients with a relatively low perceivable impact on the image. By reducing the numeric precision used to represent these coefficients, a simple conventional compression algorithm, for example run-length and Huffman coding, can achieve orders of magnitude higher compression ratios compared to the same algorithms applied directly to the original pixel data. So then *quantization*, which is the strategic removal of low order bits from the DCT coefficients, is the primary basis for compression gains in MPEG and very many other similar compression schemes. Increasing the quantization (i.e., reducing the precision of coefficients) reduces the total size of the compressed representation, in exchange for degradation of the fidelity of the reconstructed image. The job of matching a target bit rate for an MPEG stream is done by a component of the encoder called the rate control.

Whereas rate control tries to find quantization values that cause the compressed video to match a target rate statically, a spatially layered coding partitions the coefficient data so that quantization can be done dynamically and incrementally. This is the basic idea used in SPEG.

SPEG transcodes MPEG coefficents to a set of levels (one base level and three enhancement levels) as follows. If we denote the original MPEG coefficents $X[i, j]$, then SPEG partitions this coefficient data according to the following equations[3]:

[3]The $>>$ denotes the right bitwise shift operator, and the & denotes the bitwise *and* operation.

$$
\begin{aligned}
X_{base}[i,j] &= X[i,j] >> 3 \\
X_{e0}[i,j] &= (X[i,j] >> 2) \;\&\; 1 \\
X_{e1}[i,j] &= (X[i,j] >> 1) \;\&\; 1 \\
X_{e2}[i,j] &= X[i,j] \;\&\; 1
\end{aligned}
$$

The four layers are denoted $X_{base}[i,j]$, $X_{e0}[i,j]$, $X_{e1}[i,j]$, $X_{e3}[i,j]$ respectively. The layers in each SPEG frame are the basic *application level data units* (ADUs) in SPEG. The above steps can be reversed to return SPEG back to the original MPEG. Alternatively, we can drop some or or all of the enhancement layer ADUs (from high to low) subsituting zero values for the missing data. The effect of such dropping is analogous to having used higher quantization parameters during MPEG encoding, yielding lower bitrate in exchange for less spatial fidelity. SPEG suffices to demonstrate the essential properties of scalable compression, albeit with lower compression efficiency and fewer layers than something like MPEG-4 FGS.

We expect future scalable codecs will expose even more scalability mechanisms. One example is *spatial-size scalability*, where the number of pixels of height and width are scalable. Another example is *chroma scalability* which might allow a range of color fidelities, from 4:4:4 to 4:2:2 to 4:1:1 to greyscale to monochrome. The object based compression techniques starting to appear in standards like MPEG-4 might allow *content adaptation* through addition and removal of objects [42]. These possibilities raise the issue of *tailorable adaptation*. In order to take full advantage of all of these scalability options, there would need to be a good way to control how they are used together. To explore tailorable adaptation, we use SPEG's spatial scalability in combination with frame dropping to provide a minimal example of a compression scheme with more than one scalability mechanism.

Finally we note that a deficiency of SPEG, and of MPEG rate control, is that we do not explicitly know how much image degradation a given level of quantization causes. We will discuss this issue further in Chapter 8.

3.2 Priority Mapping

Having more than one quality dimension raises the issue that choosing how to best adapt the multiple dimensions may depend on the usage scenario. For example, the target device may

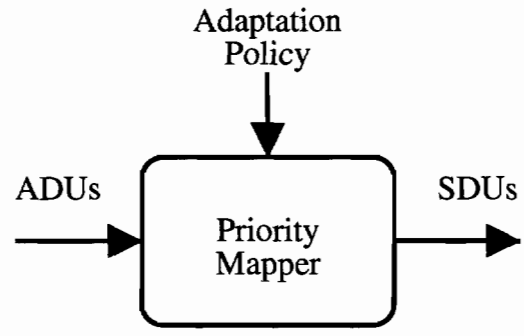


Figure 3.3: Priority Mapper

have a small screen, so preserving frame-rate may make more sense than spatial detail. A user may want to repeat a scene in slow motion, which looks smoother if more frames are inserted. Conversly, skipping frames is harder to notice when doing fast-forward scan. We have designed a priority-mapper with the intent of providing a general approach to tailoring quality adaptation to such specific quality preferences. A priority-mapper automatically assigns priorities to the units of a media stream, so that priority drop yields the most graceful degradation, as appropriate to the viewing scenario.

Figure 3.3 depicts the mapper used in our framework. The mapper's inputs are application data units (ADUs) and the quality adaptation policy. The output of the mapper is a sequence of streaming data units (SDUs). Each SDU contains a group (subset) of the input ADUs, along with a timestamp and priority computed by the mapper algorithm.

Figure 3.4 shows a set of ADUs. The ADUs have a packet like form, consisting of a fixed-length header, and a variable length payload. The header contains basic information needed by the mapper, such as the position and length of the payload. The mapper can examine the payload to infer other properties, such as the timestamp, the type of MPEG frame the ADU is part of (I, B, or P), and to which spatial scalability layer the ADU belongs[4].

We use *utility functions* as declarative specifications for adaptation policy. A utility function is a simple and general means for users to specify their preferences. Figure 3.5 depicts the general form of a utility function. The horizontal axis describes an objective measure of lost quality, while

[4]To simplify our examples, Figure 3.4 depicts only two spatial layers although our SPEG implementation has four.

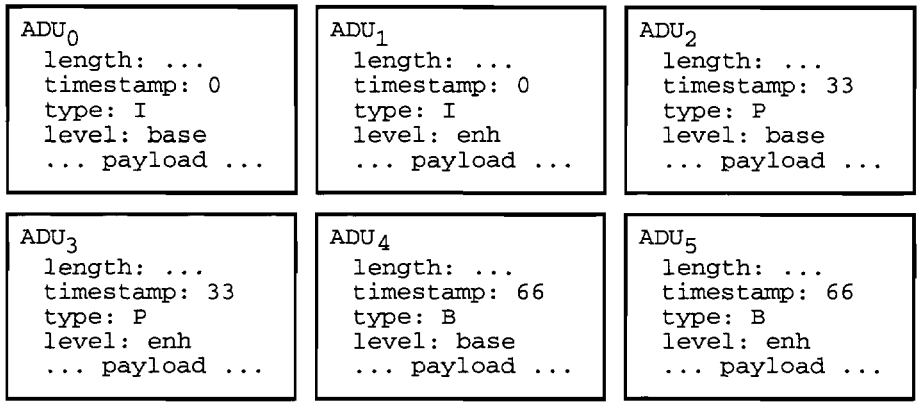


Figure 3.4: ADUs

the vertical axis describes the subjective utility of a presentation at each quality level. The region between the q_{max} and q_{min} thresholds is where a presentation is acceptable. The q_{max} threshold marks the point where lost quality is so small that the user considers the presentation "as good as perfect." The area to the left of this threshold, even if technically feasible, brings no additional value to the user. The rightmost threshold q_{min} marks the point where lost quality has exceeded what the user can tolerate, and the presentation is no longer of any use. The utility levels on the vertical axis are normalized so that zero and one correspond to the "useless" and "as good as perfect" thresholds. In the acceptable region of the presentation, the utility function should be monotonically decreasing, reflecting the notion that decreased quality should correspond to decreased utility. In the case of priority mapping for SPEG, the adaptation policy consists of two utility functions, one for spatial quality and one for temporal quality.

The mapping algorithm subdivides the timeline of the media stream into intervals called *mapping windows*. The mapper algorithm prioritizes the ADUs within each window separately. We use the ADUs from Figure 3.4 as an example mapping window, which consists of a single GOP and spans the interval 0–66 ms. The priority mapping algorithm processes the ADUs within a window in two phases.

In the first phase, the ADUs are partially ordered according to a *drop before* relationship, which we denote using the $\prec$ symbol [5], based on video data dependencies. For example, the

[5]This is really drop no-later than, since dropping is always optional.

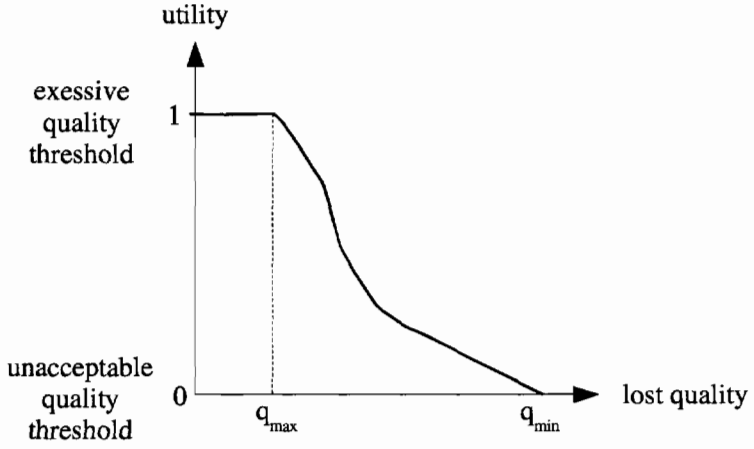


Figure 3.5: A utility function with thresholds

spatial layering requires that base layer ADUs should not be dropped before their corresponding enhancement layer ADUs, which applies to the ADUs of Figure 3.4 as follows:

$$ADU_1 \prec ADU_0 \qquad ADU_3 \prec ADU_2 \qquad ADU_5 \prec ADU_4$$

In the example of Figure 3.4, ADU_1 is the enhancement layer of the I frame with timestamp 0, and ADU_0 is the base layer of the same I frame, so it follows that ADU_1 should be dropped before ADU_0.

Similarly, MPEG's predictive coding rules (for I,P,B frames) are expressed as follows:

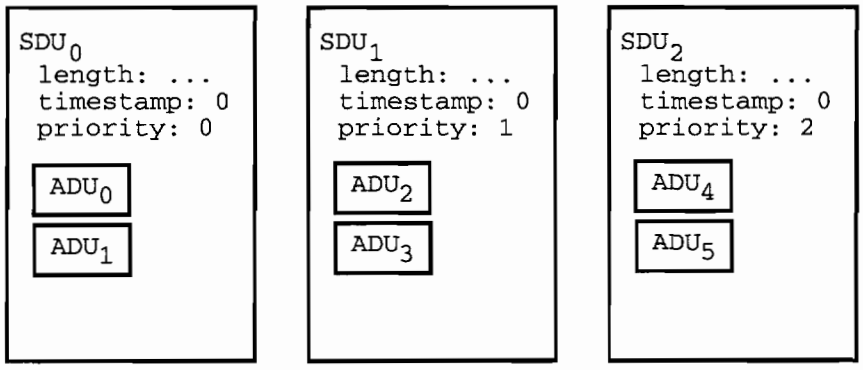
$$ADU_4 \prec ADU_2 \prec ADU_0$$

Again from the Figure 3.4, ADU_2 is the base layer of a P frame that depends on the I frame whose base layer is contained in ADU_0, so we have that ADU_2 should be dropped before ADU_0.

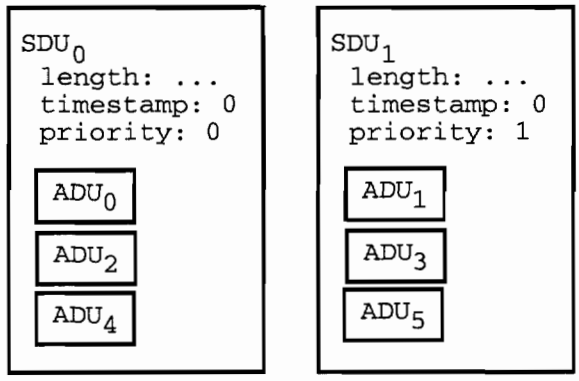
These first two sets of ordering constraints represent *hard dependency* rules, in that they simply reflect SPEG semantics. The mapper adds some other *soft dependency* rules which improve adapation results. With video, for example, the mapper would add soft-dependencies so as to ensure that frame dropping be as evenly spaced as possible[6].

[6]If half the frames are to be dropped, then our in our experience, it has been clear that it is best to drop every other frame, as opposed to more clustered dropping such as keeping even GOPs and dropping odd GOPs

After the first mapping phase embodying hard and soft dependencies, there still remains significant freedom for adaptation. For example, Figure 3.6 contains two very different mappings for the ADUs of figure 3.4, yet both mappings adhere to the phase one constraints above.



(a) Frame drop only



(b) Spatial drop only

Figure 3.6: SDUs: prioritized and grouped ADUs

The second phase of the priority mapper algorithm is where adaptation policy is used to refine the partial ordering from the first phase, generating the prioritized SDUs. The algorithm works through an iterative process of elimination over the ADUs. We say an ADU is *alive* if it is still in the set of unprioritized ADUs, and *dead* otherwise. Each iteration considers a set of candidate ADUs that are not yet dead (initially all ADUs from the mapping window), and have no living dependants, based on the constraints generated by the first phase. For each of these candidate ADUs, and for each quality dimension (spatial and temporal in SPEG), the mapper computes the

presentation quality that would result if the candidate ADU were dropped, that is, the quality is computed based on all ADUs that are still alive, less the current candidate. For the temporal quality dimension, the mapper computes the frame rate, and for spatial quality the spatial level. At this point the mapper is ready to apply the adapation policy. The utility functions are used directly to convert the computed quality values to corresponding utilities. The "overall utility" for each ADU is just the *minimum* of its per dimension utilities. The candidate ADU that has the highest utility is selected as the next victim (i.e. dropping this ADU next has the smallest impact on utility). The priority value for the victim ADU is a linear (inverse) fitting of the utility into the range of priority values. For example, in the Quasar pipeline this fit goes from a utility range of 0 to 1 to a priority range of 15 to 0[7]. The iterations stop when all ADUs have been assigned a priority.

3.2.1 Mapper window duration

The boundaries of mapper windows are chosen by the mapper to avoid the potential for broken data dependencies in the dropping actions of later stages. In particular, the mapper enforces that mapper windows are aligned with SPEG GOP boundaries. Recall that the GOP patterns (i.e., the combinations of I, B, and P frames) are decided by the video encoder (i.e., SPEG compression algorithm). The GOP pattern need not be fixed from one GOP to the next in SPEG. For stored SPEG content, such as DVDs, GOPs are selected adaptively by the encoder to improve compression efficiency. GOPs often coincide with scene boundaries of the content, so they effectively have a random distribution. We define the lower limit for mapper windows based on the GOPs: mapper windows must contain one or more whole GOPs. Thus, the mapper algorithm assumes it will be given an set of whole GOPs as input. Typically, the longest GOPs will be less than a second, which will be the approximate range for the smallest possible mapper windows. It may split the input set of GOPs into multiple mapper windows (only along GOP boundaries) if their total duration exceeds a threshold parameter. The mapper will not split the set if it would produce a mapper window smaller than the threshold value, hence the largest possible mapper window will be twice the threshold. Keeping the duration of mapping windows bounded by splitting helps

[7]The maximum priority level (for the most important ADUs) is 15

to keep the computational complexity of the mapping algorithm low, and as should become clear later, it will allow a finer granularity of adaptation later on (in the streaming stage)[8].

Once the mapping algorithm has assigned priorities to all of the ADUs in a map window, it then groups them into SDUs. In our algorithm, there is one SDU per priority level, which contains all the ADUs that ended up with that same priority. In addition to the priority, the other main attribute of an SDU is its timestamp. If we didn't group ADUs into SDUs in this way, the most obvious values for the timestamps might be the time values of corresponding video frames to which ADUs belong. However, with our method of ADU grouping, we use a more conservative (coarse grained) timestamp assignment where the SDUs (groups of ADUs) are all set to have the same timestamp as the first video frame in the whole map window. Thus, all the ADUs in a map window are grouped into a single set of SDUs, sharing the same timestamp, but distinguished by priority. This grouping simplifies matters for later stages, like the PPS algorithm and the video decoder, because the timestamps expose the minimum information needed to preserve low-level data dependencies in the dropping process[9]. This organization of one timestamp per map window forms a layer of abstraction which provides just enough detail to perform informed dropping (via the priorities) in a manner that avoids violating low level data dependencies (via the timestamps).

3.3 Mapping Results

We now present the results of mapping for several test movies. Figure 3.7 shows the set of movies, which were prepared with a variety of encoders and encoder parameters.

In Figures 3.8(a) and (b) we set a quality adaptation policy consisting of equal linear utility functions for temporal and spatial quality. Figures 3.8(c) and (d), show the presentation quality derived from this policy for various priority-drop thresholds. At each threshold, the quality corresponds to the point where all packets with priority lower than the threshold are dropped. Increased priority drop threshold means more packets are dropped.

[8]We do not formally characterize the asymptotic complexity of the mapper in this disseration, although we expect it is super-linear in the number of ADUs

[9]Otherwise, there can be pathological cases during streaming where low priority ADUs for one timestamp are kept even though higher priority ADUs with different timestamps, but belonging to the same mapping window, are dropped. For example, a P frame (low priority) might be kept when its I frame (high priority) was dropped, however the P frame can not be decoded properly without the dropped I frame.

Video	Resolution	Length (frames)	GOP length
Giro d'Italia	352x240	1260	15
Wallice and Grommit	240x176	756	3
Jackie Chan	720x480	2437	8
Apollo 13	720x480	864	6
Phantom Menace	352x240	4416	16

Figure 3.7: Movie Inputs. The movies were coded with several different MPEG encoders. A variety of content types, movie resolutions, and GOP patterns were chosen to verify our techniques perform consistently.

Ideally, the presentation quality graphs would look the same as the utility functions they were derived from. In particular, the range of acceptable presentation QoS would be covered, and the shape of adaptation would follow the shapes of the utility functions. Figure 3.8(c) shows the relationship between presentation-QoS for temporal resolution (frame rate) and priority-drop threshold. It should be noted that Figure 3.8(c) contains lines for each of the test movies, but they overlap very closely because the mapper is able to label packets to follow the utility function policy closely. Although desirable, this result was not entirely expected because MPEG's inter-frame dependencies constrain the order in which frames can be dropped, and some GOP patterns are particularly poorly suited to frame dropping. On the spatial resolution side, in Figure 3.8(d), we note that the mapper drops resolution levels uniformly across all frames, resulting in a stair-shaped graph, because there are only 4 spatial levels in SPEG. In as much as the SPEG format allows, the presentation-QoS matches the specified user preferences.

The resource side of the adaptation profiles is shown in the third pair of graphs in Figures 3.8(e) and (f). We show the average bandwidth of the movies at each drop threshold as a percentage of the bandwidth when no packets are dropped. Similarly, we show the CPU time required for client side processing of the video at each drop threshold. A good shape for these graphs would be smooth and linear over a wide range of resource levels. We see that bandwidth in Figure 3.8(e) does indeed range all the way down to only a few percent, although there is a rather sharp drop when the first SNR layer is dropped. CPU time in Figure 3.8(f) is very nice and smooth, although it does not cover as much range as bandwidth, and reaches a minimum of about 10 percent. We also note that the movies are closely clustered in their resource-QoS graphs, indicating that adaptation is independent from differences in encoders or encoder parameters. Further results for other policies

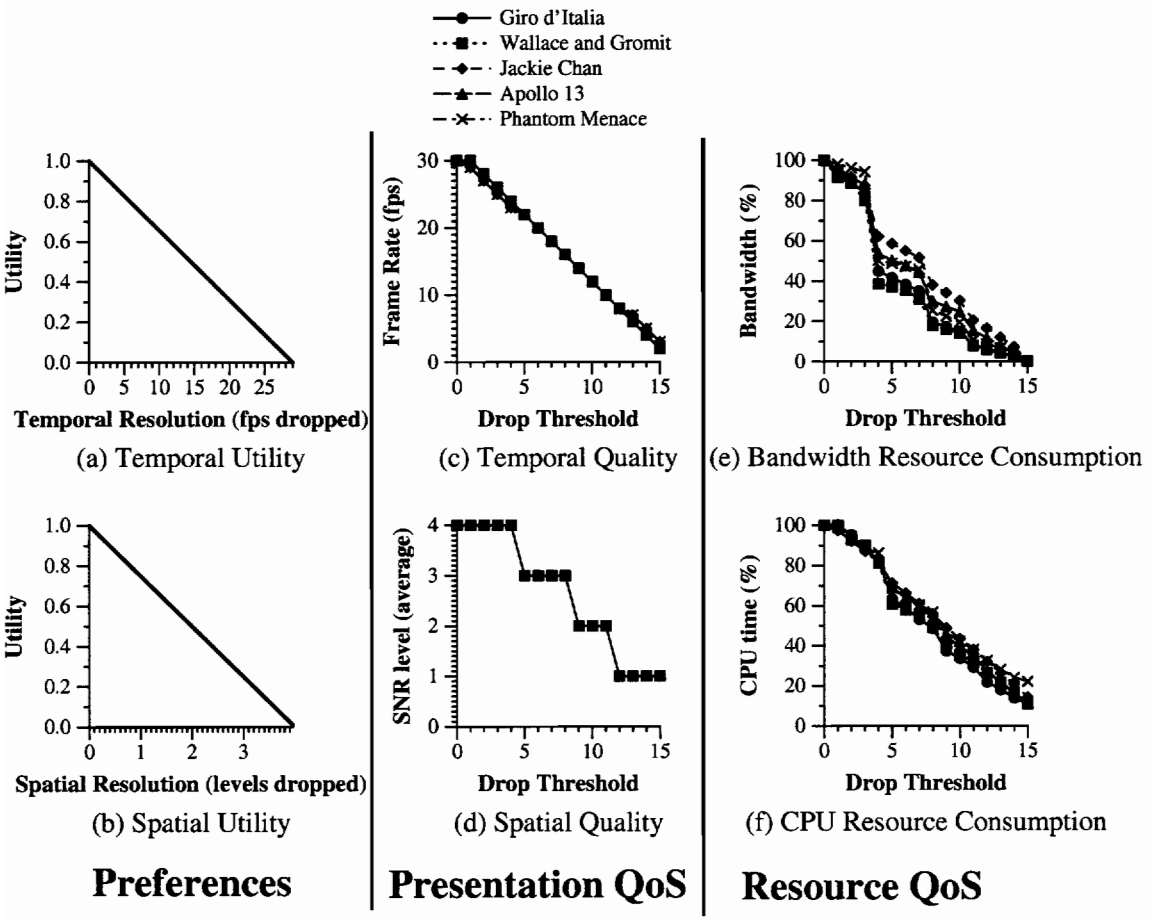


Figure 3.8: QoS Mapping Applied to SPEG

are presented in [52].

3.4 The Price of Adaptation

We now describe some of the performance costs associated with dynamic quality adjustment. Figure 3.9 compares compression performance for MPEG and SPEG versions of movies at the same presentation quality level.

Video	MPEG bandwidth (Mbps)	SPEG bandwidth (Mbps)	Increase bandwidth (%)
Giro d'Italia	1.823	2.121	16.3
Wallice and Grommit	0.968	1.081	12.7
Jackie Chan	1.839	2.479	34.8
Apollo 13	3.474	4.193	20.7
Phantom Menace	1.228	1.313	6.9

Figure 3.9: Bandwidth Overhead of SPEG

Given the wide range of adaptation shown in the adaptation experiments, the relatively small bandwidth overhead of SPEG is encouraging, especially considering the simplicity of the approach used in SPEG. The fact that SPEG defines ADUs based on MPEG slices appears to be the source of much of the variation in increase bandwidth in Figure 3.9. Some encoders only generate one slice per picture, while others generate as many as one per row of macroblocks. Since the main purpose of SPEG is a test vehicle for the Mapper and the PPS protocol, we do not persue further improvements to SPEG's compression efficiency. A production grade implementation could easily replace SPEG with something like MPEG-4 FGS.

3.5 Related Work

As mentioned in the sections above, SPEG is representative of various approaches to scalable compression such as MPEG FGS [55]. Other streaming frameworks have been based upon scalable compression, very notably the recent work of Phillippe de Cuetos on streaming of MPEG-4 FGS [17]. The most important difference in our approach is our focus on generic framing of data

and explicit prioritization. In other words, our approach deliberately prepares the video so that the network layer will only be exposed to generic attributes such as timestamps and priorities. In contrast, de Cuetos framework features a tight coupling between the encoding format and the network transmission layer. In his framework, de Cuetos has delved more deeply into complexities such as receiver-side loss concealment when using lossy transports. On the other hand, we have addressed issues of mixed adaptation across multiple quality dimensions, and (as we will show) quality-adaptive multicast.

3.6 Summary

In this chapter, we discussed how video preparation is handled in our approach. We treat video preparation as a two stage process, where video is first encoded using scalable compression, and then, once encoded, we assign priorities to the video data so as to expose how video should be adapted.

To show how scalable video coding fits into our framework, we introduced a simple scalable video format called SPEG that adds spatial scalability to MPEG. With SPEG, a single encoded video can support a wide range of bitrates and video quality levels with fine granularity. Measurements from our implementation of SPEG showed that the range of bitrates spanned about two orders of magnitude from the minimum to maximum rates. Within the space of supported bitrates, SPEG allows spatial and temporal qualities to be adapted independently. The actual adaptation is determined by the order in which SPEG data is dropped. However SPEG does not fix such an order because that would overly constrain the video before appropriate details of the usage scenario are known.

To best match the video to requirements that might be content, user, task, or device specific, we presented a general strategy for policy-driven priority assignment called the Mapper. The Mapper accepts policy specifications in the form of utility functions, which relate the supportable quality levels (in each of the controllable dimensions of video quality) to their utility to the user. The Mapper algorithm produces a prioritized version of SPEG data such that priority-order dropping has the effect that the least important aspects of video quality degrade first. We have presented results from our implementation of the Mapper to show that it provides effective control over the

mix of adaptations, in that priority-order dropping directly reflects the supplied utility functions. By separating encoding and mapping, we allow the corresponding steps to be done offline, online, or a mixture of both, according to what makes sense for the type of video application. The separation also makes it possible to apply the Mapper several times to the same SPEG video, so as to tailor the mix of adaptations to different usage scenarios.

We call our approach to video preparation *streaming friendly* because it emphasizes flexibility in anticipation of the difficulties of streaming over best effort networks. The next chapter will begin to describe the networking parts of our streaming framework, the Priority-Progress adaptive streaming protocols, where the benefits of streaming-friendly video preparation will come to fruition.

Chapter 4

Priority-Progress Streaming

The previous chapter described our scalable video encoding (SPEG) and our priority mapper (Mapper) algorithm. Recall that the Mapper transforms the application data units (ADUs) of a video into a sequence of streaming data units (SDUs) that have explicit timestamp and priority labels. The purpose of organizing the video into SDUs is to prepare it for our adaptive streaming algorithm, Priority-Progress Streaming (PPS). Based on the SDU timestamp labels, PPS can regulate the progress of the stream so as to ensure that the receiver can achieve proper playback timing. Should the data requirements for the stream exceed the bandwidth available between the sender and the receiver, then the priorities describe the order in which video data may be dropped, from least important to most important, to ensure that video quality degradation is as graceful as possible. So adaptive streaming is able to maintain timing (using timestamps) and adapt to the available bandwidth (using priorities) at the same time. However, these goals each imply two different transmission orders. To achieve both, the PPS algorithm first subdivides the timeline of the video into a sequence of time intervals using the SDU timestamps. We call these intervals *adaptation windows*[1]. Next, the algorithm transmits the sequence of windows in time order, but the algorithm transmits the SDUs *within* each window in priority order. The rest of this chapter will examine the PPS approach in more detail.

Figure 4.1 shows the conceptual outline of PPS. A pair of re-ordering buffers is employed around the *bottleneck*, for example, the TCP transport. The buffers contain the SDUs of an adaptation window. The algorithm for PPS contains three subcomponents, the upstream buffer, the downstream buffer, and the progress regulator. The upstream buffer admits all SDUs within the

[1]Recall that each *mapper window* results in a single set of SDUs with the same timestamp and differing priorities, hence an *adaptation window* is by its definition a sequence of one or more *mapper windows*.

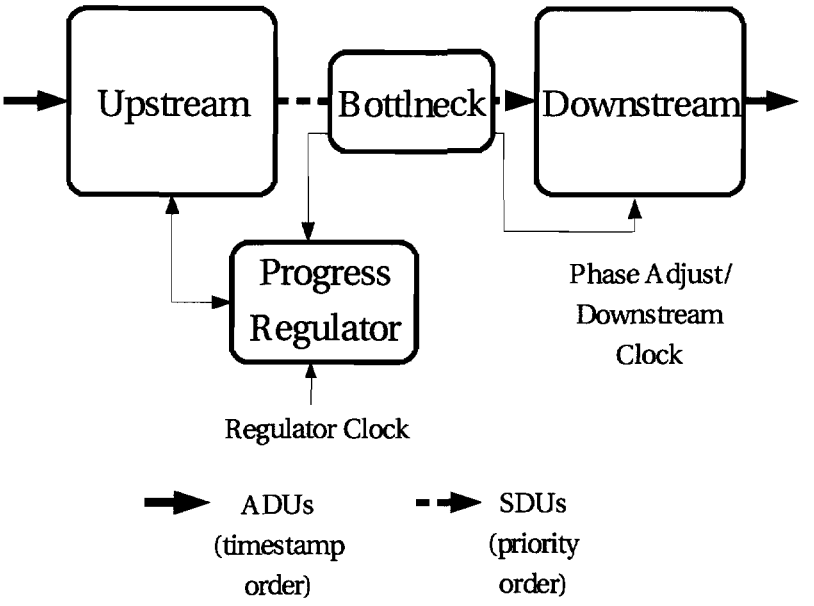


Figure 4.1: PPS Conceptual Architecture

time boundaries of an adaptation window, these boundaries are chosen by the progress regulator. Each time the regulator advances the window forward, the unsent SDUs from the old window position are expired and the window is populated with SDUs from the new position. SDUs flow from the buffer in priority-order through the bottleneck to the downstream adaptation buffer, as fast as the bottleneck will allow. The downstream adaptation buffer collects ADUs contained in SDUs and re-orders them to their original timestamp order. When the regulator advances forward, the entire downstream buffer contents are passed on for subsequent processing, which normally consists of decoding and display. SDUs may arrive late because of unexpected delays through the bottleneck. In the event of a late SDU, since the decoding window to which the SDU belongs has already begun, the late SDU is dropped. When late SDU(s) occur, the progress regulator is notified so that it may try to avoid late SDUs in the future by adjusting the phase between the clocks. The goal is that the downstream buffer receives as many SDUs as the bandwidth of the bottleneck will allow and the rest, which are of lowest priority, are dropped at the server. In this way, the dropping will adapt video quality to match the network conditions between the sender and the receiver.

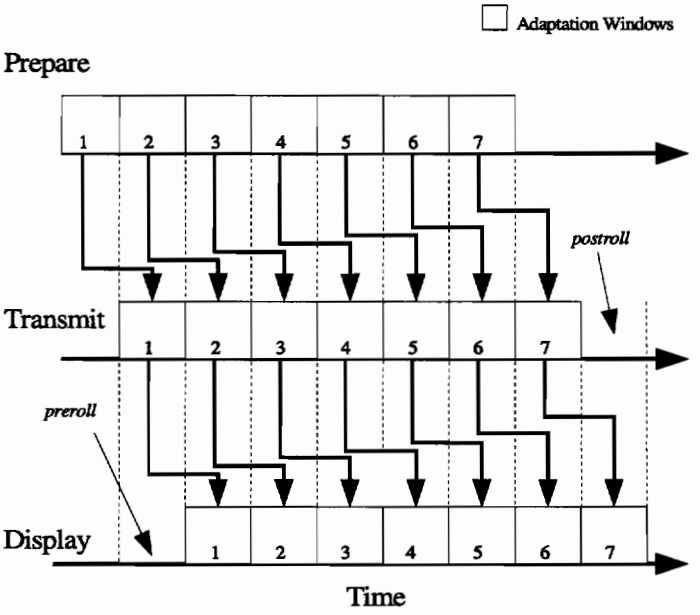


Figure 4.2: PPS Example

As described in the paragraph above, each adaptation window goes through three distinct processing phases. The first phase is *window preparation*, which includes retrieval from the source (file or live capture), prioritization, and re-ordering from timestamp to priority order. The second phase is *window transmission*, where the SDUs are transmitted in priority order. The third phase is *decoding and display*. Figure 4.2 and Table 4.1 give a simple example for a sequence of seven adaptation windows. In the table, each row describes the timing of the phases for the nth adaptation window. Referring to this example, we make some simple observations about how the PPS algorithm works.

First, observe that the timelines of the phases are continuous, so the end value for a given phase in the n^{th} window is the same as the corresponding start value of the $(n+1)^{th}$ window. A gap in the transmission timeline, for example, would imply under utilization of the network. A gap in the display timeline would indicate a failure to maintain the timeliness of the video.

The second observation is that the transmission of a window has to completely precede its display, so in Table 4.1 the transmit end value of a given window must be less than or equal to

Window Number	Prepare		Transmit		Display	
	Start	End	Start	End	Start	End
1	0	1	1	2	2	3
2	1	2	2	3	3	4
3	2	3	3	4	4	5
4	3	4	4	5	5	6
5	4	5	5	6	6	7
6	5	6	6	7	7	8
7	6	7	7	8	8	9

Table 4.1: PPS Example

the display start of the same window. This is a consequence of how re-ordering works in PPS. Transmission of a window's contents proceeds in priority order, but display in time order. On the receiving side, the re-ordering from priority-order back to time-order can not complete until all of the priority-ordered SDUs have arrived from the sender. Hence, the transmission and display phases are strictly sequential for a given window. However, the transmission and display phases do overlap in time in the sense that an earlier window is displayed while the current window transmits, except for the first and last windows[2]. This is a ubiquitous strategy called *pipelining*.

Finally, observe that the first and last windows are special cases. Since the first window has no predecessor, there is nothing to display while it transmits. This period is commonly referred to as the transmission *preroll* period. Similarly, the last window has no successor, so transmission stops when the last window begins the display phase. For symmetry, we'll call the last period *postroll*. The duration of the preroll period represents the main component of startup wait time—the streaming responsiveness, which is a subject of the next section. The postroll period is also significant, as we shall see later in Section 4.3.

4.1 Streaming Scenarios

The main control problem for the PPS algorithm is managing the sequence of adaptation windows: when should each window be transmitted, and what segment of the video timeline should each window contain? When considering the transmission schedule for windows, there are three

[2]In Table 4.1, it happens that display phase for window $i+1$ happens at exactly the same time as the transmit phase for window i, but this will not always be the case in practice

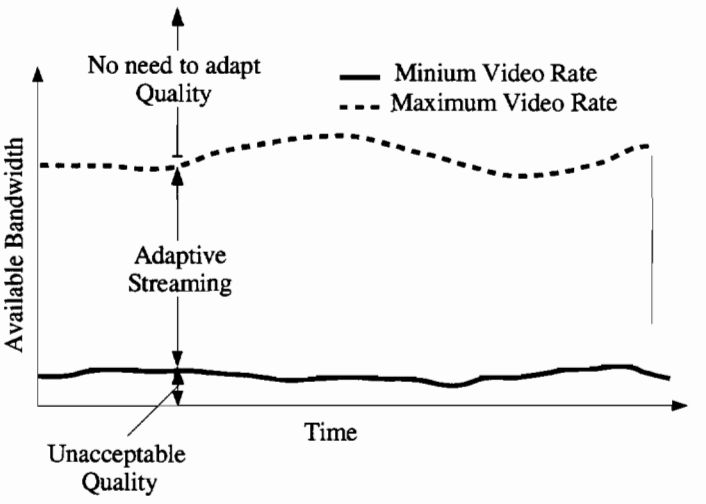


Figure 4.3: Streaming Scenarios

separate scenarios to consider, which are defined according to the relationship between available bandwidth and the adaptation range of the video stream. These scenarios correspond to the three regions of Figure 4.3: adaptive streaming, unacceptable quality, and full quality.

4.1.1 Adaptive Streaming

The main concern of this dissertation is the middle region of Figure 4.3, where available bandwidth is somewhere between the minimum and maximum of the video rates. This case is the main target of our overall approach. In this case, the timing of window transmissions will be more or less linked to the real-time rate of the video. The most basic constraint is that the windows should always be transmitted so as to maintain real-time playout at the receiver.

4.1.2 Unacceptable Quality

The lowest region of Figure 4.3 is when the available bandwidth is less than the minimum required by the video. In our approach, this lower bound corresponds to the scenario where the algorithm finds that it does not have enough time to transmit all the highest priority SDUs. When this scenario arises, a logical recourse might be to declare a fatal failure and abort the stream. Other possibilities would be to play slower than real-time, or to skip whole adaptation windows to try

and keep up with real-time[3]. In the implementation we chose a mixed strategy. If the base layer of a window has started tranmsission, we always complete it, even it means allowing playout interruptions. We call such a scenario *base layer backup*. If we discover that a window is late for display before we even begin transmission, then we skip it entirely.

4.1.3 Full Quality

Finally, the top region of Figure 4.3 is where available bandwidth is more than the maximum rate of the video. On the one hand, one might disqualify consideration of this region, on the grounds that it contradicts the observations in Chapter 1. If available bandwidth is always more than enough, then adaptive streaming isn't needed at all. However, there may be threshold cases where available bandwidth fluctuates above and below the upper limit of the video rate. Also, in a multicast tree there may be some clients that have excess bandwidth while others do not. Thus, it is worth considering what should happen in the upper region.

Work Conservation Options

PPS can use two possible strategies when network bandwidth is abundant. It can be *work conserving* or *non work conserving*.

In the work conserving strategy, as soon as the upstream buffer is emptied, that is, when all SDUs in the window have been transmitted before the window's deadline, then the algorithm advances immediately to the next adaptation window. This strategy is work conserving in the sense that it tries to ensure that the network transport always has data to transmit, which in turn ensures that the stream claims its full share of available bandwidth. However, in this strategy the transmission timeline is advancing faster than the real-time rate of the video. Because the display component of the receiver will only consume video at the real-time rate, the fill levels of its buffers will increase in proportion to the difference between network bandwidth and video rate. In the limit, if bandwidth were essentially infinite, the client would have to buffer the entire movie. The work-conserving strategy is not feasible for certain application scenarios, such as live streaming.

[3]These might be considered degenerate solutions. The semantics of the highest priority level is that the data are essential to maintain the minimum acceptable quality. So these solutions proceed on a path that, by definition, delivers unacceptable quality.

With a live source, it is not possible to advance early to the next adaptation window, since the video for the next window may not have been captured yet. The multicast overlay covered later will also turn out to be an example where the work conserving strategy does not make sense.

The non work conserving strategy, in contrast, is where the PPS algorithm does not advance immediately if the upstream buffer is emptied, but rather the algorithm waits for the adaptation window deadline anyway, effectively pausing network transmission. In this strategy, the client side buffer requirements are bounded to a single adaptation window.

Our implementation supports both strategies or a hybrid combination of the two. A workahead limit is used to specify how much work conservation is allowable. If the limit is as large as the duration of the video, then the algorithm will be fully work conserving. If the limit is less, then the algorithm switches from work-conserving mode to non-work conserving mode when a specified workahead limit is reached. Finally, if the workahead limit is zero, the algorithm is always in non work conserving mode.

4.2 Window Durations: Latency vs Consistency

The duration of the adaptation windows has major implications for the performance of the approach. The minimum possible value for window duration is constrained by the requirements of the video. Clearly, an adaptation window duration must be long enough to hold at least one basic unit of the stream, such as a single video frame. In Section 3.2.1, we described how mapper window boundaries are chosen to avoid the possibility of broken data dependencies in dropping. Therefore, we honor the data dependencies by requiring that adaptation windows contain a sequence of one or more undivided mapper windows. This restriction to whole mapper windows means that the minimum size of an adaptation window will in general be at least several video frames. However, as we described in Section 3.2.1, a single map window (which determines the minimum adaptation window size) will be on the order of one second or less. On the other hand, the upper limit for the adaptation window size could comprise as many mapper windows as we like, upto the full duration of the video itself[4], or we might deliberately constrain the upper limit

[4]The video may not have a bounded duration, if for example, it were a TV style broadcast channel which operates 24 hours a day.

in order to bound the amount of storage required for the streaming process.

Using only primary storage (RAM), a window size of several minutes is certainly feasible. For example, 64 Mbytes holds over 8 minutes of 1 Mbit/s video. With a minimum duration on the order of a second and maximum durations on the order of minutes, the range of possible window sizes is very wide (several orders of magnitude). The best size depends on several factors. As the next couple of sections will explain, the duration of adaptation windows is directly linked to a trade-off between the end-to-end latency of the streaming process and the consistency of video quality.

4.2.1 Latency

To understand the latency-consistency tradeoff, we first analyze the expected latency of the steps of PPS. The overall concern is the end-to-end latency, the time it takes between when a user makes a request, starting the stream for example, and when they see the results. However, this section will focus on the component of latency that is due to the re-ordering employed in the PPS algorithm.

An ideal streaming mechanism would have zero perceivable latency. In contrast, a download has best effort semantics and effectively unbounded latency. However, unlike the ideal, all streaming algorithms will buffer some data, which in turn will add some latency to the overall end-to-end latency. In PPS, the buffers are the adaptation windows[5]. Since one of the principal goals of the algorithm is to maintain the timing of the video, an important part of the design of the PPS algorithm is that it manages adaptation windows in terms of time. That is, the timestamps of SDUs control what goes into the buffers. This differs from traditional buffers, which are managed in units of storage space such as kilobytes, and the relationship between buffer fill and time can be imprecise (since video rates fluctuate). Just as time-sized buffers allow control over timing, they also make it straightforward to predict the latency contribution of PPS.

Figure 4.4 shows how the latency introduced by PPS is related to the window duration T. Recall that each adaptation window goes through three phases. We'll now describe why the total duration of the phases is at most $3T$, and the latency of an individual component (e.g., video frame) through these phases is at most $2T$.

[5]This is not entirely true, Section 4.4 will discuss how PPS buffers, in addition to the adaptation windows, also include a separate component to compensate transport delays (e.g. TCP delay).

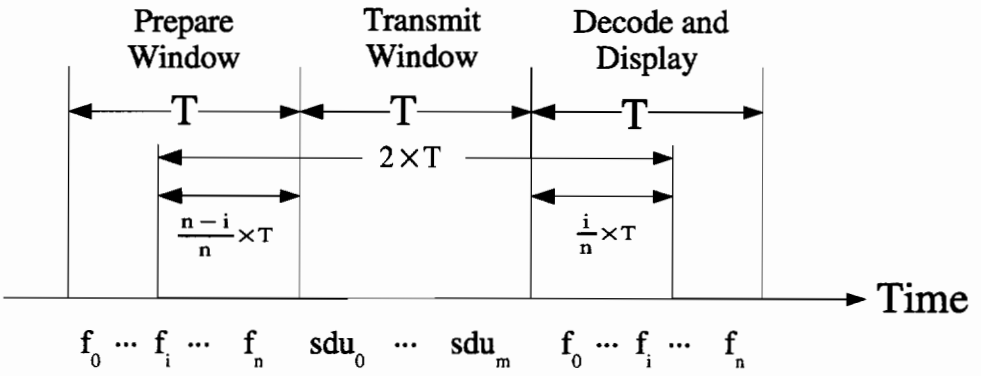


Figure 4.4: PPS Latency: Total latency for contents of a window of duration T is at most $2T$

The first phase is window preparation, where the video data for the window interval are collected, prioritized, and sorted into priority order, at which point they are ready for transmission. Preparation should take at most time T, where T is the duration of the window in the video timeline. For stored content, preparation may take much less than T, given fast enough storage and CPU time[6]. For live content the preparation will require the full time T, since real video frames are collected in real time.

The time given to the transmission phase is a decision for the PPS algorithm. A transmit time greater than T is disqualified since it would not provide data fast enough for real-time play. If the window is given time T to stream, then the transmission timeline advances at the same rate of the video, using all of the available network bandwidth. Hence, we assume for now that the full time T is given for transmission.

Finally, the duration of the display phase would also equal the duration of the window T, assuming normal playout matching the "real-time" rate of the video.

Figure 4.4 also shows the latency for each video frame in the window, in terms of the frame i, assuming each of the phases takes time T to complete. In the preparation phase, the i^{th} frame must wait until all n frames of the window have been converted into SDUs and sorted into priority order, this time is given by the term $\frac{n-i}{n} \times T$. Similarly, only when the last SDU arrives at the receiver can the receiver complete re-sorting back to timestamp order for the decode and display

[6]This is verified in our prototype, where the preparation stage is at least an order of magnitude faster than real time.

stage[7]. Hence all video frames experience the entire delay of the transmission phase, T. In the display phase, the i'th video frame of the window would be displayed only after the previous $i-1$ frames, which is given by $\frac{i}{n} \times T$. Summing the latencies for the three phases, as shown in the Figure 4.4, the total latency for each video frame is at most twice the duration of the adaptation window, i.e., $2T$. Thus, using smaller values of T (shorter adaptation windows) in PPS will reduce its contribution to end to end latency.

In the next section, we consider a technique where PPS will vary the value of T from one window to the next, with the goal of taking advantage of the following observation. Although streaming is in a sense by its definition about small startup latency, it remains that, for some video applications, the end-to-end latency is only temporarily important or perceptible the user. The latency is perceivable in terms of the startup time, or more generally in the response to interactive controls (such as fast forward, rewind, etc.). When video is actually playing, some of the video's timing properties may be very perceptible (such as rate, jitter, audio synchronization), but the end to end latency is not necessarily one of them. In applications such as video conferencing, remote control, or surveillance, the latency may be perceptible or important due to interactive aspects of the application. However, in many other video applications, the users have a completely passive role, such as when viewing video for entertainment, like a movie or an episode of a TV series. For these applications, the latency after play commences is neither perceptible nor important to the user.

4.2.2 Consistency

In PPS, the duration of the adaptation windows also has important implications on the consistency of video quality. Having fewer quality changes is generally desirable from a viewer's perspective. Although adapting video quality to match the network is the foundation of adaptive streaming, it is also true that the goal should be to adapt with least noticeable effect to the user. Making fewer changes is surely one way to make adaptation less noticeable.

[7]The last transmitted SDU could belong to the first video frame of the window.

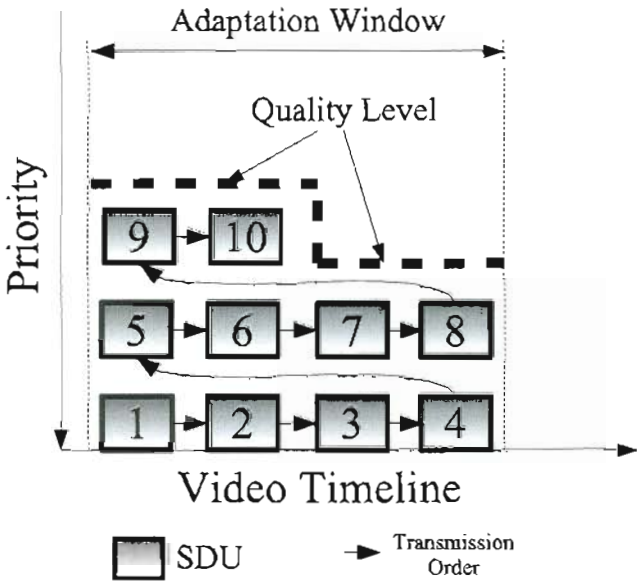


Figure 4.5: Two Quality Levels per Window. Each block represents an SDU, and each vertical column of blocks corresponds to a single map window.

In PPS, the sizes of the adaptation windows have a direct effect on the number of quality changes. Figure 4.5 shows how, for a single adaptation window, the final quality levels are determined by the transmission order used in PPS. The PPS transmission order is such that the SDUs for the window are transmitted primarily in priority-order, and secondarily in timestamp order, as in the figure. The resulting transmission pattern is like filling the rectangle from left to right, bottom to top. In the end, there are (upto) two priority levels that have been reached, hence two quality levels, as shown by the dashed line[8]. Consequently, the total number of quality changes for the whole video will be at most two times the number of adaptation windows in the video timeline. It follows naturally that since longer adaptation windows mean fewer window positions in the PPS timeline, and since fewer windows means fewer quality changes, that therefore longer adaptation windows ensure more consistent quality. In practice the quality is not only more consistent in terms of the number of changes, but also in terms of smaller variances between quality levels.

[8]This assumes that quality for a single priority level is uniform, which is true for our priority mapper algorithm.

4.3 Window Scaling

The previous section established that shorter and longer adaptation windows each have their benefits, which reflects what is probably an inherent trade-off between responsiveness and consistency in adaptive streaming. Indeed this kind of trade-off is likely common to most forms of adaptive control. However, in PPS it is not necessary to restrict all window sizes to the same value. The PPS algorithm includes the option to adjust the window size during the streaming process, which we call *window scaling*. With window scaling, the window duration starts out minimal, so that the startup latency is minimal, and then the window durations grow with each new window as the stream plays. As the window durations get larger, quality changes become less frequent. Compared to a fixed window duration, we will see that window scaling yields dramatically better balance between latency and consistency. However, we must first explain how window scaling actually works, and in doing so, we'll examine the trade-offs that arise. The main questions will be how fast can windows grow, and what are the quality implications?

Window scaling is possible because PPS can transmit the video at a faster (or slower) rate than it will be consumed at the receiver. The consumption rate at the receiver is naturally fixed to the video's "real time" rate, but the transmission schedule is not so constrained. The priority dropping mechanism is what affords flexibility in this respect. Sending a window faster just means that more SDUs might be dropped. In altering the transmission schedule, the PPS algorithm can create (or reclaim) *workahead* in the transmission schedule, which is what allows subsequent adaptation windows to be larger (or smaller). Workahead accumulates whenever the duration of the transmission phase is shorter than the display phase. By definition, the preroll period establishes the initial workahead. With the exception of the preroll window, the accumulated workahead is the upper bound on the duration of each step of the transmission phase. We call the ratio between duration of a step of the transmission phase and the duration of the corresponding step of the display phase the window scaling *growth ratio*.

Figure 4.6 and Table 4.2 describe the timelines for four adaptation windows, where the growth ratio is fixed at 2. Each box in Figure 4.6 represents an adaptation window. The height of the boxes represents the display duration of the window. The top timeline of Figure 4.6 is for the Transmit phase, the rectangular shapes of the boxes in that timeline reflect that the windows' transmission

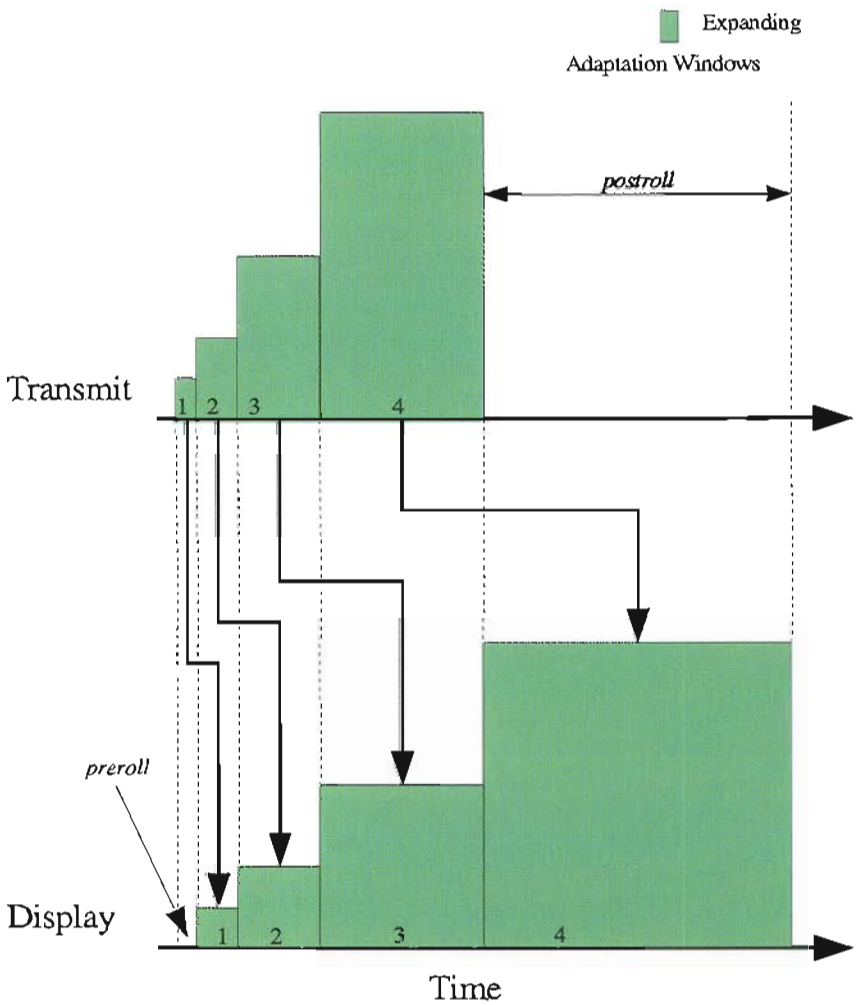


Figure 4.6: PPS with Window Scaling Example

durations are less than their display durations. The Display phase proceeds at the normal real-time rate of the video, so the boxes on the bottom timeline of Figure 4.6 are square. Table 4.1 gives the details of the timelines depicted in Figure 4.6. The group of columns marked *Video* describe the timestamps from a stored video. The *Transmit* and *Display* describe the windows in terms of the streaming timeline as depicted in the figure. In this example, the transmit duration is always the full amount of workahead, so that the display of an adaptation window starts exactly when the transmission ends. This is not the case in the actual implementation: the transmit durations do not usually match workahead exactly because the intervals of video contained within each adaptation window have to be aligned with GOP boundaries.

Window	Video			Transmit			Display		
Number	Duration	Start	end	Duration	Start	End	Duration	Start	End
1	1	0	1	0.5	0	0.5	1	0.5	1.5
2	2	1	3	1	0.5	1.5	2	1.5	3.5
3	4	3	7	2	1.5	3.5	4	3.5	7.5
4	8	7	15	4	3.5	7.5	8	7.5	15.5

Table 4.2: Window Scaling Example: windows grow at 100% rate

From this example we now extrapolate to a more general analysis of window scaling, which sheds more light on the question of quality implications of window scaling.

If the PPS growth ratio is kept constant as in this example, then the sequence of window durations forms a geometric series, whose total is the overall duration of the video. In the example the series is 1 + 2 + 4 + 8 = 15 seconds. Generically, the geometric series is $S = a + ar + ar^2 + \ldots + ar^n$ where S is the total duration of the video, a is the duration of the initial window, r is the growth ratio, and $n + 1$ is the number of windows. Recall that the solution for the sum of a geometric series is $S = \frac{a(r^{n+1}-1)}{r-1}$. Solving this equation for the number of windows $n + 1$, we get $n + 1 = \log_r \frac{S(r-1)+1}{a}$. In words, the number of adaptation windows is logarithmic, where the base of the logarithm is the growth ratio, and the argument to the logarithm is a function of the total length of the video, the growth ratio, and the initial window size. In non mathematical terms, this means that, through window scaling in PPS, the number of adaptation windows grows very slowly relative to the total duration of a video. The analysis could be extended to the case where r might be allowed to vary. For instance, it might be profitable to start with a relatively large value

of r and decrease the value as time goes on. A piecewise version of the analysis here could be used to show the same general results.

The significant implication of this analysis is that window scaling in PPS can cause video quality to become more and more consistent the longer the video plays. This result holds no matter how volatile the video rates and network rates are (within reason).

In the analysis above, we see that the consistency effects of window scaling depend on several parameters: the total length of the video, the initial window duration a, and the window scaling growth rate r. Of these variables, r turns out to be of the main interest. The length of the video is a fixed value that depends on the chosen content. The value of a is the preroll duration, for which there is an incentive to choose the smallest values the video constraints will allow, in order to have the best interactive response (as discussed in the previous section). The question remains: how fast should the windows grow, or, what values should r have? From the analysis of the previous paragraph, larger values of r yield more consistent quality (the larger the base, the smaller a logarithmic value will be). On the other hand, larger values of r have a negative impact on network utilization and average video quality, for reasons we describe next, so arbitrarily large values of r will not be acceptable.

To help explain the negatives of large values of r, we begin by considering the example in Table 4.2. Notice that the sum of the transmit durations is 7.5, while the sum of the display durations 15. In other words, the transmission occurs only for half of the time that video is displayed. If we assume that available network bandwidth is uniform, then our example results in using only one half of the network bandwidth that was available, which would be reflected in the average quality of the video. This level of utilization is unacceptable, as high utilization is one of the primary motivations of PPS and adaptive streaming in general. The cause of the problem in this case is that window growth leads to a very large postfix window, and recall that during the display of the postfix window, PPS will leave available network bandwidth unused.

Figure 4.7 and Table 4.3 give an example which shows how we can modify the window scaling approach to avoid sacrificing so much network utilization. In this example, the windows grow for the first half of the video, and then shrink down again for the second half, significantly reducing the size of the postfix. The postfix interval now represents a very small fraction of the total timeline. As a result, utilization is close to complete, and average quality will reflect the nearly double

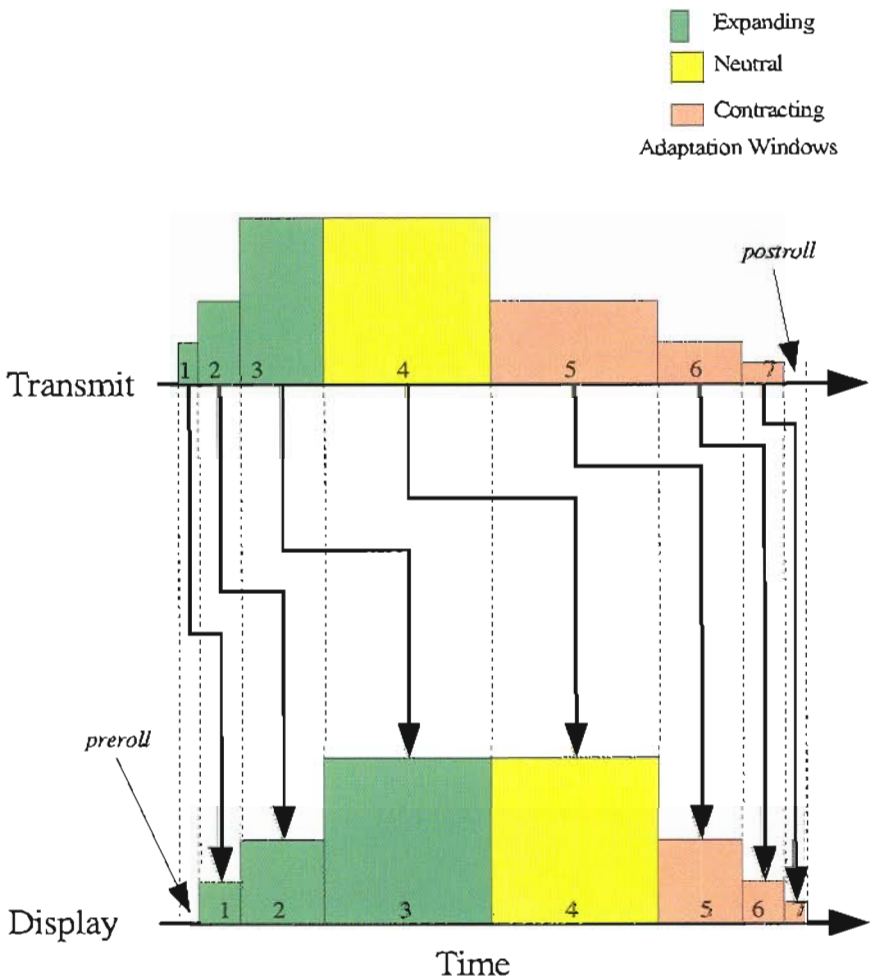


Figure 4.7: Example of PPS with Window Scaling.

Window	Video			Transmit			Display		
Number	Duration	Start	end	Duration	Start	End	Duration	Start	End
1	1	0	1	0.5	0	0.5	1	0.5	1.5
2	2	1	3	1	0.5	1.5	2	1.5	3.5
3	4	3	7	2	1.5	2.5	4	3.5	7.5
4	4	7	11	4	2.5	6.5	4	7.5	11.5
5	2	11	13	4	8.5	10.5	2	11.5	13.5
6	1	13	14	2	10.5	12.5	1	13.5	14.5
7	0.5	14	14.5	1	12.5	13.5	0.5	14.5	15

Table 4.3: Modified Window Scaling Example: windows 1–3 form an expansion phase, window 4 is neutral, and windows 5–7 are the shrinking phase.

average rate compared to the previous example where windows only grow. This improvement in utilization has come at some expense in consistency, because there are now more adaptation windows (approximately twice as many compared to when scaling only grows the window), but the total number of windows still retains a logarithmic relationship with the total video duration. That is, it retains the important property that longer videos will have more consistent quality. This example reflects the strategy for window scaling that is used in the current implementation of PPS, which is as follows.

As Figure 4.7 shows, the PPS timeline is subdivided into three phases: *expansion*, *neutral*, and *contraction*. In the expansion phase, the windows grow at a growth rate r. The expansion phase lasts until a window size limit is reached, in which case a *neutral* phase begins, or when the half-way point of the video timeline is reached. The expansion of the streaming timeline is essentially mirrored in reverse for the final part, yielding a *contraction* phase of similar length to the expansion phase, in which windows shrink at a rate of $\frac{1}{r}$. The three phases provide a balance between consistency, average overall quality, and latency. Now, we turn to the issue of what are plausible values for the growth ratio r.

For the sake of argument, let us suppose for the moment that the video has a constant relationship between data rate and video quality, and let us also assume that the available network bandwidth is constant (denoted as C)[9]. With these assumptions, we can see that the example choice of growth factor of 2 in Table 4.3 has a rather major problem. In the expansion phase,

[9]These assumptions are not true in practice, but the difference is neither here nor there for the point we make in this paragraph.

there are 3 windows that account for 7 seconds of the video timeline, but are given 3.5 seconds to stream, yielding an average video rate of about $C/2$. In contrast, during the contraction phase, there are 3 windows spanning 3.5 seconds of the video timeline and given 7 seconds to stream, so the average video rate for these windows is close to $2C$. Thus the video quality for the expansion phase will be 4 times worse than in the contraction phase. In the general case, with growth rate r, the quality imbalance will be r^2. To keep the quality imbalance between the expansion and contraction phases reasonable, the value of r needs to be much more conservative than 2.

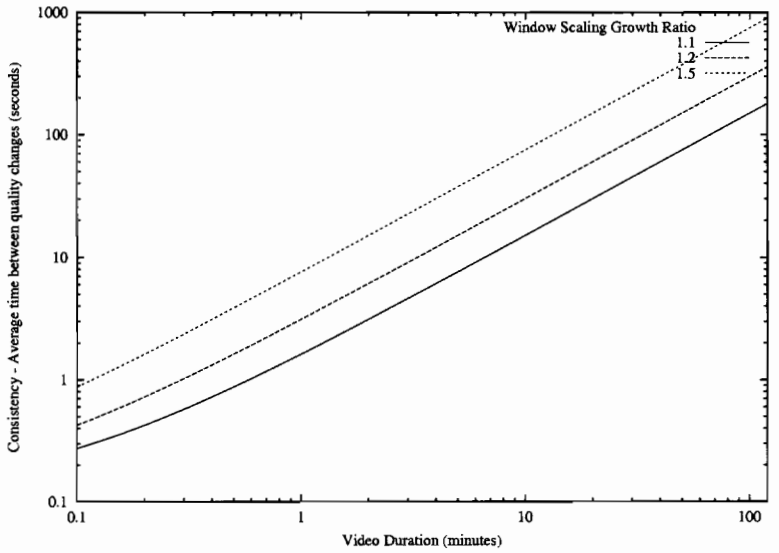


Figure 4.8: Window Scaling and Consistency: This plot shows the average time between quality changes as a function of the total duration of stream. Initial window size is 0.25 seconds. Both axes are log scale.

Fortunately, growth ratios as small as 1.1–1.5 give reasonable results. Let us define consistency to be the average time between quality changes. With window scaling, this average increases as a function of total video duration. Given that quality can change twice per window, the upper limit on the average can be approximated with the following formula: $2\int_{t=0}^{n} \frac{window\_duration(t)}{n}$ where n is the video duration, and $window\_duration(t)$ is the size of the adaptation window at time t in the stream. Figure 4.8 shows this average, for a range of video durations upto 2 hours, assuming that the initial window size is 0.25 seconds, and that the window grows until the half way point of the timeline, then shrinks back down for the second half. Even with a conservative growth rate of

1.1, quality changes are on average approximately 10 seconds apart for a 10 minute clip, and for a two hour movie they would be nearly two minutes apart.

If the user can perform interactive (VCR style) operations, then some modifications to the window scaling schedule will be required. One way to treat interactive operations would be to proceed as if the streaming process were starting over. That is, PPS would reset window size to the minimum size and restart the expansion phase, starting at the point in the video timeline where the interactive operation was initiated. This would be simple to implement, but there is certainly room for optimizations. One might be to attempt to avoid re-transmitting valid data that already made it to the client before the interactive operation occurred. Another optimization would be to continue with the original streaming schedule in the cases where the interactive operation goes backward, restarting the expansion phase only if the interactive operation goes forward. This might be combined with having the player cache video data (to local secondary storage) as it arrives, so that all of the data from previous adaptation windows is available. With these approaches, interactive operations will tend to reduce the maximum consistency, but the general benefit of window scaling (better consistency) will still hold as long as there are significant periods where play is allowed to proceed uninterrupted.

In review, latency and consistency are in conflict with each other. Through window scaling, this conflict can be mitigated in PPS. Using short windows at the start and end of streaming allows interactive operations to be responsive and network utilization to be maximized. Using larger windows elsewhere allows big improvements in average consistency. This technique is not appropriate for a live conversation, but it is feasible for the large class of applications that do not involve two way communication. Even for some live applications, such as watching live sporting events, or the evening news, a few seconds or even a few minutes of latency are unlikely to matter to most viewers.

4.4 Propagation Delay

So far in this chapter, the description of PPS has only alluded at how to deal with delay in the network transport path. For ease of presentation, the examples of the previous sections presumed a zero delay between the sender and receiver sides of a PPS stream. Actual network transports

will introduce significant delay, due to a number of factors. In this section, we now explain how PPS deals with the delay through the network transport.

In Figure 4.1, the outline of the PPS architecture shows two distinct clocks, the regulator clock and the downstream clock. The reason for separate clocks is precisely to cope with transmission delay. In particular, the PPS regulator contains a state variable we call *phase offset*, whose value should be the largest transmission delay PPS expects to experience. The regulator maintains the two clocks so that downstream clock time equals the regulator clock minus the phase offset: $clock_{downstream} = clock_{regulator} - phase\_offset$.

To ensure uninterrupted playout, the downstream side of PPS will set a deadline on each adaptation window. The deadline corresponds to the point when the workahead (see Section 4.3) reaches zero, and the display would run out of video frames to display. The normal expectation is that the last SDU for each window will arrive downstream before this deadline is reached. As long as the phase offset is larger than transmission delay, this will be true. If, however, the phase offset value is understated in error, then as soon as the deadline expires, the contents of downstream buffer will be committed for decode and display. Subsequently, any SDUs that arrive for that window are considered late, and dropped by the receiver. These late SDUs waste network bandwidth because they do not end up contributing to quality of the displayed video. To avoid future waste, the regulator is informed about unexpected delay (late SDUs), and the phase offset is adjusted, causing one of the two clocks to change. We now consider some pros and cons of which of the two clocks to adjust.

4.4.1 Server-side Phase Adjustments

The first strategy for dealing with unexpected delay is to adjust the regulator clock backward, by an amount proportional to the tardiness of the late SDU(s). This will have the effect of causing the upstream side to advance transmission of the current and future adaptation windows earlier than originally expected. The video quality for the current window will be somewhat lower as a result. However, the video quality for subsequent windows will be unaffected. Using this approach, unexpected delays manifest in transient drops in quality. With our implementation, our observation is that such quality effects are often qualitatively imperceptible. Unfortunately, this approach is not appropriate for all applications. With live content for example, it may not be feasible to adjust

the regulator clock, because it can not request video that hasn't been captured yet.

4.4.2 Player-side Phase Adjustments

Another alternative method for delay compensation is to adjust the downstream clock. This approach implies making adjustments to the video playout rate. To avoid timing artifacts in the video and in the audio component in particular, the regulator should gradually adjust the playout clock to reach the new target phase offset. Since computer audio devices do not typically provide a way to alter their timing directly, the player will need to employ digital signal processing algorithms to resample the audio data so that the audio timeline can converge to the new target phase offset. These techniques are not yet implemented in our player, but we do anticipate that doing so would not be difficult.

4.4.3 Improving latency with `MINBUF`

With either of the above two strategies, the value of the phase offset effectively adds to client side buffering, and hence to the total latency of PPS described in Section 4.2.1. In practice, if the network path is saturated with competing traffic, the phase offset typically settles to a value of a few seconds within the first minute of streaming.

Our measurements have shown that the send side socket buffer is responsible for the majority of the transport latency. This can be reduced significantly through a minor extension to the socket API called the `MINBUF` socket option [31]. Roughly speaking, the `MINBUF` option signals the kernel to buffer only as much data as TCP's congestion window requires[10]. With `MINBUF` the phase offset stays much closer to the true network delay between the sender and receiver, and the quality artifacts associated of phase offset adjustments are much less significant. Once streaming is underway, the phase offset tends to stabilize within the first seconds. Meanwhile, the window scaling mechanism takes effect to grow the adaptation windows. As time progresses, the phase offset becomes an insignificant component of the buffering, since it may be on the order of a second, whereas the adaptation window is orders of magnitude larger.

[10]The amount of data buffered is subject to tuning parameters, which can be used to adjust tradeoffs between throughput and latency.

4.4.4 Supporting Interactive Applications

There are some classes of video application, such as conferencing, where the user is continually sensitive to end-to-end latency. The general reason these applications are latency sensitive is that they are interactive. In conferencing, the dialog is based on a sequence of exchanges between the participants. High streaming latency causes the natural flow the dialog to become interrupted, which is hard for people to tolerate. Similarly, surveillance and remote control applications may require that users take some real world response to what they see. For instance, a pilot flying an unmanned aircraft based on a video feed from the aircraft itself requires that the video is as fresh as possible to avoid crashing. Generally, we consider these interactive applications to be outside the scope of this dissertation, however we can make some observations about PPS and interactive applications.

Huang began follow on work to this dissertation to examine PPS and interactive applications [38], the preliminary results show that we measured latencies in the range of 400ms with our prototype. To summarize, the total contribution to end-to-end latency by PPS is controlled by (the sum of) two parameters: the adaptation window size and the phase offset. Other delays relating to capture and decode and display will add to the PPS delay. In relation to other approaches in the literature (e.g., feedback based), the window size component of PPS delay reflects the upper bound on extra delay due to the data re-ordering aspect of PPS. It is worth noting that, as mentioned at the end of Section 4.3, window scaling is inappropriate for interactive applications, because the demand for fresh video trumps the desire to hide quality fluctuations.

Aside from the delay due to re-ordering, the phase adjustment strategy of PPS described in the sections above is probably too simplistic for interactive applications, and leads to unacceptable delay. The problem is that our current approach never attempts to reduce the phase offset, even though it may have overcompensated. In the non-interactive case, this was acceptable because the phase offset delay, even when overcompensted, represented a small amount relative to the adaptation window part of PPS delay. Recall that the window scaling mechanism increases the window size to tens of seconds or even minutes. In an interactive (live) application, the window size would be restricted to reflect the latency tolerance of the application, which might be just a few hundred milliseconds. In this case, the current phase adjustment strategy can easily allow

phase offset to grow beyond acceptable levels. Thus for interactive streaming, it might be more prudent to control the phase offset less conservatively. In particular, the control should work so that streaming latency can be reduced if and when network conditions improve. Also, it might be sensible to set an upper bound on the phase-offset, accepting occassional badput due to spikes in transmission delays. We have implemented the bounded phase offset as an option in QStream, but not the more agressive control. These and other issues of low-latency PPS are the subjects of ongoing work.

4.5 Related Work

There are numerous works in the literature on multimedia streaming in general and quality-adaptive streaming in particular. We have already discussed them generally in Section 2.2, so we do not repeat the references here. However, one rather unique issue the we have treated in this chapter has been what we termed consistency, and the role of our window scaling approach to balance consistency against latency. Our motivation to minimize the amount of quality changes has been based on our intuition and experience with our prototype implementation, QStream. Recently, Zink has performed a a proper human subject study on the perceptual effects of quality changes in video and his results affirm our intuitions [97]. It is also worth noting that Zink incorporated our SPEG implementation into his study, because at the time, it was still the only publicly available implementation of a scalable video codec.

4.6 Summary

In this chapter we have described the design of PPS, our quality adaptive streaming protocol. PPS uses timestamps and priorities to manage the real-time and adaptive requirements of streaming. The basic approach is to subdivide the video timeline into intervals called adaptation windows, and then to transmit the contents of these windows in priority order. The rate of transmission is limited according to the congestion control of the underlying transport protocol. When proper play-out timing requires it, the transmission will move from one window to the next, possible dropping unsent data (lowest priority by definition). We also described how the duration of adaptation windows plays a crucial role in controlling the trade-off between startup latency and consistency.

We then described how window-scaling can be used to manipulate the trade-off so that startup latency is low, yet the quality will still be very consistent on average.

Chapter 5

Streaming for Multicast Overlays

Multicast is a transmission technique that aims to improve the scalability of video delivery to large numbers of receivers. The basic idea is to form a tree structured distribution network in which the interior nodes perform data replication. The structure of the tree serves to reduce the resources required at any single point, in contrast to unicast, which can quickly cause resources near the source of a distribution network to be overwhelmed when there are large numbers of receivers trying to access the same content. In Chapter 2, we described the related work on multicast and some of the outstanding challenges. In this chapter, we describe how we extend the framework we've developed so far (SPEG, the Mapper, and PPS) to provide a solution for multi-rate multicast.

Our multicast extension is called Priority-Progress Multicast (PPM) streaming. The unicast PPS algorithm is adaptive at a very fine-granularity, and can even work well using TCP as the transport. PPM streaming works by implementing the multicast tree as a series of point-to-point unicast PPS sessions. However, the application level processing done in the interior nodes of the PPM tree has significantly lower complexity than for the end-points. We show that Priority-Progress Multicast (PPM) can achieve multi-rate multicast in a TCP friendly manner. By multi-rate, we mean that the bandwidth of the stream reaching each receiver of the multicast is as if there were a unicast between that receiver and the sender. Thus, slow receivers of the multicast do not penalize fast receivers of the multicast. The TCP friendliness results from the fact that each point-to-point connection in the tree is a unicast connection that employs TCP friendly congestion control. In our implementation, this unicast connection actually is TCP. Our experiments have verified that PPM correctly implements multi-rate multicast for broadcasting. They also show that PPM is very lightweight. On commodity server class hardware, our implementation of PPM can support very large volumes of multicast traffic, saturating Gigabit network links. Thus, we do not

expect node stress to be a first order scalability factor for PPM trees.

5.1 Priority-Progress Multicast

The basic Priority-Progress architecture consists of three components: a progress regulator, an upstream re-order buffer, and a downstream re-order buffer. The network link is the conceptual bottleneck residing between the two re-order buffers.

In Priority-Progress Multicast (PPM), we put a distribution tree in place of the single link. The tree consists of multiple network links, connected by PPM forwarding nodes. In this dissertation, we do not address how the topology of the tree is established. Instead, we assume that either the tree topology is pre-established statically, or that one of the recent techniques from the literature is used for dynamic tree construction [2, 11, 81]. Our implementation uses a static tree topology. In PPM, each edge in the tree is a separate unicast PPS session, which as we mentioned earlier is layered over TCP in our implementation. Thus, a multicast forwarder has one incoming (upstream side) TCP flow, and one or more outgoing (downstream side) TCP flows per active PPS session. For the purpose of this presentation, a PPS session can be thought to be analogous to a TV channel in traditional video broadcast. We now describe the operation of the PPM forwarder.

Session Startup

The first task of the PPM forwarder is to handle the start phase of PPS sessions. For the start phase we have two cases, the first being the initial activation of a multicast session due to the arrival of the first session member, and the second case being the arrival of subsequent members who join in on the already active broadcast. The arrival of the first member is the easier one to handle, in that the steps involved are very similar to the case in unicast streaming, only they are replicated as datagrams relayed through the nodes on the path from the source to the sink. Since Priority-Progress sends data in priority order, and the lower priority data generally represents enhancements relative to high priority data, it follows that joining in on an active session can only happen at points where a new window position is started, otherwise the missing high priority data would make all the remaining data unusable. As suggested, the simple solution is to have new members wait until the next window to begin. Since the decode and presentation of data does not

begin until the first window is completely received at the receiver, this means the startup process can take at least one and as long as two whole window periods from the time the member selects a session to the time session display begins.

To improve startup times in multicast, we might implement a more advanced hybrid-startup in the future, which we sketch here. The hybrid version would begin with unicast and transition to multicast. New members are started with an separate unicast that begins with a small window size, the window size then ramps up (using the window scaling option of PPS) until it synchronizes its size and position of the window of the multicast session. Once synchronized, the unicast session is terminated, and the member is switched into the multicast section proper. The goal of this approach is to support rapid "channel surfing" by way of the unicasts, while using relatively large windows in the multicast during the periods of time where the viewer has settled into viewing one particular session. The large windows are desirable for the reason of keeping quality resilient against transient bandwidth changes. After startup, the PPM forwarder enters the main multicast streaming phase.

Window Forwarding

The root of the PPM tree periodically sends window position messages to the PPM forwarder immediately below in the tree (for simplicity of explanation, we assume there is one, but extension to multiple receivers is straightforward). The PPM forwarder in turn, replicates the message along each of its downstream edges (and so on down the tree). The forwarder then begins receiving data units for the window position from upstream. We use a reference counting mechanism to track which data have been forwarded down each of the downstream edges. For each data unit received the forwarder maintains a reference counter, which is initialized to the number of downstream edges. Each data unit and its counter is entered into the head end of a FIFO linked-list data structure. The forwarder maintains a separate pointer into the list for each of the outgoing tree edges, which we call the "out pointer" vector. Each out pointer is initialized to point at list tail. For each connection the forwarder writes the data unit pointed to by the corresponding out pointer. When the write completes, the counter for the data unit is decremented. The new value of this out pointer will be the next item in the list. If the counter decrement reached zero, the item is removed from the list. In the event that the head of the list is reached, the out pointer is null, and output

for the downstream link is (temporarily) paused. A separate list is maintained to track which downstream connections are paused. For every data unit received, this list is processed to resume any paused connections. This whole process continues, with data units arriving from upstream, and writes on each of the downstream links. Eventually, the forwarder receives the message from the upstream regulator indicating the start of the next window position, followed immediately by the first data unit for the new position. At this time, the current contents of the list are flushed, dropping any remaining data units from the old window position, and the new window position message is replicated down each of the downstream links.

5.1.1 SDU Fragmentation

PPM is an example of the store and forward networking model, similar to the model used in the IP protocol. As it turns out, the description of PPM so far overlooks a problem that relates to store and forward. The problem occurs when PPS messages, SDUs in particular, are very large. In concrete terms, a single SDU in PPS can be tens or even hundreds of kilobytes. When SDUs are this large, it introduces the chance that a PPM forwarder may spend a substantial amount of time just receiving a single large SDU. Even though QStream is implemented using reactive style that avoids blocking, it is still the case that a PPM forwarder waits for the entire SDU to arrive before it starts to forward it downstream. In this time, the PPM forwarder can run out of data to send to its downstream children, which allows the downstream connections to go idle, even though network bandwidth is available. It turns out that this happens quite often at the start of each adapation window, because data from older windows have been dropped, and the first SDU must arrive in its entirety before it can be forwarded. This problem can cause serious reductions in PPM's ability to utilize available bandwidth.

To solve this problem, we choose to limit the size of SDU messages by splitting an SDU into a sequence of smaller SDU fragment messages. These SDU fragments messages are limited to a size similar to the maximum segment size of TCP. As a result, PPM is able to forward data very quickly when it is received. Chapter 7 will show that SDU fragmentation allows PPM to achieve full network utilization.

5.1.2 Multicast Flow Control

Another issue not addressed in the PPM algorithm, as described so far, is conservation of upstream bandwidth. That is, what if the upstream link is significantly faster than all of the downstream links? The algorithm above allows the upstream link to proceed at full rate. In the case where all the downstream links are relatively slow, a large proportion of SDUs received from upstream will be dropped before they make it to any receiver. This issue arises in PPM because it partitions the path between the source and the client into separate transport (TCP) sessions, one for each edge of the tree. By contrast, in unicast PPS, all of the dropping happens at the server. To prevent such waste, PPM includes a (application level) multicast flow control option.

PPM flow control works as follows. Each PPM forwarder keeps a count of SDUs in its SDU queue that have not yet been transmitted through *any* of its downstream edges. This value is called the *SDU fill*. When the SDU fill exceeds a *high water* threshold, the upstream link is paused. To pause the link, a message is sent upstream to the parent, which causes the parent to stop sending SDUs. The parent may delay this pause if it has base layer SDUs to send. Also, the start of the next adapation window (signalled by the arrival of a corresponding message) always immediately unpauses all links. While the upstream link is paused, messages will continue their transmission on the downstream links, causing the SDU fill level to drop. When the SDU fill drops below a *low water* threshold, then the PPM forwarder will resume the upstream link, by sending a resume message upstream. This flow control mechanism is naturally recursive, since pausing the link to the parent may cause the parent's SDU fill level to rise, and hence the parent may send a pause message further up the tree, and so on, until the final link to the root of the tree is paused. The resume messages will propagate in the same way.

An alternative approach to PPM flow control would be to use the transport's flow control more directly. In this version, rather than sending a pause message upstream, a PPM forwarder would cause the pause simply by halting reception (reads) of messages from upstream. The main advantage of this option over using explicit pause/resume messages is that it involves fewer changes to the PPM algorithm, in particular the algorithm at the root is unchanged. However, this approach is deficient due to its effect on end-to-end latency. If a PPM node suspends message reception as suggested, then eventually the flow control mechanism in the transport would stop upstream nodes

from sending further messages, which is the desired effect. Unfortunately this process completes only after all the upstream receive buffers become full, for every edge on the path to the root. When the PPM forwarder resumes accepting messages, all of those buffers will have to drain, which would often cause significant latency. In the worst scenario, should the root of the tree need to advance to the next adaptation window during the period where all the receive buffers are full, then it will be unable to transmit the window start message until after the buffers drain. This may cause many of the buffered SDUs to be declared late at the leaves of the tree. The phase offset will adjust eventually, but the resulting phase offset would be much larger than truly necessary. In contrast, with the explicit pause/resume flow control scheme, transport receive buffers should stay close to empty, as the Priority-Progress algorithm always consumes incoming messages as soon as possible. Also, the root of the tree will send window start messages regardless of the flow control. Therefore, the flow-control should make no difference to the phase-offset. Thus PPM, uses explicit flow control messages because they do not impair end-to-end latency.

5.2 Related Work

In section Section 2.2.6, we gave an overview of the literature on adaptive multicast. PPM distinguishing features are that it supports multi-rate adaptation of video, and since it can be implemented using TCP, it is TCP friendly by definition. Furthermore it conserves bandwidth at the top of the tree so that it only expends bandwidth at the source that will be used by at least some of the receivers.

More recently in the literature, a new development in multicast has been emerging, namely, multicast approaches which go beyond a strict tree structure. Systems such as SplitStream [9], CoopNet [65], and Bullet [51] use multi-path multicast techniques, either through multiple trees (CoopNet and SplitStream) or meshes (Bullet). These multi-path approaches have advantages in light of the greater potential for nodes to be unreliable when the trees correspond mostly or entirely of end-hosts. As these systems are all very new, a comprehensive comparison of them remains an open problem.

5.3 Summary

In this chapter, we described PPM which uses the basic PPS approach to construct a multicast overlay. In PPM, each edge of a multicast tree is a separate PPS session. Since the PPS server at the root of the tree transmits data into the tree in priority order, the PPM forwarder can do data dropping in a simple, first-in, first-out fashion. PPM uses the arrival of a new adaptation window as the trigger to drop data for slower receivers. PPM also includes an application level flow control to ensure that the subdivision of the path into separate sessions does not lead to wasted bandwidth in the higher parts of the tree.

Chapter 6

The QStream Implementation

The previous two chapters were intended to provide the reader with a basic understanding of how Priority-Progress Streaming (PPS) and Priority-Progress Multicast (PPM) work, and to explain the important performance trade-offs and how they can be controlled.

The purpose of this chapter is to present a concrete description of the PPS and the PPM algorithms, based on our prototype streaming system, QStream. This description provides pseudo-code and commentary at a level of detail which is greater than the previous chapters, yet still significantly more abstract than the actual QStream source code. In this chapter, we also elaborate on how the software developed for this thesis constitutes a framework for adaptive media streaming.

The framework we present, and its realization in QStream, is divided between parts that are implemented as directly re-usable software components (libraries) and those which are re-usable as a kind of design pattern. This division in the framework falls roughly along the line between the real-time and quality-adaptive aspects of the streaming application. Our libraries are called QSF (Quasar Streaming Framework) and GAIO (Asynchronous IO). They provide support for the conventional real-time aspects of the application. The quality-adaptive parts of our framework (SPEG, the Mapper, PPS and PPM) are implemented in the QStream application, but are not packaged as libraries or components that would be immediately re-usable in other applications. However, we do feel that Priority-Progress illustrates a relatively clear design pattern applicable to wide range of applications. We have released all of the QStream software under open source terms, as one way to promote re-use of the entire framework. The next section describes the libraries in greater detail, and the remaining sections of the chapter describe the QStream application and the details of the PPS and PPM algorithms (see Chapter 3 for the descriptions of SPEG and the

Mapper).

6.1 Quasar Streaming Framework

This section will describe the approach to real-time programming used in QStream, and the two software libraries we have developed to support the approach. We begin by describing some of the motivating design requirements of adaptive media streaming, which have to do with concurrency and timeliness.

6.1.1 Challenges: Concurrency and Timeliness

There are multiple levels at which concurrency arises in media streaming. A streaming server, like most network servers, has one obvious concurrency requirement in that a server should be able to maintain multiple sessions to distinct and unrelated network clients, all at the same time. We call this type of concurrency inter-session concurrency. Media streaming also requires the ability to deal with intra-session concurrency, because a single streaming session consists of separate control and data communications planes. The control plane implements user actions, such as the so-called VCR actions (start, pause, fast forward, rewind, etc.). The data plane implements the continuous parts of the session, such as the video stream. In addition to the concurrency between the data plane and the control plane, the data plane is typically also internally concurrent. A video player, for example, has to manage video rendering and audio playback at the same time. Concurrent programs are notoriously tricky to develop because there are types of errors that are easy to make in concurrent programming that do not exist in sequential programming. Some well known examples of the unique types of errors that arise in concurrent programming are race conditions, deadlocks, and livelocks.

Conventional servers, such as web servers or database servers, have similar concurrency issues, but their main performance concern is optimizing overall throughput of the server. A video streaming system also requires high throughput, but additionally it requires accurate timing for the time sensitive elements of the streaming process. For example, a streaming system has to deal with maintaining audio-video synchronization, which is easy to spot when it goes wrong (but not necessarily easy to correct). With all their concurrency requirements, Video players are often

used as the example application for real-time systems research. Streaming video systems add the challenges of distributed systems to the real-time challenges of regular video players.

Given the objective of developing a video streaming system which addresses concurrency and timing issues, there are several flavors of programming model to choose from. Three of the most well known programming models are preemptive multitasking, cooperative multitasking, and event driven. Various hybrid combinations of these models are possible (and do exist). Our implementation of PPS has evolved through various architectures representing different choice points in space of these three models. The current implementation is based on a flavor of event-driven model, referred to as *reactive* programming.

6.1.2 Reactive Programming

Reactive programming is so called because it places emphasis on the notion of programs that are driven by external events (which originate ultimately from real-world sources) and generate real world output in response. Reactive programming is a natural fit for building real-time systems when clocks are included in the set of event sources. The reactive model has been used very successfully as the basis for time-sensitive applications, especially those in embedded systems [4]. The model is based on state machines which are driven by the arrival of the external events. The ideal realization of the model would be a state machine which responds with optimal timeliness to all events. To achieve the instant execution property of an ideal reactive program, the program must have two properties: the first is that all IO operations are asynchronous, and the second is that CPU speed is sufficiently fast that computation times are a non-issue.

The traditional IO APIs provided in general purpose operating systems are synchronous, which is to say that an IO request returns only after its result is complete. With asynchronous IO, rather than issue an IO and wait for a completion, the IO is requested, and if it's completion is not immediate, then the completion will generate a new separate event in the future. Dividing the IO into two parts, one for issuing the request and the other for handling the result, is generally harder to program than using a synchronous IO API. The payoff is a much better approximation to the instant responsiveness the reactive model aspires to, because asynchronous IO allows the application's state machine to respond to new events while other IO requests are pending completion.

If we were free of practical resource constraints, we could imagine that achieving the ideal

of optimal responsiveness would mean that state actions execute instantly (in zero time), so the timing of the system would depend solely on the arrival of the external events. In the limit, we could consider the CPU to be infinitely fast, so that all computations take zero time. In practise, it suffices if the CPU time available for execution of event handlers is such that event response times stay within acceptable bounds. Our implementation relies on the application to bound the time of individual state actions, and hence preserve overall timeliness. Our implementation does not enforce responsiveness, in particular, it does not employ pre-emptive state scheduling. This requires that the application programmer has some understanding of the CPU requirements of the state actions. If a state does require so much computation time that it would impede the programs ability to maintain tolerable response times, then the state should be subdivided into smaller states. Often this technique is used in connection with potentially long loops, by scheduling successive iterations of a loop as separate events.

In our case, we implement the reactive programming support within an application through GAIO and QSF, the pair of support libraries we mentioned earlier. Although we've concentrated at the application level, the effectiveness of our real-time support also relies on the service provided by the OS kernel. In [30], we evaluate the Linux kernel's support for time sensitive applications. Briefly, we can summarize as as follows.

6.1.3 Kernel considerations

There are three main concerns with kernel performance for a time-sensitive application: scheduling policy, accuracy of timer facilities, and responsiveness of kernel (how quickly it can act on scheduling decisions). The first, scheduling policy, has to do with how the kernel allocates CPU time between various running processes, that is how it makes the decision to switch to the context of a target process. The second concern, the timer mechanism, has to do with how accurately the kernel can implement application specified deadlines (this accuracy is independent of what competing applications may be running), which can be one of several reasons the scheduler might switch to a target process. The last of the three concerns has to do with how responsive the kernel is in actually effecting scheduling policy. Responsiveness is independent of the scheduling policy, but has to do with granularity issues within the kernel. The evaluation in [30] shows that Linux can provide very accurate timers and that the kernel can execute with fine granularity, hence providing

a high degree of responsiveness. The Linux kernel developers generally maintain two versions of the kernel at a time: the stable branch (even numbered minor version) and the development branch (odd numbered minor version). At the time of writing, the stable development tree of the standard Linux kernel (version 2.4) requires some modifications to achieve the timer accuracy and fine granularity properties, as described in [30]. The development branch of the Linux kernel tree (version 2.5) has already incorporated adequate changes. The development branch is expected to transition into the new stable version in the near future. The issue of scheduling policy, which has been a principle issue of real-time scheduling research, remains the more open research area. However, our approach is less prone to the effects of scheduling policy, in the following ways.

One of the key properties of our implementation of reactive programming is that we generally strive to minimize the number of threads or processes used within the application. Although we've just described how media streaming has many concurrent aspects, we avoid the natural temptation to map those aspects into a multi-threaded application. In designing QStream, we've considered multi-threading strictly as a last resort, to be used when there is no other choice. The philosophy behind this policy is simple: we wish to keep as much control over timing as possible within the application. Hence, our approach tends to limit the importance of the kernel CPU scheduler so that it only matters in relation to how other applications may impact our timeliness. In contrast, a multi-threaded application architecture tends to depend on the kernel CPU scheduler even in the complete absence of competing applications. Our approach also affords a pragmatic option when competing applications do exist, that is, running the streaming system with real-time priority. This is a kind of brute-force solution, in the sense that it doesn't go very far in a scenario with several time sensitive applications. However, although it is not a panacea, the single-threaded real-time priority architecture is reasonable in many typical usage scenarios. Alternatively, one can consider it as a proof of concept demonstration of what could be achieved with an OS that provided direct support for reactive programming in the style of the GAIO and QSF APIs. For example, StreamPlay (the QStream player) can effectively manage two or more video sessions at the same time on a single display[1], without noticeable timing artifacts in video or audio. We are unaware of any other publicly available Linux video solution (streaming or otherwise) that is

[1]Provided an adequately fast CPU.

capable of this at this time.

6.1.4 GAIO

The GAIO library provides the core API for reactive programming in QStream (using the C language) consisting of the following services:

- Asynchronous IO primitives, which are emulated over the nonblocking file API.
- A primitive to schedule events for execution at a given time deadline.
- A primitive to schedule an event for execution immediately (as soon as possible).
- An event dispatcher that schedules the handlers for the mix of IO completion events, deadline events, and immediate events. GAIO allows the application to prioritize all events so that the application has some control over the order of event execution.
- A worst case execution time (WCET) profiler.

6.1.4.1 Event Dispatcher

The GAIO event dispatcher is the core of the application's state machine. The application calls GAIO and QSF primitives, providing a event-handler callback parameter (a C function pointer) to be invoked when the requested action is complete (the IO has completed or the deadline expired). Primitives that always complete immediately omit the callback. The event dispatcher also ensures that executions of event handlers are atomic; i.e., every time an event handler is dispatched, it is allowed to run to completion before another handler can be dispatched. We think this property makes it easier to avoid race conditions in the application[2]. The event handlers contained in an application constitute the states of its state machine. GAIO allows events to be prioritized so that order of execution can be tuned in cases where multiple events are ready to run.

[2]Another reason for making the handlers atomic is to bound the size of the application stack. Earlier versions of GAIO allowed the dispatcher to make nested invocations of event handlers, and under some workloads we could cause the stack to grow so large that it corrupted other program data structures.

6.1.4.2 IO Events

The IO primitives provided by GAIO are relatively similar to their counterparts in the standard Berkeley Sockets API, with the addition of asynchronous semantics. The difference between nonblocking and asynchronous APIs can be subtle. The primitives of a nonblocking API always return immediately, with a special error type (*EAGAIN*) if an immediate result is not possible. An asynchronous API always returns immediately, but the completion status is always delivered later (usually by a function callback). The difference is that calling an asynchronous primitive commits the application to the request, whereas the non-blocking version requires the application to re-issue the request if the non-blocking IO returns the *EAGAIN* error. The non-blocking version only makes sense if there is a way to check later when the IO should be re-issued, as with the *select()* and *poll()* primitives in the case for network sockets. However, these primitives do not apply to filesystems, since filesystem IO only occurs in response to committed requests. Hence, the asynchronous style of API semantics are slightly more general than the non-blocking style[3].

With GAIO, the application instantiates a separate GAIO object for each file (unix file descriptor) it uses. GAIO allows the application to specify dispatching priority on a per GAIO object basis. For example, in the QStream video player, the GAIO object for the audio device gets assigned the highest priority, since audio interruptions are very important to avoid.

6.1.4.3 Immediate Events

GAIO also provides a primitive to schedule an event handler of a specified priority for execution as soon as possible, meaning after any other outstanding events that have higher priority. As described above, this allows the program to avoid long running computations that might hamper the event dispatcher's ability to respond to other events, by dividing the computations across multiple events. For example, if a sequence of iterations of a long running loop that are scheduled as separate events, the loop may be interrupted if another higher priority event occurs, such as an IO completion or a deadline expiry.

[3]Although standardized by POSIX, asynchronous APIs remain un-supported in many OS kernels, which is one of the reasons we developed GAIO.

6.1.4.4 Deadline Scheduled Events

The GAIO dispatcher provides a timer primitive that allows the application to schedule an event handler for execution at a set deadline time. The deadline can be specified as an absolute time, or time relative to the current time. When the deadline expires, the event dispatcher will execute the handler as soon as possible. All other events have lower dispatching priority than those with expired deadlines.

6.1.4.5 WCET Profiler

Since in our reactive approach, the overall timeliness of an application depends on limiting the time in any single event handler, GAIO provides a tool to assist in diagnosing long running handlers. The tool is a Worst Case Execution Time (WCET) profiling facility. When this is enabled, GAIO tracks the duration of every event handler it dispatches. At the end of a test, GAIO can print a sorted list of the longest running event handlers, identified by the function name of the handler. GAIO uses a feature of the GNU bfd library to translate pointers (callback parameters) to their symbolic name. Enabling the facility typically adds a 10-20% CPU overhead to the program. Using this facility, the application developer can identify when an event handler might need to be restructured to improve the application granularity. In QStream, our subjective assessment was that application timeliness seemed very good when the largest WCET value was kept below 1 or 2 milliseconds.

6.1.5 QSF

In addition to GAIO, QStream includes a second library called QSF (Quasar Streaming Framework) that provides services that are more specialized to network streaming applications. The services provided by QSF are the following:

- Simplified primitives for establishing network connections (initiate and accept connections).
- Primitives for message passing style communication over QSF connections.
- A primitive to safely enable real-time OS scheduling, which includes the feature that it creates a separate watchdog process.

- Logging and tracing primitives, to help in understanding the dynamics of the program executions (for debugging and performance analysis).

6.1.5.1 Networking and Message Passing

The GAIO network primitives provide the same level of generality as the underlying Berkeley API upon which they are implemented. Thus, one could use GAIO to implement most kinds of network application, such as a web servers or file servers, using standardized protocols. The goals of QStream are not so general. QStream is an experimental platform for adaptive media streaming. Consequently, we were not concerned with compatibility with existing protocols, but rather in developing a prototype PPS protocol. Initially, we developed PPS directly with GAIO, but we discovered that some of the code was verbose and repeated within several QStream programs. In order to make the most salient details of PPS most prominent, we decided to develop a higher level API for networking which more directly supports the message oriented style of the PPS protocol[4]. QSF provides a generic API for message oriented protocols, of which PPS is one instance. QStream contains two other message based protocols which are implemented using the QSF API, one used by the MxTraf traffic generator, and the other used by the Monitor. The Monitor and MxTraf are applications provided with QStream that will be described later (see Sections 6.6 and 7.2.2).

The goal of the QSF network primitives is to provide a simple API for sending and receiving messages. These primitives require that all messages share a generic message header. Figure 6.1 shows the generic message header defined by QSF.

```
QsfMsg {
   Integer length;
   Integer type;
   Integer magic;
}
```

Figure 6.1: QSF Message

The *length* and *type* fields provide the essential information necessary to implement message

[4]QSF started with the networking support, the other features came later.

oriented communication over a (TCP style) reliable, byte-stream session. The *length* field indicates the size in bytes of the message body that follows the header. The *type* indicates the what kind of message is contained in the body. The *type* values are application specific. The *magic* field is for debugging purposes, and it is used to detect if basic framing of messages has been corrupted[5].

On the sender side, QSF provides helper functions for message creation and initialization, as well as the primitives for sending the message. On the receiving side, QSF provides a complete message dispatching facility. For each QSF session, the application registers a set of handler functions, which is indexed by message types. The QSF dispatcher then handles the low level details of receiving the messages and dispatches the appropriate handler for each complete incoming message.

Since QSF is built on top of GAIO, the application can safely mix GAIO and QSF primitives. For example, StreamPlay uses QSF for the PPS connection to StreamServ, but it uses GAIO primitives to do asynchronous IO on the sound card device.

6.1.5.2 Logging and Tracing

Although perhaps mundane, application logging (instrumentation) is nevertheless a critical facility for debugging and maintenance. The QStream programs have extensive amounts of instrumentation in two forms. The first form of instrumentation generates debugging messages to an application log. The second kind of instrumentation sends data to the QStream monitor for real-time visualization in the *gscope* software oscilloscope (see Section 6.6).

The QStream programs accept a command line argument which enables the generation of the debug messages. One of the main benefits of the QSF logging facility is that its messages contain a wealth of human readable time references.

Figure 6.2 shows an example of the output from StreamPlay when the debug option is enabled. The details of the messages are application specific, but some aspects are generic to QSF logging. Each message has a prefix which is generated by QSF on behalf of the application. The prefix can contain the session number, absolute time, session time, and a time delta, as in the last line of

[5]The value of *magic* should be *0x1337*.

```
0: 15:12:43.908909  0.003091 connected, socket pair=127.0.0.1.4005,127.0.0.1.33808,
                             starting recvs
0: 15:12:43.909005  0.000096 sending open file request:file=/mpgs/angels
0: 15:12:43.932743  0.023738 Open ok, uuid=20142b06-c802-4373-86e9-d1db5ed8afbe,
                             video duration= 00:04:10.992000, hsize=704, vsize=304
0: 15:12:44.056734  0.123991 audio fd is 10
0: 15:12:44.056780  0.000046 sending start stream
0: 15:12:44.173926  0.117146 stream start:origin=15:12:44.173870:
                                    stream time= 00:00:00.000000
0: 15:12:44.174009  00:00:00.000139 0.000017 win 00: vid range= 00:00:00.000000-
                                                              00:00:01.876874
                                                   xmit end= 00:00:01.706078
                                                   num_sdus=32 num_base_sdus=2
                                                   pictures=45
0: 15:12:44.174098  00:00:00.000228 0.000089 win 00: schedule decode start
                                                   at  00:00:01.706078
0: 15:12:44.174116  00:00:00.000246 0.000018 win 00: recv first frag length=412
0: 15:12:44.174148  00:00:00.000278 0.000032 win 00: recv cont'd frag length=6732
0: 15:12:44.174193  00:00:00.000323 0.000045 win 00: recv cont'd frag length=8

...
```

Figure 6.2: Example of QSF debug logging. This log fragment comes from StreamPlay, it shows the initial sequence of events in a PPS session.

Figure 6.2. The time delta is simply the difference between the absolute time of the current and previous message in the log. A very large value of the delta can often provide a quick hint to help in diagnosing timing related problems.

The remainder of the log message is supplied by the application, but QSF does provide some helper primitives, for making human readable time values, to ease the formatting of the application part of messages. These helpers make it easy for the application to format time values in the same format used in the prefix: *hh:mm:ss.uuuuuu* (hours, minutes, seconds, and microseconds) . We have found the debugging messages to be invaluable during the development of QStream.

6.1.5.3 Safe Real Time Scheduling

As described earlier in Section 6.1.2, our implementation of the reactive model is done at the user level, but the kernel CPU scheduler still has the ultimate control over the timing of the program. Like most general purpose operating system kernels, Linux does offer some basic support for real-time applications. The Linux scheduler allows applications (with root privileges) to specify a real-time scheduling priority, which assures the application will always get scheduling preference over non real-time processes. Linux also implements the POSIX standardized *mlockall()* system call, which allows a process to pin all of its virtual memory pages into physical memory. In Linux,

if an application runs at real-time priority (with no other real-time competitors) and has all of its pages pinned, then it will have very tight control over its timing.

However, running applications at real-time priority is commonly and rightfully considered very dangerous, because real-time processes can effectively freeze the entire system if they enter a non-terminating loop—usually due to a bug (a livelock condition). Because of the danger, the Linux kernel strongly discourages casual use of its real-time facilities by only allowing processes running as root to enable real-time priority or to pin pages with *mlockall()*.

One technique to reduce the danger of running real-time is to use an application-level watchdog. A watchdog is a separate helper process that also runs at real-time priority (higher than the main process) with the sole purpose of detecting if the main process has livelocked the system. If the watchdog detects livelock, it will terminate the main process. The technique works by having the main process emit heartbeats to the watchdog on a periodic basis[6]. If the main process erroneously enters an infinite loop, it will cease to emit the heartbeats, which in turn will be detected by the watchdog. QSF provides a primitive that enables real time priority and also automatically establishes the watchdog process, taking care of all the details of the watchdog's operation on behalf of the application.

6.1.6 Summary

The QStream implementation is a full realization of our framework for quality-adaptive streaming. Our description of the implementation is divided along the conceptual line between the real-time aspects of the problem and the quality-adaptive aspects. This section has been concerned with describing the real-time support in our framework, which is embodied by a pair of libraries, called GAIO and QSF. GAIO provides the core facilities for real-time with the reactive programming model, including an event scheduling and dispatching facility and primitives for asynchronous IO. QSF complements GAIO by providing specific support for streaming applications, such as easy setup of network sessions and application message passing over those sessions. In the following sections, we describe the implementation of the applications included in QStream, focusing on the details of the algorithms for the Priority-Progress protocols. Along with SPEG and the Mapper

[6]We use a unix pipe to communicate the heartbeats, once per second.

(described in Chapter 3), the Priority-Progress protocols realize the quality-adaptive duties in our framework.

6.2 QStream Architecture and PPS Message Protocol

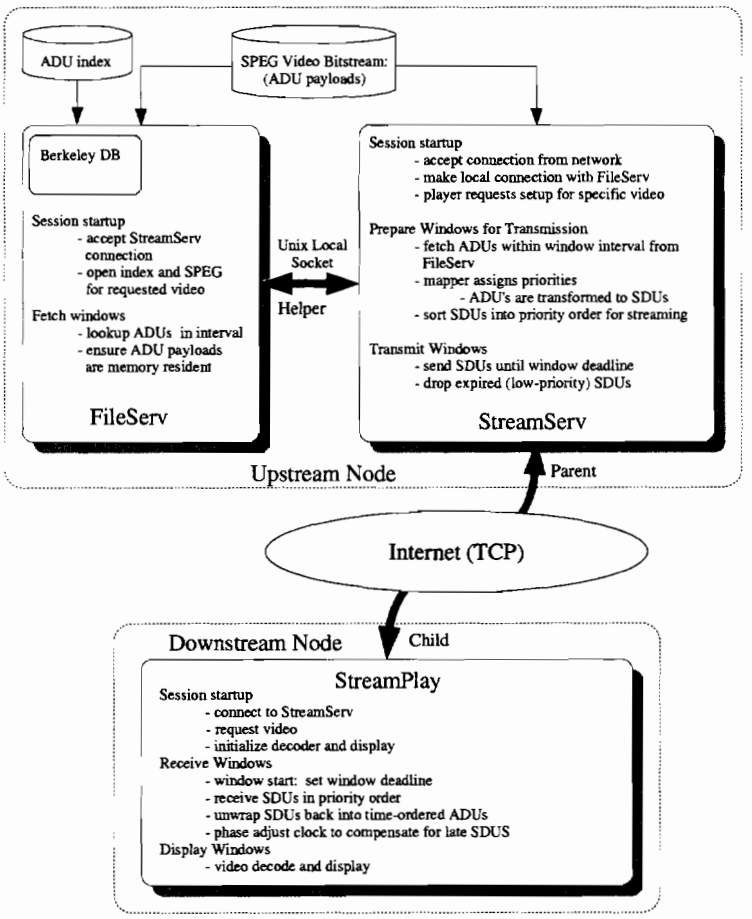


Figure 6.3: QStream in a Unicast Configuration

Figure 6.3 shows the QStream architecture in a unicast streaming configuration. The upstream node stores two types of data per video: a video bitstream and an index. In QStream, the format of the bitstream is SPEG (see Chapter 3). The index uses Berkeley DB B-trees to provide time-based lookups into the bitstream. The index supports efficient assembly of the adaptation windows in

the PPS algorithm[7]. QStream provides a pair of programs, not shown in the figure, that convert MPEG-1 video files to SPEG and generate the index (offline).

The upstream node consists of a pair of threads, called FileServ and StreamServ[8]. Most of the work of the PPS algorithm is done by StreamServ. To help keep good control over timing, StreamServ restricts itself exclusively to using non-blocking IO. Our target implementation platform is Linux, which (at the time of writing) provides only synchronous (blocking) access to filesystems. To isolate StreamServ from operations that could potentially block, a separate QStream thread called FileServ is used to perform all filesystem accesses needed for each session. StreamServ and FileServ communicate with each other through QSF and GAIO, in this case layered over Unix local sockets, which support the same non-blocking API as network sockets (over TCP).

The bottom program in Figure 6.3 is *StreamPlay*. StreamPlay implements the downstream part of the PPS algorithm (see Figure 4.1), video decode, and video display.

Another QStream program, not shown in the figure, is the QStream remote monitor, hereafter referred to simply as the *Monitor*. All of the the QStream programs contain extensive internal instrumentation which is used for generating both online and offline traces of various aspects of QStream performance and the PPS algorithms. The online versions of the traces are visually presented in real-time using a graphical software oscilloscope called *gscope* [32]. The offline versions can be plotted graphically using software such as *gnuplot* [90].

Compared to the conceptual architecture presented in Figure 4.1 of Chapter 4, the concrete architecture of QStream presented in this chapter is somewhat different. As one might expect, the upstream component of the conceptual architecture is realized in StreamServ and the downstream component in StreamPlay. As the following sections will reveal, the functionality of the PPS progress regulator is mainly contained within StreamServ, but a few aspects (mostly relating to maintenance of the phase offset) are contained in StreamPlay.

[7]Random access seeking functions are also easier to support with the index.

[8]FileServ and StreamServ were initially separate programs. They were later merged into a single, dual-threaded program for reasons of ease of use.

6.2.1 Naming conventions

The following sections will describe the algorithms of the QStream implementation of PPS. Each algorithm contains several functions. We use a particular naming convention for these functions, which is described here.

For example, we have a function named *ss_child_send_stream_start()*. Each name begins with a prefix that identifies which QStream program or to which library the function belongs. In this example, the *ss* prefix is for functions in the StreamServ program. Other program prefixes are *sp* for StreamPlay, and *fs* for FileServ. The *qsf* prefix is used for QSF library functions. The second component of the name often refers to the object on which the function acts or was triggered by. In this example, *child* refers to a network session. In this chapter, we have three names for network sessions: StreamServ's connection to FileServ is called *helper*, StreamServ's downstream connection to StreamPlay is called *child*, and StreamPlay's upstream connection to StreamServ is called *parent*[9]. The remaining suffix of the name describes the action performed by the handler, which in this example is to send a *START_REQUEST* message.

6.2.2 PPS messages

Figure 6.4 gives an example of the sequence of messages that would be exchanged between the QStream programs during the start of a PPS session. The messages exchanged between StreamServ and StreamPlay comprise the PPS protocol. The dialogue between StreamServ and, its helper program, FileServ are not part of the protocol proper, but we include them in the figure to help understand the QStream implementation.

Each PPS session begins with the StreamPlay establishing a transport level connection to StreamServ, which in turn establishes a connection to FileServ. From then on, the PPS session consists of a sequence of application level messages exchanged across the transport connections. Unlike some other protocols [74, 76], PPS interleaves control and data messages within the same transport connection.

The startup is made up of two steps. First StreamPlay sends an *OPEN_REQUEST* message

[9]When we introduce multicast, there will be intermediate nodes between StreamServ and StreamPlay.

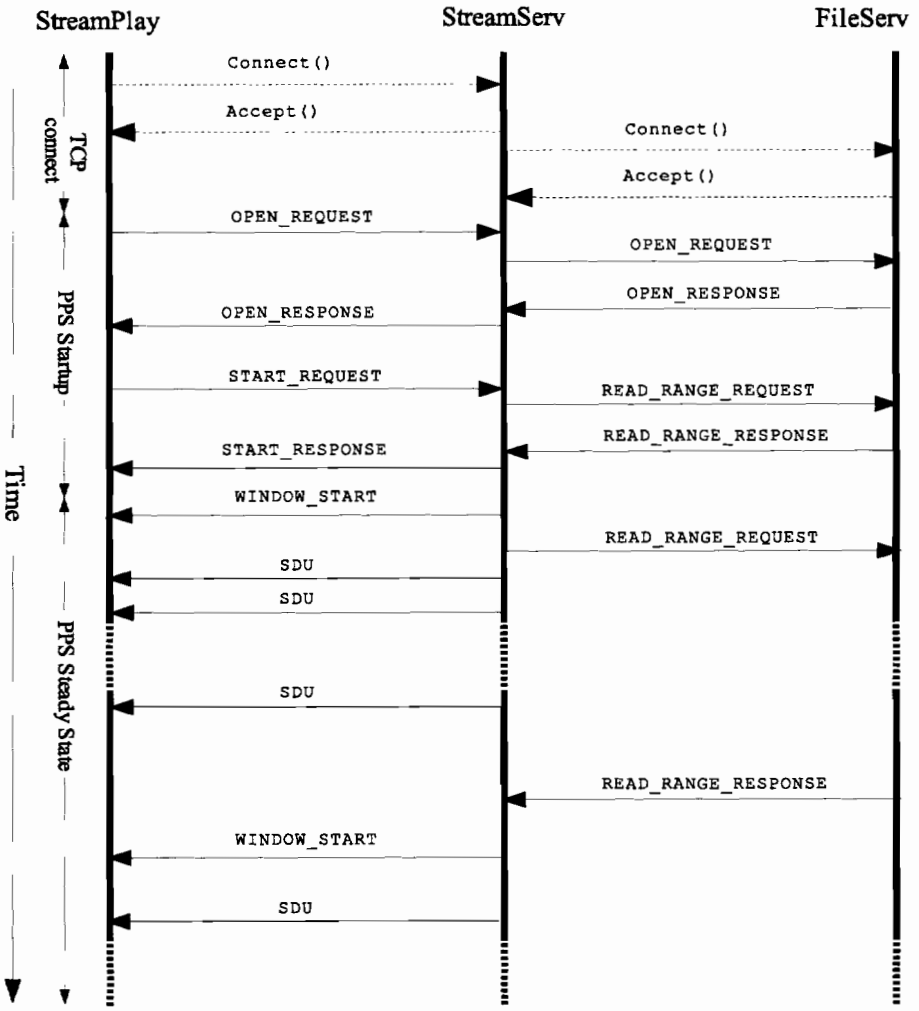


Figure 6.4: Sequence of Messages in a PPS session

to StreamServ that indicates which video to stream. StreamServ forwards this message to FileServ, which is responsible for checking that the video bitstream and index files are available for streaming. The *OPEN_RESPONSE* message is sent back to StreamServ and then forwarded from StreamServ to StreamPlay. The message indicates whether the video is available and, if so, includes a stream header for the video. The stream header contains basic information about the video (duration, resolution, audio sample frequency, ...), which the client can use, for example, to configure output devices such as the display window and audio card. StreamServ initiates preparation of the first adaptation window, which includes sending a *READ_RANGE_REQUEST* to FileServ to fetch the contents of the first adaptation window from storage. FileServ sends back a *READ_RANGE_RESPONSE* when the contents are ready. StreamServ will not actually start transmission of the first window until the second step of PPS startup completes. In the second step, the client sends a *START_REQUEST* message to StreamServ. This may not happen until the destination video and audio devices are given a chance to get ready at the client[10]. StreamServ then sends the *START_RESPONSE* message to StreamPlay.

The purpose of the *START_RESPONSE* is to provide an approximate time synchronization between the StreamServ and StreamPlay sides of the PPS session. PPS does not assume a globally synchronized time reference is available, because services such as ntp [62] are not implemented in the majority of Internet hosts. Each side maintains a session origin time, which is an absolute time reference relative to the local time. All other time values communicated between the StreamServ and StreamPlay are expressed in terms relative to the origin. StreamServ sets the origin for the PPS session to be the local time at which it sent the *START_RESPONSE* message, while StreamPlay sets it to the local time upon receipt of the *START_RESPONSE*. The basic assumption is that while the local times of StreamServ and StreamPlay may not be synchronized to the same value, they advance at the same rate[11].

Once the *START_RESPONSE* is sent, then the remainder of the session consists of a sequence of adaptation windows. From the perspective of StreamPlay, each adaptation window starts with

[10] And until the server has established a connection to the Monitor, although we elide the details of Monitor communication here.

[11] Should there be significant rate drift, the phase offset management part of the algorithm will adjust to compensate for it.

the reception of a *WINDOW_START* message from StreamServ, followed by a sequence of individual *SDU* messages. The *WINDOW_START* message contains information about the adaptation window such as its positions within the video (display) timeline and the transmission timeline. To prepare each adaptation window, StreamServ sends a *READ_RANGE_REQUEST* to FileServ, which fetches the video data from storage, and sends a *READ_RANGE_RESPONSE* back to StreamServ to indicate the data can be accessed without risk of blocking.

6.3 StreamServ Algorithm

This section presents the server-side algorithm for PPS. The presentation includes a description of the main data structures, and pseudo-code, based on the source code for StreamServ as implemented in QStream. Despite the low-level of detail here, there is still a significant amount of simplification relative to the actual source code. Some details from the source code have been omitted because they are not specific to the PPS algorithm. They include error handling, debug logging, remote monitoring, and normal session termination and cleanup. There are also some parts of the server side of the PPS algorithm that are multicast specific, they are presented later in Section 6.5.

6.3.1 StreamServ Data Structures

There are two main data structures used in the StreamServ program, one for a state related to a PPS session and the other for individual adaptation windows. They are shown in Figures 6.5 and 6.7. References to these structures will appear throughout the pseudo-code fragments that appear in the following sections.

A *StreamServSession* object, shown in Figure 6.5, is allocated for each PPS session. The *child_session* and *helper_session* fields are handles used to send and receive messages with QSF (QSF was described in Section 6.1.5). As mentioned earlier in Section 6.2.1, the *child_session* is used for the PPS protocol, while *helper_session* is used for communication to FileServ.

The StreamServ side of the PPS algorithm pipelines its preparation and transmission phases to ensure that data is always ready for transmission before it is needed, which in turn ensures that

```
ServPpsSession {
  QsfSession      child_session;     // Handle for TCP connection downstream
  QsfSession      helper_session;    // Handle for local connection to FileServ
  Queue           recv_windows;      // windows captured/fetched from storage
  Queue           mapped_windows;    // windows mapped beyond workahead limit
  Queue           xmit_windows;      // adaptation windows ready to stream
  StreamHeader    stream_header;     // contains video duration, fps, etc.
  MapSession      mapper_session;    // mapper specific sesssion state
  Integer         video_fd;          // file descriptor for video data
  Time            session_origin;    // base time of regulator clock
  Time            phase_offset;      // worst case transport latency
  Time            workahead_limit;   // for work conserving mode
  Time            expand_end;        // when window sizes stop growing?
  Time            shrink_start;      // when window sizes start shrinking?
  Time            prev_vid_end;      // video end position of latest window
  Time            prev_xmit_end;     // transmit end position of lastest window
  Float           growth_rate;       // how fast do windows grow/shrink?
  Boolean         serv_ready;        // first window ready for transmit
  Boolean         child_ready;       // child has sent start request
  ...
}
```

Figure 6.5: StreamServ PPS Session Object

the available network bandwidth will be fully utilized by PPS[12]. The basic unit of work in the pipeline is the PPS adaptation window (the *ServPpsWindow* object is described below), each stage of the pipeline maintains a separate queue (first in, first out) of windows belonging to that stage. There are three such queues per session: *recv_windows*, *mapped_windows* and *xmit_windows*. The *recv_windows* queue holds windows as they are initially fetched from storage (or captured and encoded in the case of streaming directly from a live source), and prioritized by the mapper. The *map_windows* queue holds windows that have been prioritized but are not ready for transmission (due to the workahead limit). Finally, the *xmit_windows* queue holds windows that have been fully prioritized and are eligible for transmission.

The use of queues for the pipeline stages might not seem an obvious choice. Intuitively, the server side of the PPS algorithm should only require two windows at a time (one in preparation, and the other in transmission). However, we realized queues were necessary when we extended

[12]The pipelining ensures that transient storage access latencies do not delay transmission. Non-transient latencies would be the result of insufficient sustainable throughput from the filesystem. In this dissertation, we are making the assumption that the storage system can sustain the rates required by PPS. Adapting to filesystem performance bottlenecks is outside the scope of this work.

our original implementation with features such as work conserving mode (discussed in Section 4.1.3) and live streaming, each of which can lead to scenarios with several windows in various stages of progress.

The *map_session* object contains state for the Mapper algorithm (the Mapper is described in Chapter 3). The *stream_header* field contains configuration information from the video. The contents of the *stream_header* are shown in Figure 6.6. The resolution information is used by StreamPlay to initialize the display window during startup. The *preroll_duration* is used to inform the PPS of the smallest feasible adaptation window duration, which is constrained by the GoPs that happen to occur in the particular video (see Section 3.2.1). Aside from the brief comments in Figure 6.5, we leave the discussion of remaining fields of the *ServPpsSession* object until they appear in the pseudo-code of the following sections.

```
StreamHeader {
  VideoRate video_rate;        // fps, timecode settings
  Time      duration;          // total duration of stream
  Integer   h_size;            // horizontal resolution
  Integer   v_size;            // vertical resolution
  Time      preroll_duration;
}
```

Figure 6.6: StreamHeader Object

Figure 6.7 shows the *ServPpsWindow* object that is allocated per adaptation window. The *vid_start* and *vid_end* fields delimit the position of this window within the video timeline. The *xmit_start* and *xmit_end* are the window's position within the transmission timeline.

Before priority-mapping, the contents of the adaptation window is a set of ADUs. The *fetch_done* flag is used to track whether FileServ has fetched the contents. When received from FileServ, the *adus* field stores these ADUs, which are as yet unprioritized. The content of *adus* is consumed by the mapper algorithm, which prioritizes and groups the ADUs transforming them into SDUs. As the mapper proceeds, it inserts SDUs into a heap data structure, the *sdus* field. Using a heap allows the priority sorting to be done incrementally. The ADU and SDU object types are shown in Figure 6.8.

In work conserving mode, the transmission of an adaptation window is allowed to start immediately if bandwidth was enough for the previous window to finish before its deadline. However,

```
ServPpsWindow {
  Time     vid_start;       // start position in video timeline
  Time     vid_end;         // end position in video timeline
  Time     xmit_start;      // start position in transmit timeline
  Time     xmit_end;        // end position in transmit timeline
  Time     xmit_deadline;   // end position in absolute time
  Boolean  fetch_done;      // window has arrived
  Queue    adus;            // window contents before mapping (time order),
  Heap     sdus;            // window contents after mapping (priority order)
  Timeout  start_timeout    // handle to timeout scheduled to start transmit
  Timeout  xmit_timeout     // handle to timeout scheduled to stop transmit
}
```

Figure 6.7: StreamServ Adaptation Window Object

this is allowed only up to the configurable *workahead_limit* for the session (see *ServPpsWindow* above). The *start_timeout* field stores a handle to a scheduled callback, in this case the callback enqueues the window for transmission. The use of *start_timeout* may possibly delay the transmission of the current window so that *workahead_limit* is honoured. The *xmit_timeout* is used to issue a callback that will trigger the transmission phase to stop for the current window (dropping unsent SDUs), allowing the next window to start. In the non work conserving configuration, the *start_timeout* deadline for a window will always be equal to the *xmit_timeout* deadline of the previous window.

```
PpsSdu {
   Time              timestamp;          // derived from map window
   Integer           priority;           // assigned by mapper
   Integer           num_adus;           // PpsAdus follow, then ADU payloads
   PpsAdu            adus[num_adus];     // Index ADU payloads
   Bytes             payloads[];         // ADU payloads (variable length)
}

PpsAdu {
   FileOffset offset;
   Integer    length;
}
```

Figure 6.8: ADU and SDU Objects

Figure 6.8 shows the message related objects, for SDUs and ADUs. An SDU contains the *timestamp* and *priority*, along with a group of ADUs. The *adus* field is an array which describes

the logical location of the ADU within the video bitstream. The actual ADU payloads form the suffix of the SDU.

This completes our description of the StreamServ data structures. The PPS algorithm will be described in pseudo-code in the following three sections, one each for the three phases of the algorithm: startup, window preparation, and window transmission.

6.3.2 StreamServ Phase I: Session Startup

The session startup phase is where StreamServ accepts a new child connection and the initial PPS protocol sequence is processed. This protocol sequence starts with an *OPEN_REQUEST* message from downstream that indicates which video is requested. After receiving this message, StreamServ initiates a connection to FileServ for the session[13]. Once the connection to FileServ is established, StreamServ forwards the *OPEN_REQUEST* message to FileServ, which opens the requested video and its index. After opening the files, FileServ generates a response message to StreamServ that includes the video stream header. StreamServ forwards the response downstream. StreamPlay uses the information in the stream header for initializing the output window and audio device. Finally, StreamServ initiates the preparation phase for the first adaptation window.

6.3.2.1 ss_child_acceptor()

SS_CHILD_ACCEPTOR(*child_session*)
1 *pps* ← NEW(*ServPpsSession*)
2 *pps.child_session* = *child_session*
3 *pps.helper_session* ← QSF_CONNECT(*fileserv_address*, *helper_methods*, *pps*)

This function is the entry point for the the PPS algorithm, it is called when a new connection has been accepted. This marks the beginning of a new PPS session. Line 1 creates a new *ServPpsSession* object. Line 3 initiates a local connection to FileServ, *fileserv_address* is the name of a unix local socket. The *helper_methods* argument is an array that associates PPS messages with the corresponding functions below.

[13]Another option would be to have a single connection between StreamServ and FileServ, multiplexing requests from all active PPS sessions. This is how communication with the remote monitor works.

6.3.2.2 ss_helper_connected()

SS_HELPER_CONNECTED(*pps*)
1 QSF_START_MSG_RECVS(*pps.child_session*)
2 QSF_START_MSG_RECVS(*pps.helper_session*)

This function is called when the per-session connection to FileServ has been established. At this point, the application-level message dispatching is enabled for both the downstream TCP connection and the local connection to FileServ (lines 1 and 2). Between the time a connection is established and the time dispatching is enabled, messages transmitted over the connection would simply accumulate in OS kernel buffers. The only message expected at this point in PPS is the *OPEN_REQUEST* message from downstream (see Figure 6.4), which mainly indicates which video the user wishes to stream.

6.3.2.3 ss_child_recv_open_file()

SS_CHILD_RECV_OPEN_FILE(*pps*, *open_file_request*)
1 QSF_SEND_MSG(*pps.helper_session*, *open_file_request*)

ss_child_recv_open_file is called upon receipt of the *OPEN_REQUEST* message from StreamPlay, it simply forwards the request to FileServ (line 1).

6.3.2.4 ss_helper_recv_open_file()

SS_HELPER_RECV_OPEN_FILE(*pps*, *open_file_response*)
1 *pps.stream_header* ← *open_file_response.stream_header*
2 QSF_SEND_MSG(*pps.child_session*, *open_file_response*)
3 SS_WIN_PREP_FIRST(*pps*)

ss_helper_recv_open_file is dispatched upon the arrival of an *OPEN_RESPONSE* message from FileServ. The message contains a stream header object which contains information that will be needed in *ss_win_prep_first*, such as the required preroll duration and the total duration of the video (line 1). The message is forwarded downstream to StreamPlay so that it may use the stream header information to initialize its display (line 2). Line 3 calls *ss_win_prep_first* to initiate preparation of the first adaptation window.

6.3.2.5 ss_win_prep_first()

```
SS_WIN_PREP_FIRST(pps)
 1  win ← NEW(ServPpsWindow)
 2  win.xmit_start ← 0
 3  win.vid_start ← 0
 4  win.vid_end ← initial_win_size
 5  if pps.growth_rate > 1
 6    then pps.expand_end ← pps.max_win_duration/(pps.growth_rate − 1)
 7  pps.shrink_start ← pps.stream_header.duration − pps.expand_end
 8  if pps.shrink_start < pps.expand_end
 9    then pps.expand_end ← pps.stream_header.duration/2
10         pps.shrink_start ← pps.expand_end
11  ENQUEUE(pps.recv_windows, win)
12  SS_HELPER_SEND_READ_RANGE(pps)
```

ss_win_prep_first allocates a *ServPpsWindow* object for the first window of the session timeline (line 1). The *xmit_start* and *vid_start* fields are set to zero since this is the first window in the stream (lines 2–3). The *vid_end* is set to according to a configuration value *initial_win_size* (line 4), which may be adjusted downward later by FileServ to align with the first GoPs in the video. If window scaling is enabled (checked on line 5), then lines 5–10 compute the *expand_end* and *shrink_start* values that are to be used in later steps for computing the sequence of adaptation windows. Note that if *growth_rate* is 1, then the *expand_end* and *shrink_start* values will have no effect (they will be 0 and *duration* respectively). Line 11 puts the new window in the *recv_windows* queue, where it will stay until the preparation phase is complete. The first step in window preparation is for FileServ to locate and fetch the contents of the window from storage. This is initiated by the call to *ss_helper_send_read_range* function (see Section 6.3.3.1) which sends a read range message for the new window to FileServ (line 12).

6.3.2.6 ss_child_recv_start_stream()

```
SS_CHILD_RECV_START_STREAM(pps)
1  pps.child_ready ← TRUE
2  if pps.serv_ready
3    then SS_START_STREAM(pps)
```

ss_child_recv_open_file is called upon receipt of the *START_REQUEST* message from StreamPlay. This message indicates that the child is ready to receive the first adaptation window, which is recorded in the variable *child_ready* (Line 1). Line 2 tests *serv_ready*, which indicates whether the first adaptation window has been fully prepared. If so, the call to *ss_start_stream* (see Section 6.3.3.5) will begin the transmission phase for the session. Otherwise, transmission will begin later when the preparation of the first adaption window completes (see Section 6.3.3.4).

6.3.3 StreamServ Phase II: Window Preparation

This phase is driven by the arrival of *READ_RANGE_RESPONSE* messages from FileServ. Each message will include descriptors for all the ADUs that fall within an adaptation window interval. StreamServ calls the priority map algorithm to convert the ADUs to a set of prioritized SDUs. When mapping is complete, the adaptation window can enter the transmission phase of StreamServ.

6.3.3.1 ss_helper_send_read_range()

```
SS_HELPER_SEND_READ_RANGE(pps)
1  win ← PEEK_HEAD(pps.recv_windows)
2  read_range_msg.vid_start ← win.vid_start
3  read_range_msg.vid_end ← win.vid_end
4  read_range_msg.win ← win
5  QSF_SEND_MSG(pps.helper_session, read_range_msg)
```

The first step in the preparation phase is to send a *READ_RANGE_REQUEST* message to FileServ (lines 1–5), which directs FileServ to locate and retrieve all the ADUs that fall within the interval of the window.

6.3.3.2 ss_helper_recv_read_range()

```
SS_HELPER_RECV_READ_RANGE(pps, read_range_response)
1  win = PEEK_HEAD(pps.recv_windows)
2  win.fetch_done ← TRUE
3  win.adus ← read_range_response.adus
4  win.vid_start ← read_range_response.start
5  win.vid_end ← read_range_response.end
```

```
6  vid_duration = win.vid_end − win.vid_start
7  switch
8    case win.xmit_start < pps.expand_end :
9        xmit_duration ← vid_duration ÷ pps.growth_rate
10   case win.xmit_start ≥ pps.shrink_start :
11       xmit_duration ← vid_duration × pps.growth_rate
12   case default :
13       xmit_duration ← vid_duration
14 win.xmit_end = win.xmit_start + xmit_duration
15 display_start = pps.stream_header.preroll_duration + win.vid_start
16 win.xmit_end = MIN(display_start, win.xmit_end)
17 G_AIO_SCHEDULE_EVENT(ss_map_adus, pps)
18 pps.prev_vid_end = win.vid_end
19 pps.prev_xmit_end = win.xmit_end
```

ss_helper_recv_read_range is dispatched when the *READ_RANGE_RESPONSE* message arrives from FileServ. The response message contains an array of ADU descriptors for the ADUs that fall within the range requested, which are stored for later processing by the priority mapper (line 3). The response also includes the adjusted adaptation window boundaries (lines 4–5). FileServ is allowed to adjust the requested video boundaries downward if necessary; for example, to make sure that the adaptation window boundary is aligned with timestamps in the video[14]. This alignment is necessary to ensure that window duration accurately matches the amount of video the window contains.

Lines 6–16 compute the transmission expiry deadline for the window. The transmission deadline is derived from the window's duration in the video timeline and the session's window scaling policy. Lines 8–9 treat the case where the window is part of the window scaling expansion phase. Similarly, lines 10–11 treat the shrink phase, and line 13 treats the neutral phase. Lines 14–16 clamp the deadline to ensure the transmission ends by the time display must start. The next step of the preparation phase is priority mapping. Since the mapper algorithm is CPU oriented, the mapper function is scheduled as an event with the GAIO event dispatcher (line 17), which will call *ss_map_adus* as soon as no other higher priority events are pending. Lines 18–19 store the actual end position of this window which will become the start position for the next window.

[14]It also ensures that the response is non-empty, that is, it contains at least one GoP

6.3.3.3 ss_map_adus()

```
SS_MAP_ADUS(pps)
1  win ← PEEK_HEAD(pps.recv_windows)
2  MAPPER_PRIORITIZE(win.adus, win.sdus)
3  if QUEUE_EMPTY(win.adus)
4    then SS_MAP_DONE(pps)
5    else G_AIO_SCHEDULE_EVENT(ss_map_adus, pps)
```

ss_map_adus is a wrapper function that calls the mapper algorithm to prioritize the contents of the adaptation window. The adaptation window consists of a sequence of one or more smaller intervals called *map windows*, which were discussed in Section 3.2. Line 2 calls to the mapper to prioritize a single map window, which will consume some number of the ADUs in the *win.adus* queue. When the mapper has prioritized the entire adaptation window, the *win.adus* queue will be empty (line 3) and the adaptation window is ready to enter the transmission phase (line 4). Otherwise, the mapper is scheduled to continue processing the ADUs of the next map window (line 5).

6.3.3.4 ss_map_done()

```
SS_MAP_DONE(pps)
 1  win ← DEQUEUE(pps.recv_wins)
 2  ENQUEUE(pps.mapped_wins, win)
 3  if win.number = 0
 4    then pps.serv_ready ← TRUE
 5         if pps.child_ready
 6           then SS_START_STREAM(pps)
 7    else xmit_deadline ← pps.session_origin + win.xmit_start − pps.workahead_limit
 8         win.start_timeout ←
 9                 G_AIO_SCHEDULE_TIMEOUT(xmit_deadline, SS_WIN_XMIT_START, pps)
10  win ← PEEK_HEAD(pps.recv_wins)
11  if win ≠ NIL and win.fetch_done
12    then G_AIO_SCHEDULE_EVENT(ss_map_adus, pps)
```

When preparation of an adaptation window is complete, *ss_map_adus* calls *ss_map_done*. At this point, the window is moved from the *recv_wins* queue to the *mapped_wins* queue (lines 1–2).

Lines 3–6 of *ss_map_done* treat the case of the very first adaptation window in the stream.

Line 4 updates *serv_ready* to indicate that the first adaptation window is ready for transmission. Line 5 checks whether the client has send a *START_REQUEST* message yet. If so, the call to *ss_start_stream* (described below) in Line 6 will initiate the transmission phase.

Lines 7–9 of *ss_map_done* treat all adaptation windows other than the first. Line 7 computes the deadline (in absolute time units) at which transmission of the window should start. If the value of *workahead_limit* is greater than zero, then the algorithm is in the work conserving configuration. Lines 8–9 schedule the dispatch *ss_win_xmit_start* at the deadline. If the value of *xmit_deadline* happens to have already past, then the event dispatcher will dispatch *ss_win_xmit_start* as soon as possible. Lines 10–12 check whether the next adaptation window is available and ready for mapping, and if so, an event is scheduled to invoke the mapper.

6.3.3.5 ss_start_stream()

```
SS_START_STREAM(pps)
1  pps.session_origin ← GET_CURRENT_TIME()
2  SS_CHILD_SEND_STREAM_START(pps)
3  SS_WIN_XMIT_START(pps)
```

The *ss_start_stream* routine initiates the transmission phase for the first adaptation window in the stream. As described in Section 6.2.2, line 1 is where the server side session clock is started, by recording the current time in *session_origin*. The ss_child_send_stream_start function (line 2) simply constructs the *START_RESPONSE* message and sends it to the child. Line 3 calls *ss_win_xmit_start* (described next) which begins transmitting the actual window contents.

6.3.3.6 ss_win_xmit_start()

```
SS_WIN_XMIT_START(pps)
1  win ← DEQUEUE(pps.mapped_wins)
2  ENQUEUE(pps.xmit_wins, win)
3  xmit_deadline ← pps.session_origin + pps.phase_offset + win.xmit_end
4  win.xmit_timeout ← G_AIO_SCHEDULE_TIMEOUT(xmit_deadline,
5                                            SS_WIN_XMIT_EXPIRE, win)
6  SS_WIN_PREP_NEXT(pps)
7  if QUEUE_LENGTH(pps.xmit_windows) = 1
8    then SS_CHILD_SEND_WIN_START(pps)
```

ss_win_xmit_start is called each time an adaptation window is ready to begin the transmission phase. By this point, the window contents have been fetched into memory, prioritized, and sorted into priority order. By queueing the window for transmission, the window enters the transmission phase (line 1). A timeout is scheduled to expire the window (lines 2–5). If the timeout fires before the transmission of the entire window contents completes, then the algorithm will proceed to drop unsent SDUs for this window (see Sections 6.3.3.7 and 6.3.4.4). Line 6 invokes a helper called *ss_win_prep_next* to initiate preparation of the next adaptation window in the video timeline, this preparation will then proceed concurrently with the transmission of the current window. Lines 7–8 start (or resume) the transmission phase if necessary, which is determined by the absence of other adaptation windows in the transmission queue[15].

6.3.3.7 ss_win_xmit_expire()

```
SS_WIN_XMIT_EXPIRE(win)
1  win.xmit_timeout ← NIL
```

The *ss_win_xmit_expire* function simply clears the *xmit_timeout* field. This will trigger the retirement of the current window in *ss_sdu_next* (see Section 6.3.4.4).

6.3.3.8 ss_win_prep_next()

```
SS_WIN_PREP_NEXT(pps)
 1  if pps.prev_vid_end = pps.stream_header.duration
 2    then return ()
 3  new_win ← NEW(ServPpsWindow)
 4  ENQUEUE(pps.recv_windows, new_win)
 5  new_win.vid_start ← pps.prev_vid_end
 6  new_win.xmit_start ← pps.prev_xmit_end
 7  xmit_duration ← pps.stream_header.preroll_duration + new_win.vid_start−
 8                  new_win.xmit_start
 9  switch
10    case new_win.xmit_start < pps.expand_end :
11         vid_duration ← xmit_duration ÷ pps.growth_rate
```

[15]The transmission phase pauses if it empties the queue, which would happen if bandwidth were abundant.

```
12    case new_win.xmit_start ≥ pps.shrink_start :
13        vid_duration ← xmit_duration × pps.growth_rate
14    case default : vid_duration ← xmit_duration
15  vid_duration = CLAMP(vid_duration, pps.min_win_duration, pps.max_win_duration)
16  new_win.vid_end ← new_win.vid_start + vid_duration
17  SS_HELPER_SEND_READ_RANGE(pps)
```

ss_win_prep_next instantiates the *ServPpsWindow* object for the next adaptation window in the PPS timeline, and initiates the preparation phase for the new window. This includes computing the new window's position in the video and transmission timelines, taking into account the window scaling settings of the session.

Line 1 checks if the previous adaptation window reached the end of the stream. If so, the function returns immediately without creating a window (line 2). Otherwise, line 3 allocates the object for the new window and line 4 puts that window into *recv_windows* queue. Lines 5–6 set the start positions for the new window in the transmission and video timelines. These start positions are set to the end positions of the last window, ensuring that the timelines are free of gaps. Lines 7–16 compute the target duration of the new window in the transmit and video timelines. It should be noted that the actual durations (in the video and transmit timelines) may end up shorter, since FileServ may round the window boundary down to align with the timestamps present in the stream (see Section 6.3.3.2). The target duration is computed based on the amount of time available to transmit, which is computed in Line 7 as the difference between the time that window will be needed for display (preroll + video start) and the time that the window will start transmission. Then, using this transmission time and the window scaling schedule, lines 9–14 compute the duration in the video timeline. Lines 10–11 treat the case where the window falls within the window scaling expansion phase. Lines 12–13 treat the shrink phase. Line 14 treats the neutral phase. Line 15 ensures that the resulting duration falls within configuration parameters for the session. Line 17 initiates the iteration of the preparation stage for the newly created adaptation window, issuing the read range request to FileServ.

6.3.4 StreamServ Phase III: Window Transmission

The third phase of StreamServ handles network transmission of the adaptation window. A *WINDOW_START* message is sent at the beginning of each window, followed by as many SDUs as the network will allow before the window's transmission expiry deadline. When the deadline passes, low priority SDUs are dropped, and transmission of the next adaptations window commences.

6.3.4.1 ss_child_send_win_start()

```
SS_CHILD_SEND_WIN_START(pps)
1  win ← QUEUE_PEEK_HEAD(pps.xmit_windows)
2  win_start_msg.vid_start ← win.vid_start
3  win_start_msg.vid_end ← win.vid_end
4  win_start_msg.xmit_start ← win.xmit_start
5  win_start_msg.xmit_end ← win.xmit_end
6  win_start_msg.num_sdus ← win.num_sdus
7  win_start_msg.num_base_sdus ← win.num_base_sdus
8  QSF_SEND_MSG(pps.child_session, win_start_message, SS_CHILD_SEND_SDU_HEAD)
```

ss_child_send_win_start is the entry point for the transmission phase of StreamServ. It instantiates a *WINDOW_START* message and sends the message downstream.

The SDU data structure consists of a header and a payload. Recall that all ADUs with the same priority in a mapper window are grouped into one SDU (see Section 3.2). Thus, an SDU payload consists of the ADUs that have been grouped together. The header part of the SDU contains and array of ADU descriptors which specify offset and length of each ADU within the video bitstream. The descriptors are present in the SDU so that StreamPlay can re-sort ADUs back from priority order back to bitstream order. The second part of an SDU contains actual video data from the bitstream file.

Transmission of each SDU is divided across two functions in QStream. The extra function is required in order to take advantage of the *sendfile* primitive to forward raw video data directly from the filesystem to the downstream socket[16]. Using *sendfile* instead of read and write reduces CPU overhead. The reduction is due to fewer user-kernel context switches, fewer memory copies, and

[16] *sendfile()* is available on Linux and BSD. Similar primitives are available in most other operating systems.

offloading of memory copies from the CPU to the NIC. The code to transmit an SDU is divided into the following two functions, which send the header and payload parts respectively.

6.3.4.2 ss_child_send_sdu_head()

```
SS_CHILD_SEND_SDU_HEAD(pps)
1  win ← QUEUE_PEEK_HEAD(pps.xmit_windows)
2  msg ← HEAP_PEEK_MIN(win.sdus)
3  win.adu_count ← 0
4  QSF_SEND_MSG(pps.child_session, msg, SS_CHILD_SEND_ADU)
```

ss_child_send_sdu_head transmits the header part of the SDU. Lines 1–2 select the SDU from the head of queue for the current adaptation window of the transmission phase. Line 3 initializes the count of adus sent (for the current SDU) to zero. Line 4 sends the header part of the SDU message.

6.3.4.3 ss_child_send_adu()

```
SS_CHILD_SEND_ADU(pps)
 1  win ← QUEUE_PEEK_HEAD(pps.xmit_windows)
 2  if win.adu_count < sdu.num_adus
 3    then msg ← HEAP_PEEK_MIN(win.sdus)
 4         sdu ← msg.sdu
 5         offset ← sdu.adus[win.adu_count].offset
 6         count ← sdu.adus[win.adu_count].count
 7         win.adu_count ← win.adu_count + 1
 8         qsf_sendfile_msg(pps.child_session, pps.video_fd, offset, count,
 9                          SS_CHILD_SEND_ADU)
10    else HEAP_DELETE_MIN(win.sdus)
11         SS_SDU_NEXT(pps, win)
```

ss_child_send_adu transmits the video data parts of the current SDU to the downstream socket using *sendfile()*. The function iterates through the path from lines 3–9 for each ADU in the SDU, transmitting them one by one. When all ADUs have been transmitted, the current SDU is retired (line 10) and the helper function *ss_sdu_next* is called to start transmission of the next SDU.

6.3.4.4 ss_sdu_next()

```
SS_SDU_NEXT(pps, win)
1  if HEAP_IS_EMPTY(win.sdus)
2    then SS_WIN_DONE(pps, win)
3    else sdu ← HEAP_PEEK_MIN(win.sdus)
4         if win.xmit_timeout = NIL and sdu.priority < MAX_PRIORITY
5           then SS_WIN_DONE(pps, win)
6           else SS_CHILD_SEND_SDU_HEAD(pps)
```

When transmission of the current SDU is complete, *ss_sdu_next* is called to start transmission of the next SDU. Line 1 checks if the current SDU was the last, if so, line 2 calls *ss_win_done* (described below) to perform end of window processing. *ss_sdu_next* will also call *ss_win_done* (line 5) if it detects that the window expiry timeout has occurred, and also provided that at least the base layer SDUs are done (line 4). Otherwise, Line 6 initiates transmission of the next SDU in the window.

6.3.4.5 ss_win_done()

```
SS_WIN_DONE(pps, win)
1  QUEUE_REMOVE(pps.xmit_windows, win)
2  if win.xmit_timeout ≠ NIL
3    then CANCEL_TIMEOUT(win.xmit_timeout)
4         win.xmit_timeout ← NIL
5  if not QUEUE_IS_EMPTY(pps.xmit_windows)
6    then SS_CHILD_SEND_WIN_START(pps)
7    else if win.end = pps.stream_header.duration
8           then QSF_SEND_MSG(pps.child_session, eof_msg)
```

ss_win_done treats end of window processing. Line 1 removes the current window from the transmission queue. If the current window has finished before its expiry timeout has occurred, then the current window's timeout is cancelled (lines 2–4). If the transmission queue contains another adaptation window, then transmission of that window commences immediately (lines 5–6). Otherwise, if the current window is the last in the stream, then an end of stream message is sent downstream (lines 7–8).

This completes the description of the server side of PPS. The next section will describe the client side of PPS.

6.4 StreamPlay Algorithm

The client side of the PPS algorithm is implemented in StreamPlay. Similar to the server case, there are three main phases in the client side of the PPS algorithm.

The first phase is session startup. A connection to StreamServ is initiated. Once established, an *OPEN_REQUEST* message is sent upstream to StreamServ, to identify the video to stream. An *OPEN_RESPONSE* from StreamServ will provide information about the video. This information is used to initialize output devices. Once they are ready, a *START_REQUEST* message will be send upstream. A subsequent *START_RESPONSE* message will indicate the start of continuous streaming, and marks the point where the client-side clock for the session should start.

The second phase in StreamPlay receives adaptation windows from StreamServ. Each window consists of a *WINDOW_START* message and a sequence of SDUs. Since the contents of the window are re-ordered as they arrive, the reception of the entire window must complete before it can be displayed. To maintain continuous play-out, the window must be ready for display—at the latest—by the time all frames of the previous window have been displayed. Thus a timeout is set for each window to prevent gaps in the display timeline. If the start of the next window arrives before the timeout occurs, the timeout is cancelled. Otherwise, when the timeout fires, the ADUs already present are committed for display, and subsequent SDU arrivals are considered late. In the event of late SDUs, an adjustment is made to the phase offset to try and prevent late SDUs from also occurring in future windows.

The final StreamPlay phase decodes and displays the contents of the adaptation windows.

6.4.1 Data Structures

Similar to StreamServ, StreamPlay's main data structures are a per-session object called *PlayPpsSession* and a per-adaptation window object called *PlayPpsWindow*, shown in Figures 6.9 and 6.10.

A *PlayPpsSession* object is allocated for each active PPS session. Normally the player will have just one active session at a time, although multiple sessions are plausible, for example in

```
PlayPpsSession {
  QsfSession    parent_session;    // handle for TCP connection upstream
  StreamHeader  stream_header;     // contains video duration, fps, etc.
  Time          session_origin;    // base time of regulator clock
  Time          slack;             // how early did last SDU arrive?
  PlayPpsWindow xmit_window;   // window in transmission
  Queue         decode_windows;    // adaptation windows in decode/display
}
```

Figure 6.9: StreamPlay Per-Session State

surveillance applications. The *parent_session* field is a handle to the network socket to StreamServ corresponding to the PPS session. The *stream_header* is also received during the startup phase, and is used to initialize decode and display components. The *session_origin* contains the start time of the transmission phase for this session, in absolute (wall-clock) time units, and is used as the basis for converting time of day values to the transmission and display timelines. For every SDU that arrives, the *slack* field is set to the amount of time remaining before the deadline for its adaptation window. A negative value means the last SDU received was late. The *slack* value is used to maintain the correct phase offset between the StreamServ and StreamPlay clocks. The *xmit_window* field is a *PlayPpsWindow* object (described below) corresponding to the current adaptation window of the transmission phase. The *decode_windows* field is a queue of *PlayPpsWindow* objects for adaptation window(s) in the process of decoding. A queue is used to allow the transmission phase to work more than one window ahead of the decode and display phase, which can be necessary when PPS is configured to be work conserving.

```
PlayPpsWindow {
  Time       vid_start;       // start position in video timeline
  Time       vid_end;         // end position in video timeline
  Time       xmit_start;      // when should xmit start
  Time       xmit_end;        // when should xmit end
  Timeout    xmit_timeout;    // handle for cancellation
  Boolean    decode_started;  // has decode already started?
  Integer    sdu_count;       // how many SDUs so far
  Integer    num_base_sdus;   // how many SDUs in base layer
  Heap       adus;            // window contents (time order),
}
```

Figure 6.10: StreamPlay Adaptation Window Object

Figure 6.10 shows the *PlayPpsWindow* object, which is instantiated for each adaptation window in the video timeline. The *vid_start* and *vid_end* fields delimit the position of the window in the video timeline. The *xmit_end* field contains the time at which the decode/display phase for the window must start, which is also the time that any subsequent SDUs for this window must be considered late. The *xmit_timeout* field is a handle to a scheduled timeout, which can be used to cancel the timeout in the event that processing of the window completes prior to the timeout deadline. The *num_sdus* and *num_base_sdus* are used to track the status of the current transmission phase window, in order to detect conditions such as when the base layer is complete and when the entire window is complete (reached full quality). The *adus* field is a handle to the heap data structure that is used to sort the contents of SDUs from priority order back to the original time order.

This completes our description of client-side data structures. The following sections will provide pseudo-code for the three phases of the client side of the PPS algorithm: startup, receive windows, and decode and display.

6.4.2 StreamPlay Phase I: Session Startup

The first stage of the StreamPlay PPS algorithm is where session startup is treated. The startup consists exchange of four messages with the server (as described in detail in Section 6.2.2) : *OPEN_REQUEST*, *OPEN_RESPONSE*, *START_REQUEST*, and *START_RESPONSE*.

6.4.2.1 sp_parent_connected()

```
SP_PARENT_CONNECTED(pps)
1  QSF_START_MSG_RECVS(pps.parent_session)
2  QSF_SEND_MSG(pps.parent_session, open_file_msg)
```

The *sp_parent_connected()* function is the entry point of the session startup phase, when the upstream connection has been established. Line 1 starts message dispatching for incoming messages on the connection to the parent. Line 2 sends an *OPEN_REQUEST* request to StreamServ to identify which video to transmit.

6.4.2.2 sp_parent_recv_open_file()

```
SP_PARENT_RECV_OPEN_FILE(pps, open_file_response)
1  pps.stream_header ← open_file_response.stream_header
2  SP_DISPLAY_INIT(pps)
3  QSF_SEND_MSG(pps.parent_session, start_request)
```

sp_parent_recv_open_file is dispatched when the *OPEN_RESPONSE* message has arrived. The stream header field of the response includes basic parameters necessary to initialize play-out, such as duration of the stream, width and height of the video, and frame rate. Line 2 calls *sp_display_init* to do whatever is necessary to initialize the video decoder and display window. After that, line 3 sends the *START_REQUEST* message upstream to indicate the player is ready to commence streaming. An alternative option would be to wait until a separate start request is made by the user. This might be preferable in some circumstances, for example to give user a chance to select preferences, such as adjusting window size.

6.4.2.3 sp_parent_recv_stream_start()

```
SP_PARENT_RECV_STREAM_START(pps, stream_start_msg)
1  pps.session_origin ← GET_CURRENT_TIME()
```

The *START_REQUEST* message has arrived. The client side stream clock is initialized at this point. The first adaptation window of the stream should follow immediately, marking the start of the receive phase.

6.4.3 StreamPlay Phase II: Receive Windows

The second phase of StreamPlay is where adaptation windows are received, each window consisting of a window start message and a sequence of SDUs in priority order. Based on information in the window start message, an expiry deadline is set for the adaptation window. Before the deadline, as SDUs arrive they are disassembled into their constituent ADUs, and the ADUs are sorted to their original bitstream (time) order. If the deadline passes before the next window starts, then the ADUs that have already arrived are committed to the decode and display phase, and any subsequent SDUs for the current window are considered late.

6.4.3.1 sp_parent_rev_win_start()

```
SP_PARENT_RECV_WIN_START(pps, win_start_msg)
 1  if pps.xmit_window ≠ NIL
 2    then SP_WIN_DONE(pps)
 3  win ← NEW(PlayPpsWindow)
 4  win.num_sdus ← win_start_msg.num_sdus
 5  win.num_base_sdus ← win_start_msg.num_base_sdus
 6  win.xmit_deadline = pps.session_origin + win_start_msg.xmit_end
 7  win.xmit_timeout = G_AIO_SCHEDULE_TIMEOUT(win.xmit_deadline,
 8                                            SP_WIN_EXPIRE, win)
 9  pps.xmit_window ← win
```

ss_parent_recv_win_start is dispatched when a *window start* message arrives from StreamServ. This is also the normal indication that the previous window is done, and that it should be committed to the decode and display phase (lines 1–2). A new instance of *PlayPpsWindow* is created for the new window, and initialized according to the contents of the window start message (lines 3–6). Lines 7–8 schedule a timeout to expire the window at its deadline to ensure that the display does not stall due to late arrival of the next transmission window. Finally, the new window object is set to be the current window for the transmit phase of the session (line 9). After the window start message, SDUs for this window will start to arrive.

6.4.3.2 sp_parent_recv_sdu()

```
SP_PARENT_RECV_SDU(pps, sdu_msg)
 1  win = pps.xmit_window
 2  win.sdu_count ← win.sdu_count + 1
 3  win.slack ← win.xmit_deadline − GET_CURRENT_TIME()
 4  if win.xmit_timeout ≠ NIL or sdu_msg.priority = MAX_PRIORITY
 5    then for i ← 0 to sdu.num_adus − 1
 6              do HEAP_INSERT(win.adus, sdu.adus[i])
 7  switch
 8    case win.sdu_count = win.num_sdus :
 9         SP_WIN_DONE(pps)
10    case pps.xmit_timeout = NIL and win.sdu_count = win.num_base_sdus :
11         SP_WIN_EXPIRE(pps)
```

sp_parent_recv_sdu is called upon the arrival of each SDU message. A counter in the adaptation window object tracks how many SDUs have arrived so far for the window (line 2). The counter is used to check two special case conditions. The first is whether the base layer of the window is complete. The second is whether all of its SDUs have arrived. In addition to the SDU counter, a *slack* value for the window is updated upon the arrival of each SDU, where the slack is the difference between the expiry deadline for the window and the arrival time of the SDU (line 3). The slack represents how early or late the SDU arrived. When the window experiences late SDUs, the slack value will be used to adjust the phase offset between server and player clocks.

If the display phase hasn't begun for the current transmission window, then the ADUs contained within the SDU are entered into a heap, which has the effect of sorting all of the ADUs for the window back to their original time-order (lines 4–6).

Lines 7–11 treat two cases where the window might be ready for the decode and display phase. The first case is that the SDU counter has reached the total number of SDUs for this window, so the window is committed immediately (lines 8–9). The second case is that the window's timeout previously expired, but the base layer is only now complete with the arrival of the current SDU (lines 10–11).

6.4.3.3 sp_win_done()

```
SP_WIN_DONE(pps)
1  if pps.xmit_window.xmit_timeout ≠ NIL
2    then CANCEL_TIMEOUT(pps.xmit_window.xmit_timeout)
3         SP_WIN_EXPIRE(pps)
4  if pps.xmit_window.slack < 0
5    then phase_adus_msg.tardiness ← (−pps.xmit_window.slack)
6         QSF_SEND_MSG(pps.parent_session, phase_adjust_msg)
```

sp_win_done is called by *sp_parent_recv_win_start* or *sp_parent_recv_sdu* (for the last SDU in a window) to retire the current transmit-phase window. Lines 1-3 handle cancelling the window's expiry timeout, if it hasn't already fired. Lines 4–6 check whether there were late SDUs in this window. If so, a *phase adjust* message is sent to StreamServ, which tells StreamServ how much it should adjust its clock in order to avoid late SDUs in future windows[17].

[17] Alternatively, we could adjust the display clock, as discussed in section 4.4.

6.4.3.4 sp_win_expire()

```
SP_WIN_EXPIRE(pps)
1  pps.xmit_window.xmit_timeout ← NIL
2  if pps.xmit_window.sdu_count ≥ pps.xmit_window.num_base_sdus
3    then pps.xmit_window.display_started ← TRUE
4         ENQUEUE(pps.decode_windows, pps.xmit_window)
5         if QUEUE_LENGTH(pps.decode_windows) = 1
6           then SCHEDULE_IDLE(SP_DECODE, pps)
```

sp_win_expire is either triggered by the adaptation window expiry timeout, or by the arrival of the last base layer SDU in the event of *base layer backup* (see Section 4.1.2). Line 1 clears the value of the *xmit_timeout* field in the adaptation window object. This way, *xmit_timeout* can be used elsewhere to detect whether the window expiry has occurred. If the base layer is complete (line 2), then the window enters decode phase immediately (line 4), restarting decode processing if necessary (lines 5–6).

6.4.4 StreamPlay Phase III: Decode and Display

The third phase of client side PPS algorithm is where the video data are decoded and displayed.

6.4.4.1 sp_decode()

```
SP_DECODE(pps)
1  window ← QUEUE_PEEK_HEAD(pps.decode_windows)
2  while not HEAP_EMPTY(window.adus) and ...
3      do VIDEO_DECODE(HEAP_DELETE_MIN(window.adus))
4  if HEAP_EMPTY(window.adus)
5    then QUEUE_POP_HEAD(pps.decode_windows)
6  if QUEUE_LENGTH(pps.decode_windows) > 0
7    then SCHEDULE_IDLE(SP_DECODE, pps)
```

Since this chapter's main concern is the PPS details, the pseudo-code for *sp_decode* is just a sketch of final StreamPlay phase, omitting video specific processing. The receive phase will commit adaptation windows to the decode queue. Lines 1-3 of *sp_decode* do some decode processing for the the adaptation window at the head of the *pps.decode_windows* queue, consuming ADUs from the window's heap (line 3). The loop condition is partially unspecified (line 2), but the idea is

that the worst case execution time of sp_decode should be limited to maintain the responsiveness of the StreamPlay algorithm. In QStream, the video decoding step (line 3) includes the conversion from SPEG back to MPEG, and MPEG decode. It also schedules a timeout for each decoded frame to display it at the appropriate time.

When the adaptation window's heap becomes empty, the window is removed from the decode phase (lines 4–5). If there are more windows already in the queue, then *sp_decode* immediately schedules itself for execution at the next idle time (lines 6–7).

This concludes our description of the core PPS algorithm for unicast streaming. In the next section we describe the Priority-Progress multicast algorithms.

6.5 Priority Progress Multicast

Recall from Chapter 5 that PPM implements a multicast tree by treating each of the edges of the tree as a separate PPS session. The nodes of PPM trees fall into three distinct categories: the root, interior nodes, and leaves. In QStream, the root and leaves are and are implemented by StreamServ and StreamPlay. To support PPM, some extensions to StreamServ, relating to PPM flow control, are necessary. Otherwise, the PPS algorithm in StreamServ and StreamPlay is the same in unicast and multicast. For the interior nodes, we developed a separate program called MCastProxy. The basic role of MCastProxy is to forward PPS messages from the upstream edges to downstream edges in a best effort fashion. The arrival of a `WINDOW_START` message from the upstream edge is used as the trigger that initiates dropping of unsent SDUs from the previous window. MCastProxy also implements the PPM flow control mechanism, which is intended to prevent waste of upstream bandwidth when downstream links are constrained. The remainder of this section will consist of two parts: first, we describe the core algorithms in the MCastProxy implementation, and second, we describe the extensions in StreamServ to support PPM flow control.

6.5.1 MCastProxy

MCastProxy implements the interior node part of the PPM algorithm. This part of the algorithm has two main phases: *Stream Startup* and *Forward Windows*.

The Stream Startup phase is where new children arrive. The first message received from a

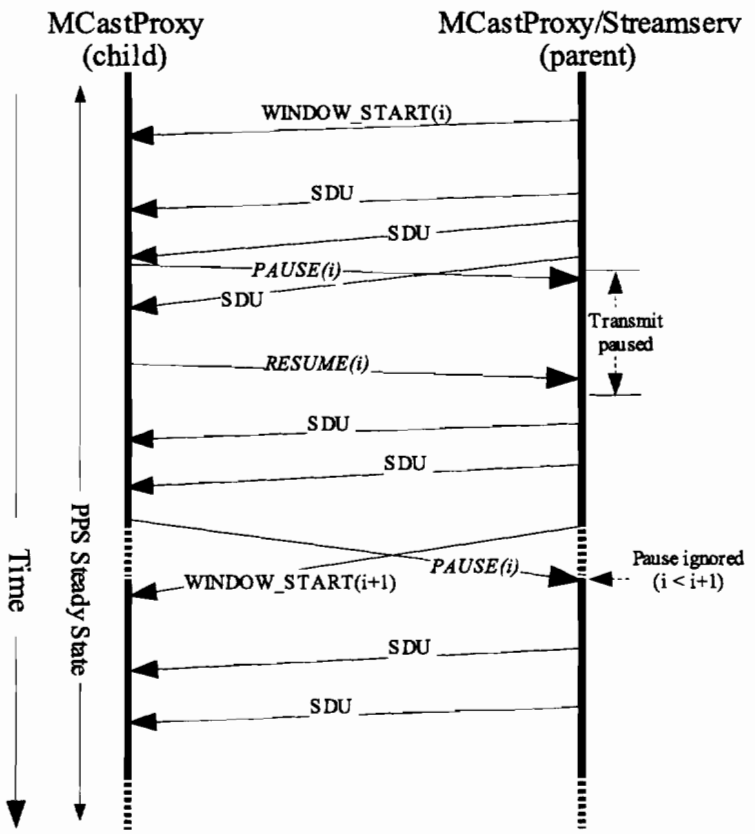


Figure 6.11: Sequence of Messages in a PPM session

child will identify which video the child is requesting. If a session is already started for that video, the child is added into that session. Otherwise a new session will be started, forwarding the startup process up the tree to the parent. In the case that the child is joining an already active session, the logic of the startup phase will delay completion of the forwarding phase for the new child until the next adaptation window begins.

The Forward Windows phase of the MCastProxy algorithm forwards adaptation windows to the children, dropping unsent SDUs from the current window when a new adaptation window starts. Since the messages arrive in priority order, the dropped SDUs are the lowest priority ones.

6.5.1.1 PPM Messages

The messages exchanged in PPM are a superset of those in PPS (see Section 6.2.2). Figure 6.11 depicts an example of the messages exchanged during the steady state of PPM. PPM adds two new messages to PPS, *PAUSE* and *RESUME*, which are used to implement PPM flow control.

When a child sends a *PAUSE* message to its parent, the parent will suspend transmission of *SDU* messages. The child will do this when it determines that the parent is getting too far ahead of all of the child's descendants. After a *PAUSE*, the child may later send *RESUME* message upstream, which happens if and when at least one of its children are sufficiently caught up. Alternatively, the *WINDOW_START* message implies a resume, since the start of a new window in PPM is what triggers dropping of unsent SDUs, which in turn forces all of the children to become caught up[18]. Because the transmission of messages in opposite directions may overlap in time, the *PAUSE* and *RESUME* messages contain the window number for which they are issued. The parent uses the window number to detect if they have crossed paths with a *WINDOW_START*, and if so, it ignores them. For example, in Figure 6.11 the second *PAUSE* message is ignored because it arrives after the parent has already started transmission of the next adaptation window.

6.5.2 Data Structures

The are two kinds of object in the MCastProxy algorithm. The first, called *MCastPpsSession*, is used to manage per session state. Each session consists of a set of two or more unicast Priority-Progress connections, one to the parent in the tree and the rest to the children. The second kind of object called *MCastChild* is allocated one per-child, it tracks the state for a child connection.

Figure 6.12 shows the *MCastPpsSession* object. The *parent_session* field holds a Qsf handle for the connection to the parent in the tree. Four lists are used to keep track of children (*MCastChild* objects) in various states: *startups*, *xmits*, *stalled*, and *paused*. The *startups* list contains children that are waiting for a new adaptation window so they can join the session. Recall that PPM uses a simple policy where new children can not join a window after it has already started, because some of the SDUs for that window may have already been discarded. Due to the dependencies between SDUs, the remaining SDUs may be of no use to new children without

[18]The exception to this rule is the base layer (maximum priority), which will not be dropped by PPM.

```
MCastPpsSession {
  QsfSession        parent_session;     // handle for TCP connection to parent

  // Children in various states
  List              startups;           // waiting for first window start
  List              xmits;              // streaming has started
  List              stalled;            // waiting for upstream
  List              paused;             // paused due to downstream request

  QsfMsg            open_request;       // cached for new arrivals
  McastMsg          open_response;      // cached for new arrivals
  QsfMsg            stream_start_request;  // cached for new arrivals
  McastMsg          stream_start_response; // cached for new arrivals
  Queue             mcast_msgs;         // messages to be forwarded
  Integer           max_ref_count;      // used to initialize sdu.ref_count
  Boolean           parent_started;
  Boolean           parent_paused;      // for limiting upstream bandwidth
  Integer           sdu_fill;           // how many unsent SDUs are there?
  Integer           win_num;            // most recent window
}
```

Figure 6.12: MCastProxy Per-Session State

the missing SDUs. The list will be used to start such children when the next *WINDOW_START* message arrives. The *xmits* list contains children to which transmission is active[19]. Once transmission to a child is active, it may be suspended for two reasons: lack of data to send, and a flow control request from the child. The *stalled* list contains children that have stopped transmission because they have completely caught up with the messages received so far. The *stalled* list is used to resume such children as soon as a new message arrives from the parent. The *paused* list contains children for which transmission has been suspended because they sent *PAUSE* messages. This list is used to resume them all immediately in the event that a *WINDOW_START* message arrives from the parent. Note that a child can be both paused and stalled at the same time. The *open_request*, *open_response*, *stream_start_request* and *stream_start_response* are cached copies of messages from the startup phase of the session. They are used in adding new children that arrive after the session has already started. The *mcast_msgs* field is a queue that stores all other PPS messages as they arrive from the parent in the tree. These messages will be forwarded to the children, each of which has a separate pointer into the *mcast_msgs* list to track what messages have been

[19]The list is used mainly for cleanup purposes.

sent so far. A reference count is used for each message to determine if it has been forwarded to all of the children. When the reference count reaches zero, the message will be removed from the queue. Otherwise, messages stay in the queue until the next window start message arrives. At that time, all non-base layer messages are dropped (recall that base layer SDUs are never dropped). The value *max_ref_count* stores the number of children that are eligible to forward each incoming message, this number is used to initialize the reference count for each message. When an message's reference count reaches zero, the message will be discarded. The *parent_started* field is used to record if a *START_REQUEST* has been sent to the parent. This is used to ensure that only one em START_REQUEST is forwarded upstream. The *parent_paused*, *sdu_fill* and win_num fields are used to implement the flow control part of the PPM algorithm, which aims to keep upstream bandwidth usage from exceeding the downstream bandwidth to the fastest of the children (see Section 5.1.2).

```
MCastPauseStatus :=
    UNPAUSED
  | PAUSE_PENDING
  | PAUSED

MCastChild {
  QsfSession       child_session;     // handle for downstream connection to child
  MCastPpsSession  pps;               // backpointer to session wide state
  McastMsg         mcast_msg;         // private pointer into pps.mcast_msgs
  Boolean          start_reqd;        // received stream start request
  Integer          win_num;           // what window is child working on
  MCastPauseStatus pause_status;      // downstream flow control state
}
```

Figure 6.13: MCastProxy Per-Session Child State

The per-child state object is shown in figure 6.13. The *child_session* field stores the Qsf handle for the network connection to the child. The *pps* field is simply a back-pointer to the session wide state. The *mcast_msg* field is a pointer into the *mcast_msgs* list of the *pps* object, the pointer marks the boundary between those messages that have already been forwarded to this child and those that have not. If a child has sent all of the messages received so far–meaning it is completely caught up with the parent–then this pointer will have a *NIL* value, and this child will be in the list of *stalled* children. The *start_reqd* field is used during session startup to record whether the *SESSION_START*

message has been received from the child. MCastProxy will only start forwarding adaptation windows after it arrives[20]. The *win_num* and *pause_status* fields are used to implement PPM flow control (see Section 5.1.2).

This completes the description of MCastProxy's data structures. The next two sections will describe the PPM algorithm which consists of two phases: startup and window forwarding.

6.5.3 MCastProxy Phase I: Stream Startup

In the first phase of the PPM algorithm, the startup of new sessions, and addition of new children to existing sessions are handled.

6.5.3.1 mp_child_acceptor()

```
MP_CHILD_ACCEPTOR(child_session)
1  child ← NEW(MCastChild)
2  child.child_session ← child_session
3  child.win_num ← −1
4  child.pause_status ← UNPAUSED
5  QSF_START_MSG_RECVS(child_session, dispatch_table)
```

MCastProxy's entry point is *mp_child_acceptor*, which is called when a new connection has arrived. Lines 1–4 instantiate a new MCastChild object. The protocol sequence will begin with the child sending an *OPEN_REQUEST* message to identify which video to transmit.

6.5.3.2 mp_child_recv_open()

```
MP_CHILD_RECV_OPEN(child, msg)
1  pps ← HASH_TABLE_LOOKUP(active_sessions, msg.filename)
2  if pps = NIL
3    then pps ← NEW(MCastPpsSession)
4         pps.parent ← QSF_CONNECT(LOOKUP_PARENT(open_file_msg.filename),
5                                  parent_methods, pps)
6         pps.open_request ← msg
```

[20]In the implementation, there can be delay because the child may not send the *SESSION_START* until it has fully established its connection to the Monitor. This is to ensure that the Monitor will record the complete history of a session.

```
7        pps.children ← LIST_PREPEND(parent.children, child)
8        HASH_TABLE_INSERT(active_sessions, msg.filename, pps)child.pps ← pps
9   pps.startups ← LIST_PREPEND(child, pps.startups)
10  if pps.open_response ≠ NIL
11    then QSF_SEND_MSG(child.child_session, pps.open_response)
12         child.open_sent ← TRUE
```

mp_child_open_file is dispatched when the *OPEN_REQUEST* message arrives. A hash table is used to map video names to their corresponding session objects. The table contains entries for all active sessions. Lines 1–2 check whether there is an active session for the requested video.

If the requested video does not have an active session, a new session is created (lines 3–8). Line 3 instantiates a new *MCastPpsSession* object. Lines 4–5 initiate a connection to the parent. The *lookup_parent()* function is called to provide the address of the parent in the multicast tree. lookup_parent() might be a hook into a multicast routing subsystem, such as End System Multicast [11]. In the current QStream prototype, the multicast topology is completely static, so the *lookup_parent()* call is simply a placeholder, returning an address fixed as a startup parameter of MCastProxy. Line 6 stores the *OPEN_REQUEST* so that it can be forwarded upstream after the connection to the parent has been established. Line 7 attaches the child to the just created session object. Line 8 enters the new session into the hash table, so that subsequent requests for the same video will join the existing session.

Line 9 enters the session into the *startups* list, which will be used later when the next *WINDOW_START* message arrives from the parent. If the child is joining an existing session for which the *OPEN_RESPONSE* has already arrived, then the *OPEN_RESPONSE* is forwarded to the child immediately, so that downstream players may initialize their displays as soon as possible (lines 10–12).

6.5.3.3 mp_child_recv_start()

```
MP_CHILD_RECV_START(child, msg)
1  pps ← child.pps
2  child.start_reqd ← TRUE
3  if pps.parent_started = FALSE
4    then pps.parent_started = TRUE
```

```
5        QSF_SEND_MSG(pps.parent_session, msg)
```

The *mp_child_recv_start* function is dispatched on the arrival of a *START_REQUEST* from downstream. Line 2 records the fact that it has arrived, which will allow this child to start streaming when the next *WINDOW_START* message arrives from upstream. If this is the first *START_REQUEST* to arrive for the session, then the message is forwarded upstream (lines 3–5).

6.5.3.4 mp_parent_connected()

```
MP_PARENT_CONNECTED(pps)
1  QSF_START_MSG_RECVS(pps.parent_session, dispatch_table)
2  QSF_SEND_MSG(pps.parent_session, pps.open_request)
```

mp_parent_connected is dispatched when the connection to the parent in the multicast tree is established. Line 1 enables message dispatching for incoming messages on the connection. Line 2 starts the upstream protocol sequence, by forwarding the open file request up to the parent.

6.5.3.5 mp_parent_recv_open()

```
MP_PARENT_RECV_OPEN(pps, msg)
1  pps.open_reply ← msg
2  for child in pps.startups
3      do if child.open_sent = FALSE
4          then child.open_sent ← TRUE
5               QSF_SEND_MSG(child.child_session, pps.open_response)
```

mp_parent_recv_open_file is dispatched when the *OPEN_RESPONSE* message has arrived from upstream. Line 1 caches the message in the session object, for use in the future when new children connect and request the same file. Lines 2–5 immediately forward the message to existing children.

6.5.3.6 mp_parent_recv_stream_start()

```
MP_PARENT_RECV_STREAM_START(pps, msg)
1  pps.stream_start_response ← msg
```

mp_parent_recv_stream_start is dispatched when the *START_RESPONSE* message arrives from upstream. StreamServ sends the *START_RESPONSE* when it begins the transmission of the first adaptation window. The message is stored, and will be forwarded when the first *WINDOW_START* arrives.

6.5.4 MCastProxy Phase II: Forward Windows

Once the session is established, the session enters the second phase of the PPM algorithm, which consists of forwarding adaptation windows to the children. A first-in first-out queue (*mcast_msgs*) is used to store messages as they arrive from upstream. Each child maintains a private pointer into the queue, where the pointer signifies the next message to transmit for that child.

6.5.4.1 mp_parent_recv_win_start()

```
MP_PARENT_RECV_WIN_START(pps, msg)
 1  pps.win_num ← msg.win_num
 2  pps.parent_paused = FALSE
 3  pps.sdu_fill = 0
 4  pps.max_ref_count = LIST_LENGTH(pps.xmits)
 5  for child in pps.startups
 6      do if child.start_reqd = TRUE
 7          then pps.max_ref_count = pps.max_ref_count + 1
 8  msg.sent_flag ← FALSE
 9  msg.ref_count ← pps.max_ref_count
10  ENQUEUE(pps.mcast_msgs, msg)
11  for child in pps.startups
12      do if child.start_reqd = TRUE
13          then pps.startups ← LIST_REMOVE(pps.startups, child)
14               pps.xmits ← LIST_PREPEND(pps.xmits, child)
15               QSF_SEND_MSG(child.child_session, pps.start_response)
16               MP_CHILD_SEND_MSG(child)
17  MP_CHILDREN_WAKEUP(pps, msg)
```

mp_parent_recv_win_start is dispatched when a *window start* message arrives from upstream, each adaptation window begins with such a message. Line 1 sets the window counter for the session. This will trigger the children to skip forward to the new window as soon as possible (see Section 6.5.4.6). Lines 2–3 reset the *parent_paused* and *sdu_fill* values, since *WINDOW_START*

messages restart the flow control process. Line 4 sets the starting value for SDU reference counter to include children that have already started, while lines 5–7 add children that will be newly started due to the arrival of this message. Lines 8–10 initialize the reference count and send status for the *WINDOW_START* message itself, and enter it into the transmission queue. Lines 11–16 initiate the transmission phase for newly joined children. Finally, the call to *mp_children_wakeup* (described next) will restart the transmission logic for children that were previously paused due to flow control, or stalled due to a lack of messages to send.

6.5.4.2 mp_children_wakeup()

```
MP_CHILDREN_WAKEUP(pps, msg)
1  for child in pps.paused
2      do old_status ← child.pause_status
3         child.pause_status ← UNPAUSED
4         pps.paused = LIST_REMOVE(pps.paused, child)
5         if old_status = PAUSED and child.mcast_msg
6           then MP_CHILD_SEND_MSG(child)
7  MP_CHILDREN_UNSTALL(pps, msg)
```

mp_children_wakeup is simple helper function for *mp_parent_recv_win_start* that iterates through the list of previously paused children and restarts them. The check in line 5 prevents re-starting children that haven't actually paused yet (their *pause_status* is still *PAUSE_PENDING*). The call to *mp_children_unstall* will resume those children that were stopped because they ran out of messages to send.

6.5.4.3 mp_children_unstall()

```
MP_CHILDREN_UNSTALL(pps, msg)
1  for child in pps.stalled
2      do pps.stalled ← LIST_REMOVE(pps.stalled, child)
3         child.mcast_msg ← msg
4         if child.pause_status = UNPAUSED
5           then MP_CHILD_SEND_MSG(child)
```

The *mp_children_unstall* helper function is called when a *WINDOW_START* message or a *SDU* message arrives. It simply resumes transmission for all children that had previously caught up to

the parent completely and had run out of messages to send. Note, the check in line 3 ensures that the child is not restarted if it is also paused due to PPM flow control.

6.5.4.4 mp_parent_recv_sdu()

```
MP_PARENT_RECV_SDU(pps, sdu_msg)
 1  if sdu_msg.priority < MAX_PRIORITY
 2    then pps.sdu_fill ← pps.sdu_fill + 1
 3  if pps.sdu_fill ≥ SDUS_HIGH_WATER and not pps.parent_paused
 4    then pps.parent_paused ← TRUE
 5         pause_msg.win_num = pps.win_num
 6         QSF_SEND_MSG(pps.parent_session, pause_msg)
 7  sdu_msg.ref_count ← pps.max_ref_count
 8  sdu_msg.sent_flag ← FALSE
 9  QUEUE_PUSH_TAIL(pps.mcast_msgs, sdu_msg)
10  MP_CHILDREN_UNSTALL(pps, sdu_msg)
```

mp_parent_sdu is called upon each SDU message arrival. Lines 1–6 are part of PPM flow control. Lines 1–2 update the *sdu_fill* value to count the number of non base layer messages that are as yet unsent by any child. When too many SDUs are backing up behind all the children (line 3), the parent's status is changed and a *pause message* is sent upstream (lines 4–6).

Lines 7–8 initialize the reference counter and flow control flag for the SDU and then enter it into the message transmission queue. The transmission logic for the children loops on the contents of the queue. Line 9 adds the message to the tail of the transmission queue. A child enters *stalled* status when it has transmitted all of the messages available. Lines 10 restarts any such children, since there is now a message available for them to send.

6.5.4.5 mp_child_send_msg()

```
MP_CHILD_SEND_MSG(pps, child)
1  if child.mcast_msg.type = WINDOW_START
2    then child.win_num ← child.mcast_msg.win_num
3  QSF_SEND_MSG(child.child_session, child.msg, QSF_CHILD_SENT_MSG)
```

mp_child_send_msg starts transmission of the current message for a child. Lines 1–2 update the *win_num* field if the child is starting a new window (this field is used elsewhere to detect if

the child should catch up to a new window position). Line 3 actually initiates transmission of the current message for this child.

6.5.4.6 mp_child_sent_msg()

```
MP_CHILD_SENT_MSG(pps, child)
1  MP_CHILD_NEXT_MSG(pps, child)
2  if child.win_num < pps.win_num
3    then while child.msg.priority < MAX_PRIORITY
4           do MP_CHILD_NEXT_MSG(pps, child)
5  if child.pause_status = PAUSE_PENDING
6    then child.pause_status ← PAUSED
7    else if child.mcast_msg
8           then MP_CHILD_SEND_MSG(pps, child)
```

mp_child_sent_msg is dispatched when the transmission of the current message to a particular child is complete.

A helper function *mp_child_next_msg* (described next) handles the details of advancing the child's private pointer to the next message in the transmission queue (line 1). If the child notices that it is working on an older window than the parent, then it will skip any non-base layer messages to try to catch up (lines 2–4). Line 5 checks whether a pause is pending for this child due to the previous arrival of a *PAUSE* message. If so, the status for the child is updated (line 6) and transmission stops. Otherwise, if there is a message to send, it begins sending right away (lines 7–8).

6.5.4.7 mp_child_next_msg()

```
MP_CHILD_NEXT_MSG(pps, child)
1  cur ← child.mcast_msg
2  next ← child.mcast_msg.next
3  if cur.sent_flag = FALSE
4    then cur.sent_flag ← TRUE
5         if child.win_num = pps.win_num and cur.priority ≠ MAX_PRIORITY
6           then pps.sdu_fill ← pps.sdu_fill − 1
7         if pps.sdu_fill ≤ LOW_WATER_THRESH and pps.parent_paused
```

```
 8            then pps.parent_paused ← FALSE
 9                 resume_msg.win_num = pps.win_num
10                 QSF_SEND_MSG(pps.parent_session, resume_msg)
11  cur.reference_count ← cur.reference_count − 1
12  if cur.reference_count = 0
13    then QUEUE_REMOVE(pps.mcast_msgs, cur)
14  child.mcast_msg = cur.next
15  if child.mcast_msg = NIL
16    then pps.stalled ← LIST_PREPEND(child, pps.stalled)
```

mp_child_next_msg is responsible for advancing the child's private pointer into the transmission queue (line 14). In doing so, it also performs the bulk of the work for PPM flow control and message cleanup.

Lines 3–10 treat flow control duties. The *sent_flag* is used to ensure that the *sdu_fill* value is decremented once for every non-base layer SDU that has been transmitted to at least one child (lines 5–6). If the parent has been previously paused, then lines 7–10 take care of resuming the parent when the *sdu_fill* value hits the *LOW_WATER_THRESH* value. Lines 11–13 maintain the reference count and free messages when the reference count hits zero.

Lines 15–16 treat the case that there is no next message, so the child enters the *stalled* list. The child will resume when the next available message arrives from the parent.

6.5.4.8 mp_child_recv_pause()

```
MP_CHILD_RECV_PAUSE(child, pause_msg)
1  pps ← child.pps
2  if pause_msg.win_num = pps.win_num and child.pause_status ≠ PAUSED
3    then if child.mcast_msg ≠ NIL
4           then child.pause_status = PAUSE_PENDING
5           else child.pause_status = PAUSED
```

mp_child_recv_pause is dispatched on the arrival of a *PAUSE* message from downstream. Line 2 checks that the message matches the current adaptation window and that the child is not already paused. If not, the *PAUSE* message can be ignored. Line 3 checks whether this child is currently forwarding messages. If so, line 4 sets the child's *pause_status* to *PAUSE_PENDING*, which will signal transmission to stop after the current message is done (see Section 6.5.4.6). Otherwise, line

5 marks the child as *PAUSED* immediately. Message forwarding to this child will be suspended until either a *RESUME* message arrives from downstream or a *WINDOW_START* message arrives from upstream.

6.5.4.9 mp_child_recv_resume()

```
MP_CHILD_RESUME(child, resume_msg)
1  pps ← child.pps
2  if resume_msg.win_num = pps.win_num
3    then old_status = child.pause_status
4         child.pause_status ← UNPAUSED
5         pps.paused ← LIST_REMOVE(pps.paused, child)
6         if old_status = PAUSED and child.mcast_msg
7           then MP_CHILD_SEND_MSG(child)
```

mp_child_resume is dispatched when a *RESUME* message arrives from downstream. The message will be ignored if it doesn't match the current adaptation window (line 2). Lines 3–7 handle updating the child's *pause_status* to *UNPAUSED* and if a message is ready, resume forwarding to the child immediately (lines 6–7).

This concludes our description of the PPM algorithm parts of MCastProxy. The next section will describe the modifications to StreamServ necessary to support PPM flow control.

6.5.5 StreamServ: Multicast Extensions

A PPM tree consists of three node types: the server at the root of the tree, the interior nodes, and the video players at the leaves of the tree. The bulk of PPM algorithm is located in the interior nodes, as implemented in MCastProxy that was described in the previous section. The server and the players in PPM are essentially the same as for unicast PPS. The only significant component of the PPM algorithm outside of interior nodes is support for flow control required in the root of the tree, that is, in StreamServ. In this section, we describe the additions to StreamServ that implement the required support for PPM flow control.

```
ServPauseStatus :=
    UNPAUSED
  | PAUSE_PENDING
  | PAUSED

ServPpsSession {
  ...
  ServPauseStatus xmit_status;        // for PPM flow control
  Integer         xmit_window_number
  ...
}
```

Figure 6.14: StreamServ PPS Session Object: extensions for multicast

6.5.6 Data Structures

Figure 6.14 shows the additional data structures required to add PPM flow control support into StreamServ (see Section 6.3.1 for the original data structures). The *xmit_status* field tracks the flow control state of the child, like the *pause_status* value used in the MCastProxy algorithm. The *xmit_window_number* is used to make sure that *PAUSE* and *RESUME* messages do not take effect in the case that they cross paths with a *WINDOW_START* message.

6.5.6.1 ss_child_send_sdu_head()

```
SS_CHILD_SEND_SDU_HEAD(pps)
1  win ← QUEUE_PEEK_HEAD(pps.xmit_windows)
2  msg ← HEAP_PEEK_MIN(win.sdus)
3  win.adu_count ← 0
4  if pps.xmit_status = PAUSE_PENDING
5    then pps.xmit_status = PAUSED
6    else QSF_SEND_MSG(pps.child_session, msg, SS_CHILD_SEND_ADU)
```

The *ss_child_send_sdu_head* function is slightly extended for PPM. The original version was described in Section 6.3.4.2. The version here is the same in the first three lines. The added code is lines 4 and 5, which make the transmission of the SDU conditional on whether the a *PAUSE* message has arrived from downstream.

6.5.6.2 ss_win_xmit_expire()

```
SS_WIN_XMIT_EXPIRE(win)
1  win.xmit_timeout ← NIL
2  SS_UNPAUSE_XMIT(win.pps)
```

The *ss_win_xmit_expire* is also slightly modified for PPM (the original is in Section 6.3.3.7). Rather than waiting for the child to send a *RESUME*, the transmission of the new window is started immediately by the call to *ss_unpause_xmit* (described next).

6.5.6.3 ss_unpause_xmit()

```
SS_UNPAUSE_XMIT(win)
1  old_status ← pps.xmit_status
2  pps.xmit_status ← UNPAUSED
3  if old_status = PAUSED
4    then SS_SDU_NEXT(pps, QUEUE_PEEK_HEAD(pps.xmit_wins))
```

ss_unpause_xmit restarts transmission for the child if it was paused. This function is called either when a *RESUME* message arrives from downstream (described next) or when the timer event expires to start a new adaptation window (described above).

6.5.6.4 ss_child_recv_pause()

```
SS_CHILD_RECV_PAUSE(pps, pause_msg)
1  if pause_message.window_number ≠ pps.xmit_window_number
2    then if not QUEUE_EMPTY(pps.xmit_wins)
3           then win ← QUEUE_PEEK_HEAD(pps.xmit_wins)
4                if pps_win.xmit_timeout
5                  then pps.xmit_status ← PAUSE_PENDING
```

ss_child_recv_pause is dispatched when a pause request is received from the multicast tree. The check in line 2 ensures the request matches the adaptation window currently in the transmission phase. If so, line 3 sets *xmit_status* to reflect that pause has been requested. This will signal *ss_child_send_sdu_head* to pause transmission before starting the next SDU.

6.5.6.5 ss_child_recv_resume()

SS_CHILD_RECV_RESUME($pps, pause\_msg$)
1 **if** $pause\_message.window\_number = pps.xmit\_window\_number$
2 **then** SS_UNPAUSE_XMIT(pps)

ss_child_recv_resume is dispatched when a resume request is received from the multicast tree. The check in line 2 ensures the request matches the adaptation window currently in the transmission phase. Line 4 changes the transmission status to unpaused. Line 6 initiates transmission of the next SDU, but only if the transmission phase actually reached the paused state since the prior pause request (line 5).

This completes our description of the PPS and PPM algorithms. The following section will briefly describe the support for remote monitoring of experiments in the QStream implementation.

6.6 The QStream Monitor

In this section, we give an overview of the QStream Monitor (hereafter referred to as the Monitor), which is a separate program in QStream that gathers data in real-time from the other QStream programs, and displays (or records) them as signals in a set of software oscilloscopes (or in graphs generated by gnuplot). At present, the Monitor contains more than a half dozen scopes, and close to a hundred different signals. The signals fall into several categories, such as information about SPEG, information about the PSS and PPM, and information about traffic generated by *mxtraf*. The Monitor facility has played an essential role in the development and performance evaluation tasks of this dissertation. The Monitor has also been a prominent feature when we show the QStream software in demonstrations and research talks.

The Monitor is implemented as a network server. The other QStream programs establish a session with the Monitor when they start. As they perform their tasks (e.g. PPS), they send data samples to the Monitor, typically in the form of timestamped, attribute-value tuples. In addition to the server, the Monitor also provides a client-side library which provides a stub API for data collection. This API hides the work of network communication from the instrumented programs.

Unlike PPS, the Monitor protocol is quite different in that it multiplexes sessions into a single connection. That is to say, each QStream program establishes at most one network connection

to the Monitor, even when several logical sessions (PPS streams or mxtraf flow) are monitored. For example, if StreamServ accepts several PPS sessions at the same time, coming from different instances of StreamPlay, the monitoring data for all of the PPS sessions are sent by StreamServ over single connection to the Monitor. To sort out the data, every attribute-value tuple carries with it a unique session id. We use UUIDs for these IDs, which have the property that they can be generated without global communication, and without fear of collisions. This also allows data pertaining to the same session, but originating from different hosts to be combined in single view, for example to have signals for the same PPS session, but originating separately from StreamServ and StreamPlay, displayed in the same oscilloscope. Since the data originating from different programs may take different network paths, and hence different amounts of time to arrive, the Monitor uses a priority-queue (implemented via a heap) to buffer incoming samples, and sort them for display according to their timestamps. This allows certain signals to be displayed with perfect synchronization, even when their data samples originate from different hosts. The buffer is managed in the Monitor according to a *late-offset* parameter, which is used by the monitor to add a fixed delay between the timestamp values and when they are displayed. If the samples arrive at the monitor later than their timestamp adjusted by this offset, then they are considered late and dropped. In practise, we set the delay to a few seconds. This allows the TCP session to combine samples into full sized segments, which is good for keeping the TCP packet overheads minimized.

In addition to the graphical oscilloscope, the Monitor can also store data to a database for offline processing. For example, we use this feature to generate plots with *gnuplot*. In this case, the timestamp is used as the sort key for the database. As samples arrive, they are inserted into the database (in this case, samples never need to be dropped). At the end of an experiment, the database will have all the samples available in timestamp sorted order, and an in-order traversal can be used extract the data for plotting.

Chapter 7

Streaming Evaluation

In this chapter, we present an experimental evaluation of Priority-Progress Streaming (PPS) and Priority-Progress Multicast (PPM), based on a real implementation of them in our prototype streaming system, QStream (Quasar Streaming). As we described in Chapter 6, QStream includes significant internal instrumentation that generates data for various quantitative assessments. In our experiments, we use a combination of simulated, emulated and live networks, as appropriate to the questions we are trying to answer. In the next section, we expand upon the advantages and disadvantages of each type of network. The remaining sections present three major groups of results. The first group addresses rate and quality metrics of SPEG video as they vary over the timeline of a video, expanding upon some of the basic results presented in Chapter 3. The second group of results addresses the performance of PPS in unicast streaming scenarios. The third group of results treats multicast streaming scenarios.

7.1 Experimental Approach

In distributed systems research, there are three general approaches to conducting experiments: simulation, emulation, and live experiments. Each approach remains popular because they each occupy different points in a space defined in terms of realism, control, and ease-of-use [89]. The results presented in this chapter are based on a mixture of simulation and emulation. For reasons that we describe below, our first preference is the emulation approach. To support experiments in an emulated network setting, we have invested a significant amount of effort in instrumenting the QStream code so as to produce a prolific set of measurements.

Of the three experimental approaches, simulation is the most attractive approach in terms of

control and ease-of-use. In simulation, all entities of an experiment, end-hosts, network nodes, and links, are modelled using discrete-event programming techniques. Simulation offers complete control, and thus repeatability, and is generally the easiest to use because experiments can be conducted entirely on a single machine. Simulation is also attractive because a small number of tools, especially `ns` and `ns2` [21, 82], have achieved de-facto standard status in the networking community, which helps to make sharing of research results with the community more effective. Simulation's greatest weakness is realism. Although simulation tools go to great lengths to maximize realism, they always abstract some details of real implementations. In the case of `ns`, the de-facto standard for network simulation, one such detail is the socket buffer used in real OS protocol stacks. In a real in-kernel protocol stack, the socket buffer is used both for the purpose of supporting TCP retransmission, and for reducing CPU overhead due to user-system transitions. For the purposes of evaluating most networking topics such as routing protocols, congestion control mechanisms, etc., the socket buffer does not have significant effects, so the simulation does not model the socket buffer at all. However, for streaming applications using a congestion controlled transport such as TCP, the socket buffer has the dominant impact on end to end latency [31]. To address this issue, we might have chosen to try and extend the simulators to encompass the missing socket buffer dymamics, or to find other simulators that do model socket buffers. However, it seems clear that even with such extensions, it would be prudent to compare simulation results to some baseline based on measurements from a real implementation. So, we defer simulation of PPS to future work, and instead focus on the real system options, network emulation and live network experiments.

Where simulation might be thought of as attempting to recreate the Internet in a single host, emulation can be thought of as extending the idea to recreate the Internet in a single laboratory testbed. The basic idea of emulation is to replace wide area network links with local ones, using routing software for each link to emulate the wide-area counterparts' behaviour in terms of delay, rate, and queuing characteristics. Additionally, traffic generation tools may be used to try and replicate the traffic conditions that occur on the real Internet. With the dramatic drops in hardware costs and the increased breadth of available open source software in recent years, it has become feasible for research institutions to construct testbeds capable of emulating a relatively large range of network scenarios. Potentially, emulation can be much more realistic than simulation, because

a much larger fraction of software and hardware involved, relative to simulation, are the same as in the Internet. Experiment control and ease of use are more difficult since many of the important control parameters must be separated across the many machines that makeup the testbed. Furthermore, since multiple systems are involved, a degree of non-determinism is introduced that makes exact repeatability impossible. However, with appropriate setup, experiments in an emulation testbed can be expected to yield acceptably consistent results under repetition. Scripting and other forms of automation can mitigate the ease-of-use issues. The balance between realism and control affored by emulation make it our main choice of experimental method.

Live experiments, where experiments are conducted over the Internet, are at the other extreme from simulation in terms of control, ease-of use, and realism. Live experiments obviously hold the potential for the most realistic results. However, they are the most difficult from the perspective of control and ease of use. A live experiment cannot include control over traffic of other users, so the results from an experiment may be impossible to repeat. A live experiment is also the most difficult to co-ordinate because of issues of physical separation between components of the experiment.

As mentioned above, we are very concerned with realism in our work, so we prefer to conduct our development and experiments under an emulation network, which gives a balance between realism and control. We did chose to use simulation to generate baseline performance measurements for a selection of other approaches previously described in the literature. Emulation is generally more realistic than simulation because it can reveal performance issues that might get missed in simulation. In the case of the baselines, we felt that simulation was acceptable, because by using simulation we would at worst overstate the performance of the baselines and understate the advantages of our approach.

In the next section, we describe the emulation testbed that we constructed for our experiments. The results of these experiments will be the main focus of our evaluation. However, we note that we have tested the QStream prototype regularly over live networks, including over cable-modem based broadband and 802.11 wireless links. Although we will not present any data here from live network tests, we can report that the behaviour of QStream in live network conditions has conformed to the results we will present from our emulation testbed experiments. Furthermore, all of the software components used in our experiments, including those we developed ourselves,

are publicly ally available, so it should be feasible for other researchers to validate and extend upon our results.

7.2 Network Emulation Testbed Setup

7.2.1 Testbed Hardware

The hardware of our experimental testbed consists of a rackmounted cluster of commodity hosts and a Gigabit network. The cluster contains 12 identical 1U sized hosts, which we use in the roles of end hosts and emulation routers in the testbed. The 1U hosts were manufactured by SuperMicro Inc., model name SuperServer 6012P-6. Each host is a dual processor machine configured with 1.8Ghz Pentium 4 Xeon processors, a pair of Gigabit NICs (Intel e1000), a single 100Mb NIC (Intel e100), 1GB of RAM, and three 120GB disks (WD Caviar). At the time the cluster was constructed (Summer 2002), commodity PC hardware was generally unable to reach the full potential of Gigabit links due to limitations of system memory and PCI bus implementations. One of the reasons we chose the SuperServer 6012P-6 models was because they were among the first to ship with the Intel E7500 server chipset, which has memory and IO bus capacities sufficient (3MBs each) to allow the Gigabit NICs to run at full speed.

The cluster hosts are connected to each other via a CISCO Catalyst 4000 Gigabit switch. We use VLANs to partition the set of network links into separate virtual networks, whereby the switch enforces physical separation of traffic. We used the 100Mb links for cluster management, on a separate VLAN from the Gigabit links. The Gigabit links were also partitioned into two or more VLANs as required by the desired topology for each particular experiment.

7.2.2 Testbed Software

The operating system we use on the hosts is RedHat Linux version 8.0. The kernel version is RedHat 8.0's 2.4.18, with an small additional patch of our own for our `TCP_MINBUF` option to `setsockopt()`, described in Section 4.4.3. The `TCP_MINBUF` option is only used in a subset of our experiments. RedHat kernels include a number of patches beyond the version maintained by Linus Torvalds. Notably, this RedHat kernel included a low-latency patch and a patch to allow finer granularity kernel clock (1ms).

To emulate wide area links, we use NISTnet [63]. NISTnet is a software package that enables a Linux router to mimic various characteristics of a wide area path, such as delay, drop probability, and bandwidth limitations through rate shaping or queue bounds.

For many of our experiments, we use a traffic generator to mix competing traffic into the network path with PPS flows, to ensure that PPS is robust in the face of busy links. Creating traffic in a way that reflects what happens on the Internet is a non-trivial task. We examined several available traffic generation tools, but then decided to write our own, which is called `mxtraf`. The reason we resorted to writing `mxtraf` was that the existing tools were either too simplistic in the type of traffic they generated, or they were too slow to generate significant amounts of traffic. For example on the simple side, the `ttcp` package only generates a single flow at a time. In contrast, the `SURGE` traffic generator goes to great lengths to generate flows that accurately model the behaviour of individual web users, in terms of various statistical distributions such as file size, time between file accesses, etc. Unfortunately, when we tried `SURGE` we found that it required a considerable number of CPUs to generate just one or two megabits of traffic. In their study of aggregate traffic performance, Iannaconne *et al.* suggest that a mix of flows can reasonably approximate the dynamics of busy Internet routers [39]. In particular, they suggest that the traffic consists of three classes of flow: long-lived TCP flows, short-lived TCP flows which repeat, and a small amount of non-TCP traffic. However, their study used simulation. Our program, `mxtraf`, allows a similar mix of real flows to be injected into an emulation testbed. Although we sacrifice some of the accuracy of `SURGE`, `mxtraf` can scale the number of flows up so as to generate much more realistic amounts of traffic. Together, NISTnet and `mxtraf` are able to emulate a broad range of network scenarios.

In the following sections we present measurements taken from QStream in our experimental testbed.

7.3 Adaptive Video

Recall, in Chapters 1 and 2, we gave our motivations for an adaptive approach. Recapping briefly, the two basic reasons we need to adapt are that video bitrates are variable over time, and that available network bandwidth also varies. Video bitrates vary due to the use of compression. Although

variable bitrates pose a challenge for streaming, the resource savings that result from compression are significant enough to justify its use. In this section, we will present some bitrate measurements to illustrate the volatilty of video bitrates from several perspectives. We will also re-affirm the first claim of our thesis statement: through informed dropping, it is possible to cover a very wide range of quality-rate combinations and with fine granularity.

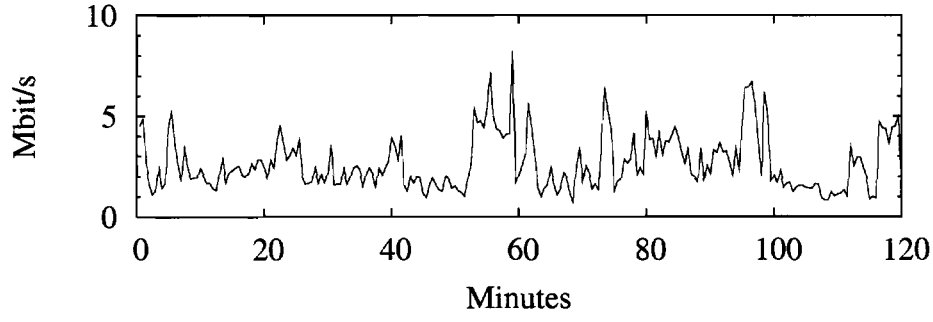


Figure 7.1: Maximum video rate for full 2 hour video

We expect that the need for adaptation grows stronger for longer duration content. The simple intuition behind this expectation is that over a longer period there are more chances that mismatches between video and network will occur. In Figure 7.1, we show the bitrate requirements (at maximum quality) for an SPEG encoding of a full length movie (approximately 2 hours). The movie in this case is "Crouching Tiger Hidden Dragon". The SPEG file was created in a two step process. First the DVD version of the movie was transcoded from the MPEG-2 on the DVD to MPEG-1. The MPEG-1 was transcoded to SPEG using Quasar software[1]. It should be noted that the data in Figure 7.1 is smoothed to 1 minute intervals in order to make it easier to read. Even with this level of smoothing the bitrate varies quite dramatically, spanning a wide range from about 1Mbps to close to 10Mbps. Although we do not present the data here, our own experience and other anectdotal evidence suggests that the bursty profile of this movie is reasonably representative of the level of burstiness in other movies (e.g., Feng *et al.* present a survey of data gathered from over 100 of the most popular DVDs which confirms this [23]).

Another characteristic of major interest is the granularity of adaptation afforded by SPEG.

[1]We used the transcoder from the Quasar pipeline, which is the predecessor to the QStream prototype.

Section 3.3 summarized granularity issues of SPEG. Here, in Figures 7.2 and 7.3, we show the granularity over time. Each line in the figure represents the video bitrate at each of 16 priority thresholds, normalized to the maximum video bitrate. Each priority level represents a video quality, with a corresponding frame rate and level of spatial detail (number of SPEG layers). In these figures, the bitrate is computed for each PPS mapper window. Recall that, in PPS, the timeline of the video is divided into mapper windows for the purpose of prioritization. A mapper window consists of one or more SPEG GoPs, and a PPS adaptation window consists of one or more mapper windows. In these figures, the mapper windows are 0.5 seconds each. Refer to Chapter 4 for more detailed descriptions of these subdivisions of the video timeline. Figure 7.2 shows the normalized bitrates by priority over the course of the entire video.

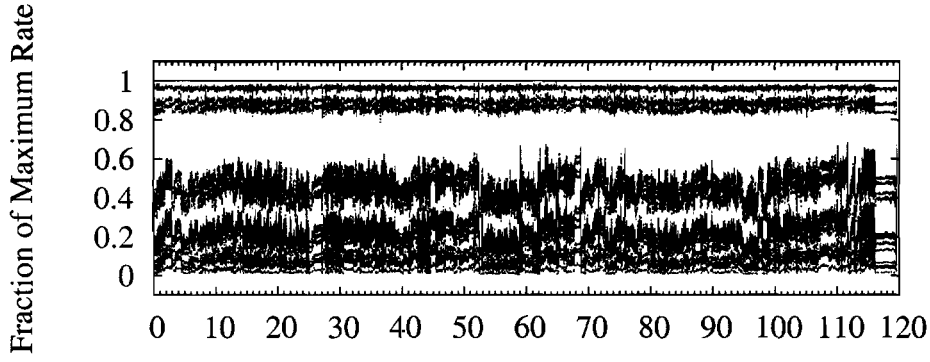


Figure 7.2: Relative video rates by priority of full 2 hour video

Figure 7.3 shows the same data as Figure 7.2 narrowed down to a selected 30 second interval. At this smaller timescale, it is possible to see the rates of individual mapper windows, noting that there are visible changes from one window to the next.

These figures reinforce a couple of the basic messages from Chapter 3. SPEG covers a wide range of rates, and with fine granularity. However, there are some large gaps, due to the limited spatial scalability in SPEG (only 4 spatial levels). The relative distribution of bitrates by priority level is interesting too. If the bitrate variations were the same across priority levels, then the lines in these figures would have a constant spacing across the whole timeline. Figures 7.2 and 7.3 show that there are in fact some fairly substantial variations over time in the per-priority bitrates. This is of interest in relation to video smoothing techniques (we referred to this area of the literature in

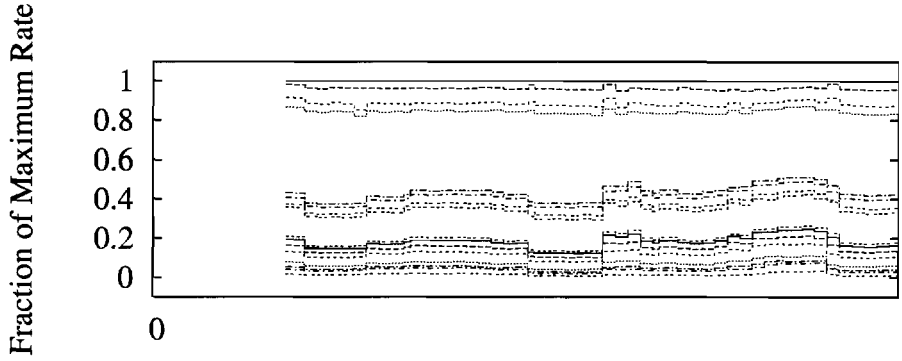


Figure 7.3: Relative video rates by priority of a selected 30 second interval

Section 2.1.2). Recall that those techniques use a priori knowledge of the video's bitrate profile to guide the buffering process during transmission, in such a way as to smooth the transmission rate requirements of the video stream. However, the smoothing algorithms in the literature were restricted to the assumption of a single target rate. In that case, the goal of smoothing was to aid in provisioning network resources. With adaptive streaming and scalable video, the goal of smoothing would be to improve the consistency of video quality. The most direct way to extend existing smoothing algorithms to scalable video (with multiple rates) is to calculate the smoothing plan based on the maximum rate. However, the variations in relative rates that we see above suggest a buffering plan that effectively smooths the video at the maximum quality level will not smooth the rate as well for lower quality levels.

As we explained in Chapter 4, PPS uses adapation windows to match the video to the available network bandwidth. The window scaling feature of PPS, which expands the size of the windows over time, has a simultaneous smoothing effect in relation to the network bandwidth *and* the video bitrate. Unlike the smoothing techniques mentioned above, the window scaling mechanism does not use rate profiles in any way, because window scaling is intended to smooth the network rate and our assumption is that the network rate is effectively unpredictable. It might be viewed as a kind of side-effect that PPS window scaling also smoothes the video rate. We believe it would be relatively straightforward to extend window scaling to take advantage of a priori video rate knowlege, for example by making minor adjustments to the Critical Bandwidth Allocation (CBA) algorithm by Feng et al. [26]. However, as the results later in this chapter will show, the PPS

window scaling already achieves a significant degree of smoothing.

To summarize this section, our measurements verify that, through our approach to video, a wide range of adaptation is supported with fine granularity, which is the first of the claims in our thesis statement. The measurements also confirm the first component of our motivation for adaptive streaming, that is, that video requirements are very bursty. In the rest of this chapter, we show how PPS treats these bursty aspects of video bitrates with the goal of providing the best possible user experience.

7.4 Unicast Streaming

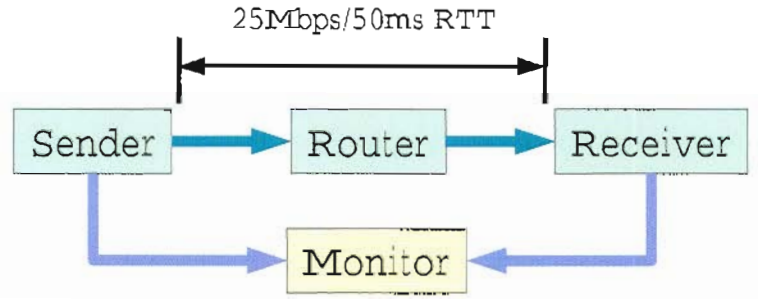


Figure 7.4: Unicast Experiment Setup

In this section, we examine the performance of PPS in streaming the full-length movie described above through a network saturated with competing traffic. The goals of the experiment are to evaluate the robustness, utilization, and consistency of PPS, and to compare them with existing streaming protocols. Figure 7.4 depicts the configuration of this experiment. This is a basic dumbell setup where a router running `NISTNET` is used to impose a 25Mbs rate limitation and a 50ms RTT delay between the sender and the receiver. The rate limitation in `NISTNET` is imposed by two mechanisms. The maximum capacity of the forwarding queue is set according to maximum number of packets in flight given the 50ms RTT and the 25Mbs rate target. Also, the maximum speed of the router's incoming link is limited using a token bucket style shaper (which ensures that burst rates do not exceed our modelled link capacity).

For the entire duration of the experiments, the network is *saturated* with competing traffic. We use `mxtraf` (see Section 7.2.2) to generate the various mix of competing traffic, which is made up of non-responsive UDP traffic (10%), short-lived (20Kb) TCP flows (~60%), and long-lived infinite-source TCP flows (~30%), similar to measurements reported in [39]. The overall `mxtraf` workload was balanced across a set of hosts, including the same hosts used for the PPS streams. We use the QStream implemenation of PPS to stream a two hour video through this saturated network path. We measure the performance of PPS in two cases, the first using a fixed adaptation window, and the second with the PPS adaptation window scaling feature enabled.

To provide baseline performance references, we simulate two existing streaming algorithms

assuming they are given the same video and available bandwidth during our PPS experiments. The first algorithm is based on the Berkeley CMT [58], and the second on Feng's technique [24]. In the CMT algorithm, layered data is transmitted from low to high quality upto a fixed duration (2s) ahead of the current play point. The Feng priority-algorithm allows the workahead to grow to a much larger size (60s) in order to increase resiliency to short term rate fluctuations.

Figure 7.5 shows the transmission rate over time of a TCP session used by PPS. The competing traffic generated by `mxtraf` ensures that the video stream never gets enough bandwidth to reach the maximum quality (refer to Figure 7.1). We were careful in QStream to ensure that PPS acts as a greedy source toward TCP, so that the transmission rate is exclusively determined by TCP's congestion control. This rate profile was used as an input to our simulations of the CMT and Feng algorithms. Our intent is to stress the adaptive capabilities of each of the streaming algorithms, and to compare them in terms of the resulting video quality. The high volatility of the session rate in Figure 7.5 demonstrates the second part of our overall motivation to investigate adaptive streaming. Even though we have a constant mix of background traffic, the session still experiences significant network rate variations.

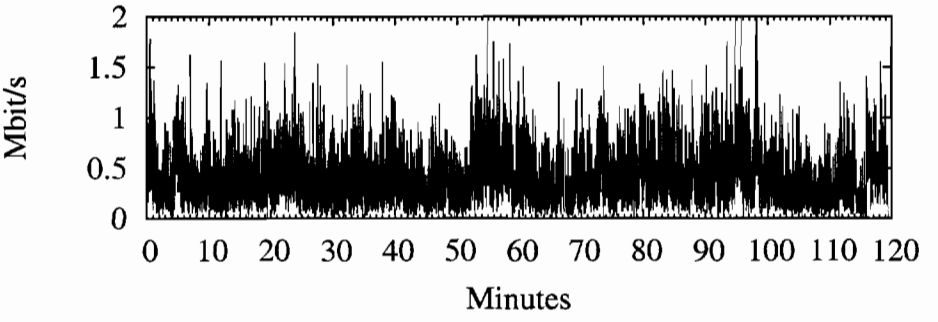


Figure 7.5: Video stream TCP Transmission Rate (smoothed to 1s intervals)

Figure 7.7 shows the performance of the streaming algorithms under these conditions. For each streaming algorithm we give one figure showing the frame-rate and another showing SNR level over the course of the whole stream[2]. Figures 7.7(a) and 7.7(b) show that the CMT algorithm has great difficulty with the conditions of our experiment. Video quality is extremely volatile, and

[2]Recall SPEG has four SNR levels.

there are several instances where the algorithm is not able to deliver even the minimum quality. Figures 7.7(c) and 7.7(d) show that the sliding window algorithm fares much better, with fewer quality changes and no failures. The number of quality changes is still quite large though. Figures 7.7(e) and 7.7(f) show that PPS with a fixed adaptation window behaves quite similarly to the sliding window approach. It would be possible to improve the consistency of PPS in the fixed window case by increasing the size of the window, but that would come at the direct expense of startup latency. The startup latency is perhaps why a sliding window approach might seem more intuitive at first.

The major benefits of PPS arise when the adaptive window scaling is enabled, shown in Figures 7.7(g) and 7.7(h), where quality gets more consistent over the course of the stream. For the majority of the movie timeline, quality changes are infrequent—several minutes apart, even though startup latency is in the range of 1 second. We notice that the quality is volatile at the start, in the first minute or so, and for several minutes at the end of the movie (approximately minutes 110–120).

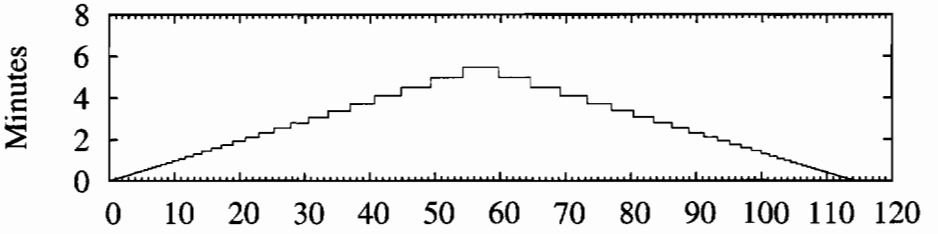


Figure 7.6: Window size (growth rate=1.1)

Figure 7.6 shows the PPS adaptation window size for the window scaling version of the experiment. We see that the anomalies of Figures 7.7(g) and 7.7(h) correspond to the period at the end of the contraction phase in Figure 7.6, where window size is at the minimum. We deliberately chose to allow this to occur in this experiment because we wanted to see PPS behaviour in boundary conditions. In real use, we manage the contraction phase of the timeline more conservatively, only allowing the window size to reach the minimum at the last possible time. Moreover, it is not necessary to use the same minimum window size in the contraction phase as the expansion phase.

Recall from Section 4.3, the purpose of the contraction phase is to ensure that PPS utilizes the network bandwidth over most of the streaming timeline. For a two-hour movie, we typically set the final window size in contraction phase to be on the order of 10s of seconds, which has only a slight impact on overall utilization, but avoids the drastic loss of consistency we observed in this experiment.

One of the keys to PPS's ability to control consistency is due to the sequential, non-overlapping treatment of windows, as was described in Section 4.2.2. Since existing algorithms use sliding windows, simply increasing their window sizes will not yield the same consistency benefit.

To summarize the unicast portion of this chapter, our experiments have shown that, under saturated network conditions, PPS adapts effectively so that it is able to achieve full utilization, and is robust in that it avoids streaming failures. Furthermore, through window scaling, PPS is able to balance between low navigation latency and consistent video quality, which sets it apart from existing streaming algorithms. These experiments have confirmed two of the main claims of our thesis statement: c) this kind of video leads to an enhanced user experience when streaming takes place over typical Internet links; and d) the video can be streamed over networks in a TCP-friendly way, making it easier to deploy in the real world.

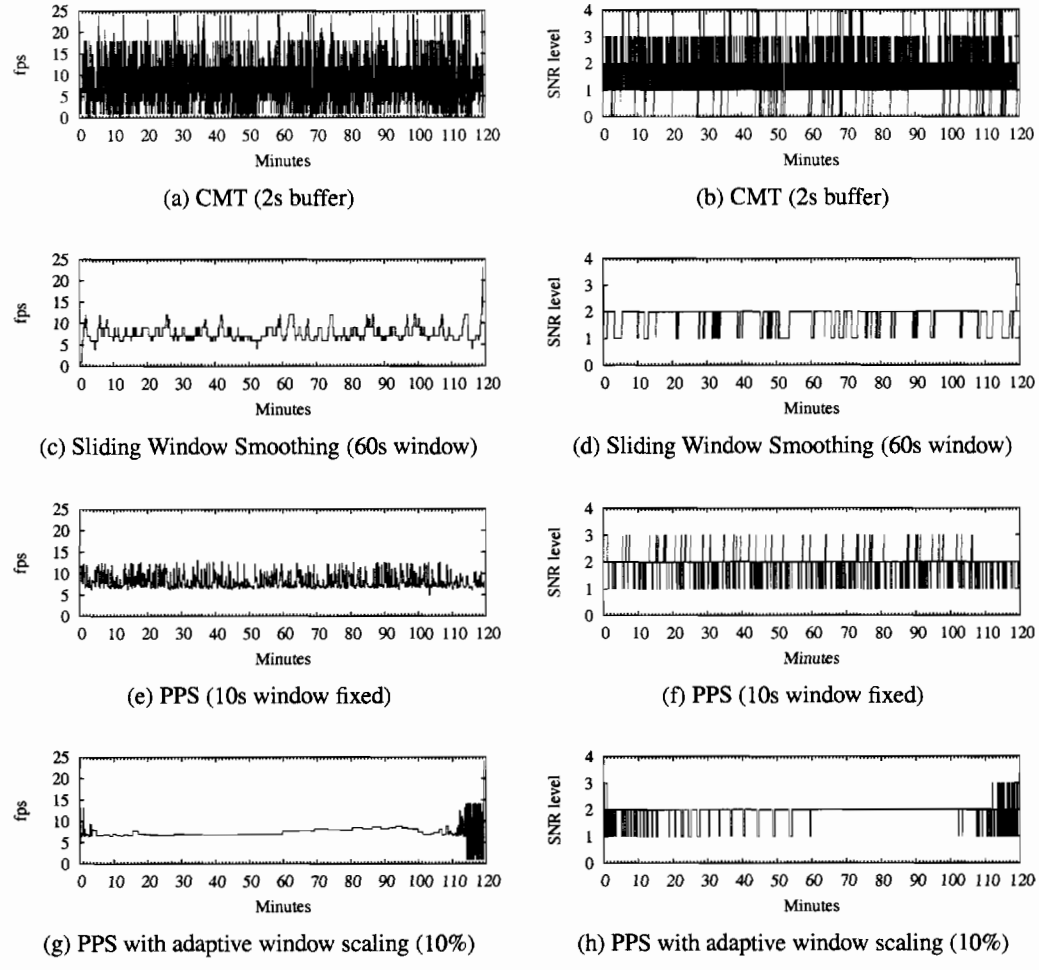


Figure 7.7: Streaming results: Sub-figures (a)-(h) show the resulting video quality with each of four streaming algorithms. The left column shows temporal quality (frame rate). The right column shows spatial quality (number of SPEG levels).

7.5 Multicast Streaming

In this section, we turn our attention to the performance of Priority-Progress Multicast (PPM) in our QStream prototype. Here our concern is the ability of PPM to accomodate a diverse set of receivers, and its ability to do so without wasting bandwidth high in the multicast tree. To address the diversity, we aim to show that PPM correctly performs multi-rate adaptation. On the efficiency side, our goal is to show that the multicast flow control in PPM correctly avoids sending unnecessary data in higher parts of the multicast tree if none of the receivers below are able to use it.

7.5.1 Multi-rate Adaptation

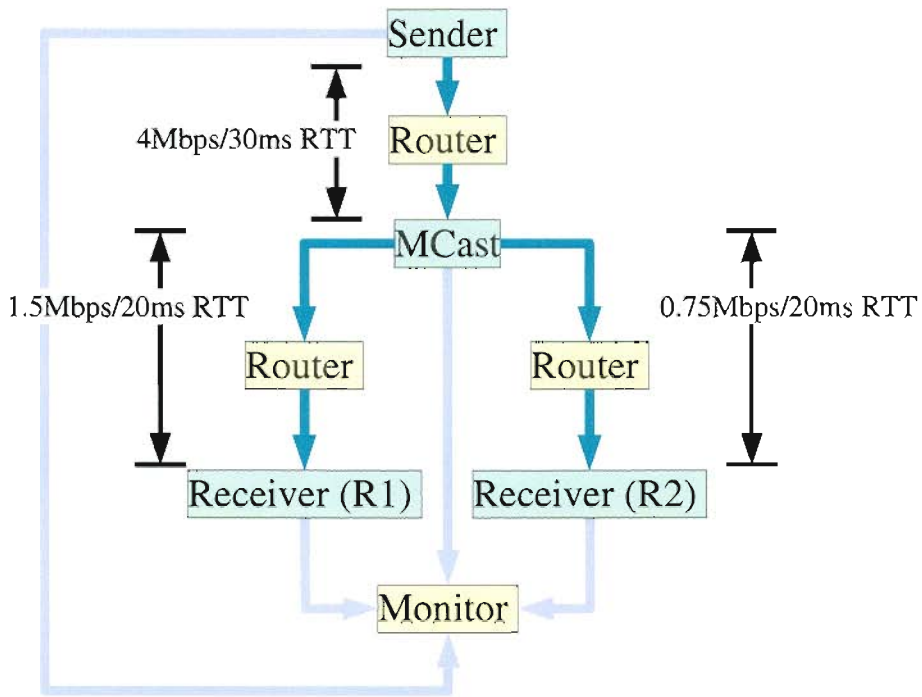


Figure 7.8: Testbed setup for basic multi-rate multicast experiment

This experiment is concerned with testing basic multi-rate adaptation, the first functional requirement of PPM—that is to match the rate of the video to the available bandwith to each receiver.

We would like to see that PPM can match the video rate to the bandwidth available to each reciever in the tree. Figure 7.8 depicts the topology we setup in our testbed to test the basic rate matching functions of PPM. The topology is the minimal multicast tree, with a single sender, a multicast node and two recievers. Between each pair of nodes, a router machine emulates wide area link characteristics with `NISTnet`. On a separate network, each of the nodes particpating in the tree sends measurements to the QStream monitor, which stores all of the data in a central location for subsequent analysis. The links of this tree are set so that neither of the links between the multicast node and the receivers have as much bandwidth as the upstream link between the sender and the multicast node. Unlike the unicast experiments above, this configuration imposes a static bandwidth limit on each link, which makes the following rate plot easier to understand than it would be if we were to allow the link rates to vary in the presence of competing traffic.

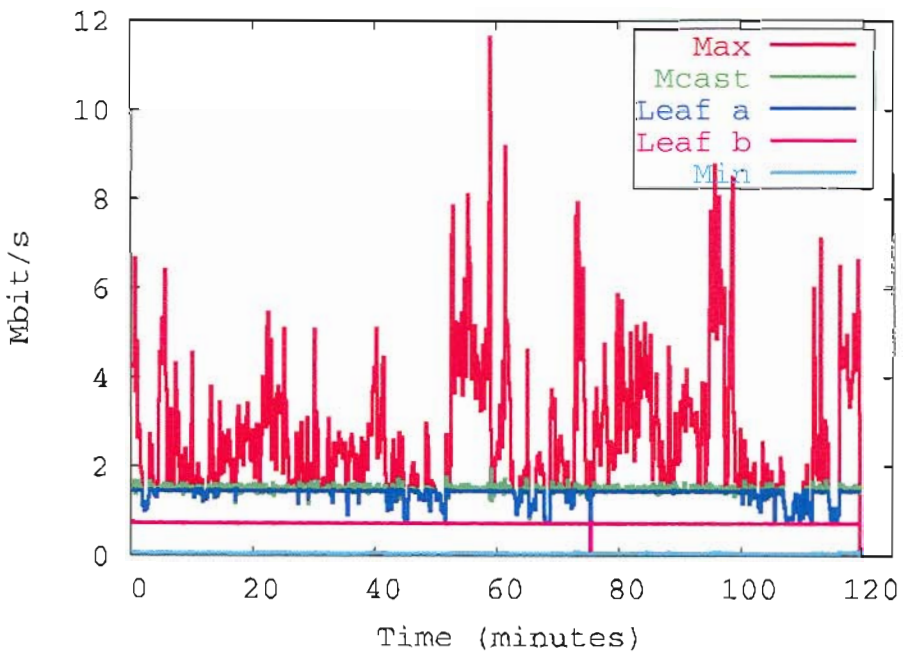


Figure 7.9: Measured link rates in multi-rate multicast with PPM

Figure 7.9 shows the measured bitrates over the course of a complete broadcast of a two hour movie. The PPS windows were fixed at a size of 10 seconds. In addition to the link rates, we also show the maximum and minimum rate requirements of the movie, noting that in some

periods, the maximum rate of the movie drops below the link capacity. The figure shows two basic results. First, PPM properly matches the rate of the video to the link capacity of each of the receivers. Second, the PPM flow control mechanism is properly conserving upstream bandwidth conservation, in that the upstream link between the sender and the mulicast node tracks the greater of its downstream links, as opposed to the maximum rate of the video.

7.5.2 Upstream Bandwidth Conservation

In this experiment, we expand the multicast topology to stress the PPM flow control mechanism. Recall from Section 5.1.2 that the PPM design includes flow control at the application protocol level, as opposed to just using the flow control functionality already provided by TCP. The purpose of this design decision is to conserve upstream bandwidth by reducing the number of places in the multicast tree where data will accumulate in buffers. If PPM just used TCP flow control, the receive buffer upstream of each stepdown point (a multicast forwarder that has more upstream bandwidth available than all of its children) would fill completely before the TCP flow control mechanism would cause the corresponding upstream node to stop sending. As a result, a complete path through the tree from the source to a reciever may include several reciever buffers worth of accumulated data. This would have a negative impact on the total source to receiver transmission latency and on the responsiveness of the flow control. PPM's flow control sends an explicit application level message upstream to stop sending, so that the multicast node can keep the receive buffer for its upstream TCP session drained. Figure 7.10 shows the topology we used to stress the PPM flow control mechansim. Here we use a deeper tree, where each level of the tree on the path from the sender to the receiver is progressively more bandwidth limited.

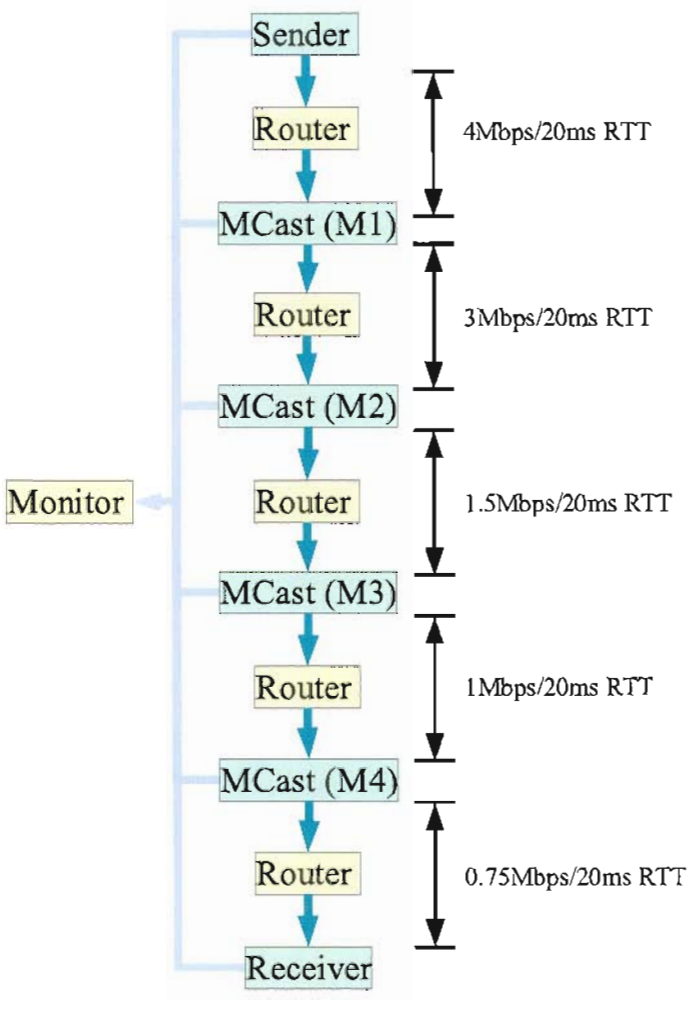


Figure 7.10: Stressing flow control with a deep multicast tree

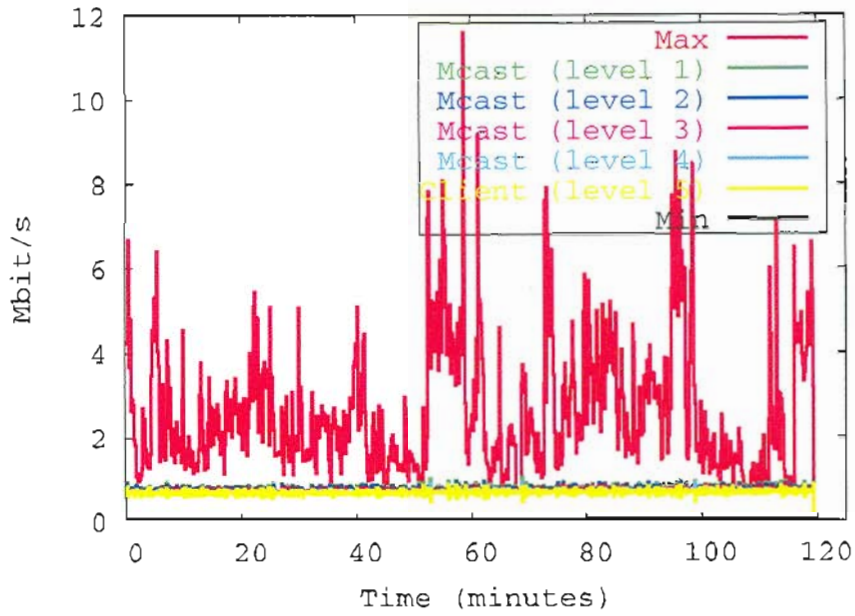


Figure 7.11: Measured link rates in a deep PPM tree

In Figure 7.11, we show the rates for each link of the deep tree of Figure 7.10, again using a two hour movie. Here we see that PPM flow control functions as we'd hope, in that the rates of all of the links are properly limited to match the most constrained link at the bottom of the tree.

7.5.3 Bandwidth Conservation with Progressive Bottlnecks

While the previous experiment showed that PPM properly constrained link usage in a deep tree, we now add additional receivers to the tree to ensure that PPM doesn't allow downstream limits to overconstrain upstream links. In Figure 7.12, we show the modified topology which adds a receiver at each level of the tree. The idea of this topology is that the receiver at each level should cause the multicast node to fully utilize its upstream link.

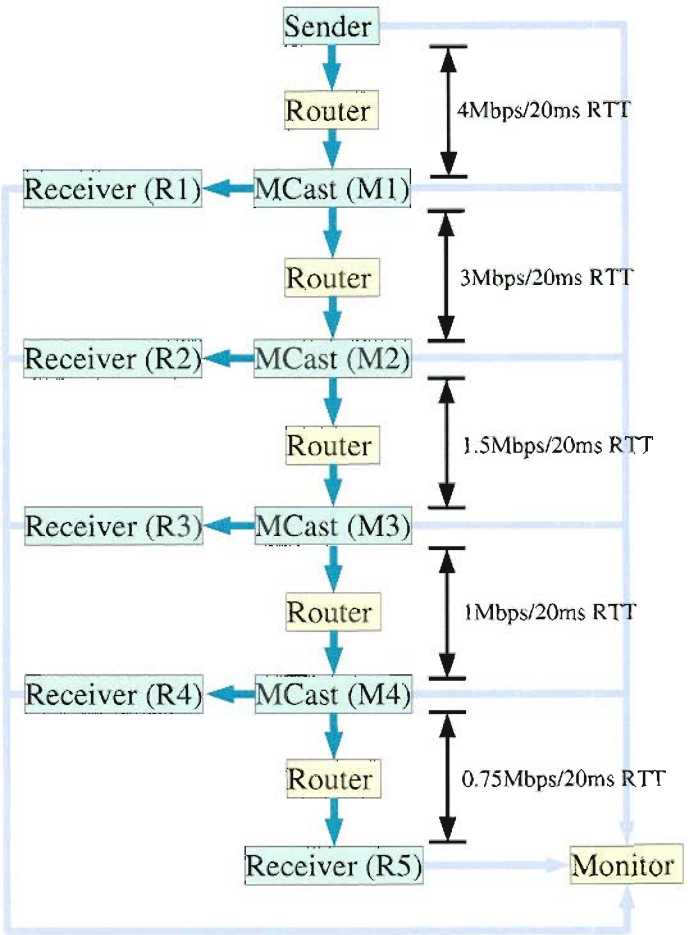


Figure 7.12: Stressing flow control with a deep and wide multicast tree

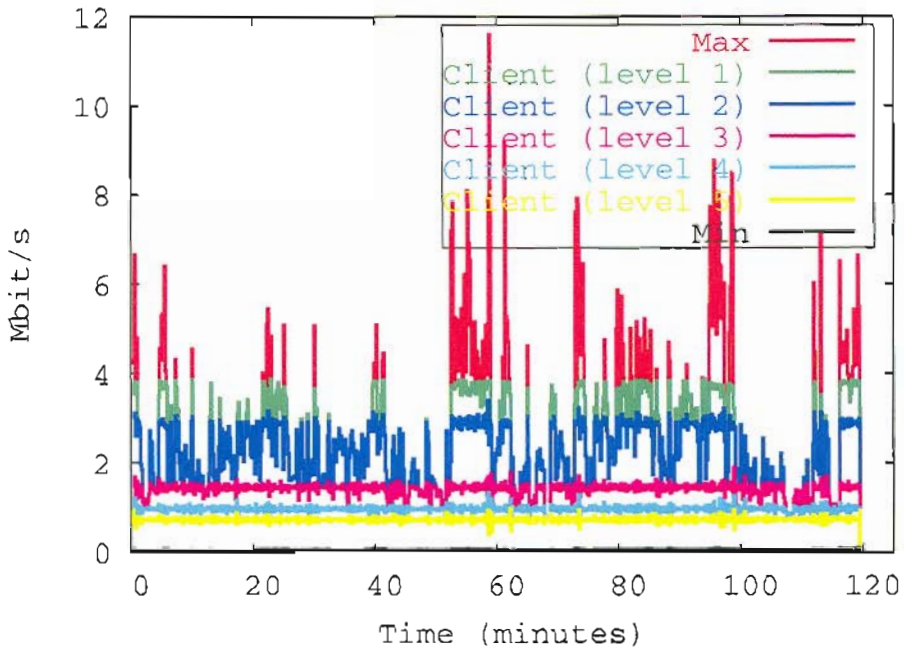


Figure 7.13: Measured link rates in a deep and wide PPM tree

The resulting data rates are shown in Figure 7.13, which confirm that PPM is able to tightly match the rate to each receiver closely to the available capacity between itself and the root of the tree.

To summarize, we have shown that PPM performs multi-rate multicast using an overlay tree where the individual links are PPS multicast sessions. By virtue of the fact that each link is a separate TCP friendly session, PPM is TCP friendly by definition. Our experiments showed that PPM successfully preserves the basic strengths of PPS in the multicast setting meaning that, through adapatation, each reciever is able to utilize the full bandwidth between it an the root of the tree. Furthermore, the flow control mechanism in PPM ensures that upstream bandwidth utilization is limited only to what is required below. These results confirm the final claim of our thesis statement: e) TCP-video streaming can be applied efficiently to multicast delivery, enabling large scale video broadcast distribution.

Chapter 8

Conclusions and Future Work

In the first section of this chapter, we summarize the contributions of this dissertation, recapping our motivating arguments, the conceptual contributions we made, the system components we implemented, and our evaluation methodology and the results we obtained. After that, the second section describes some of the open problems that we leave for future work.

8.1 Conclusions

This dissertation has presented a framework for adaptive media streaming toward an overall goal of an *encode-once, stream anywhere* level of flexibility and simplicity.

8.1.1 Motivating arguments

In this dissertation, we argued that an adaptive approach to streaming is necessary due to bitrate variations with compressed video and to the volatile dynamics of best-effort networks. We defined the goals of adaptive streaming as effectiveness, efficiency, and scalability. Our notion of effectiveness consisted of a number of sub-goals: robustness, utilization, latency and consistency.

To achieve the basic goals of robustness and utilization, we argued for the necessity of a scalable video representation, which should have the goals of supporting a wide range of quality levels and bitrates, with fine granularity. We recognized that in light of the multi-dimensional nature of video quality, adaptive video streaming should allow control over the mix of video adaptations. These factors, and additional concerns for scalability, lead us to select the priority-drop approach for our investigation. To stream such video over the network, we made the case that an adaptive protocol is needed to match the rate of the video to available bandwidth (so that

robustness and utilization can be maximized), and that the protocol should also balance the goals of latency and consistency. Finally, to support continued scalability of the Internet, we argued that adaptive video streaming needs to be TCP friendly, and should support multicast distribution.

8.1.2 Conceptual contributions

Our conceptual contributions began with our treatment of video in Chapter 3. To serve as a demonstration of scalable video, we described *SPEG*, which is a simple extension to MPEG that adds spatial scalability. We used SPEG as a basis to exercise the priority-drop approach throughout the rest of our framework. To show how we could offer control over the mix of adaptations, we described our approach to quality specification based on *utility functions*. We used utility functions to capture preferences that dictate the best mix of quality adaptations across the range of acceptable quality-resource tradeoffs. We showed how a *Mapper* could be used to effectively translate these preferences into priority assignments on the basic application data units (ADUs) of video data, the result being a set of timestamped and prioritized streaming data units (SDUs).

For network transport, we described the *Priority-Progress Streaming* (PPS) algorithm (in Chapter 4), which adaptively matches the bitrate of a stream of SDUs to the network rate, as detected by a TCP-friendly congestion-control mechanism. We described how our PPS approach balances the simultaneous requirements of maintaining the real-time progress of the stream while adapting to the best-effort service of the network, thereby addressing the goals of robustness and utilization. We showed how PPS can do this by subdividing the timeline of the video into intervals called adaptation windows, and then re-ordering the data (SDUs) within those intervals so that transmission proceeds from high to low priority. When moving from one adaptation window to the next, PPS drops unsent (low-priority) data at the server without transmitting it to the network. In this way the rate of the PPS stream naturally matches the available bandwidth. We described the role of adaptation window size in determining the latency and consistency performance of PPS. Noting the conflict between these two objectives, we introduced a window scaling mechanism that allowed the size of the adaptation windows to change as streaming proceeds. We showed how this mechanism adjusts the compromise between latency and consistency over the course of a long stream, thereby allowing PPS to provide low navigation latency, and to deliver progressively better

consistency (and robustness) as streaming proceeds uninterrupted (by navigation actions). To address network delay, we showed how to manage a phase offset between the server and receiver-side streaming clocks. The overall contribution of PPS is to show how to achieve effective streaming at the micro-level (single user).

To address scalable distribution, we described how to extend PPS to a multicast overlay, called *Priority-Progress Multicast (PPM)* in Chapter 5. PPM defines an adaptive data forwarding discipline for multicast by composing a multicast tree as an overlay of unicast PPS sessions. Each edge of a PPM multicast tree is a self-contained PPS unicast session. We described how PPM enables priority data drop at each interior node of the multicast tree, so that the whole tree performs multi-rate adaptation, meaning that the video rate to each receiver is matched to the available bandwidth on the path between the receiver and the root of the tree. We showed that the presence of slower receivers in the tree does not penalize faster receivers. We also described the multicast flow control mechanism in PPM, which ensures that upstream bandwidth is conserved if none of the receivers below in the tree are able to utilize it. With PPM, we showed how the Priority-Progress approach can address macro-level (network wide) concerns of scalability.

8.1.3 Implementation

We have built a complete software system for video streaming based on the framework presented in this dissertation. Our streaming system is called QStream (Quasar Streaming). The QStream prototype is a fully operational end-to-end video streaming system. QStream includes a server (StreamServ), a player (StreamPlay), a multicast proxy (MCastProxy), and a number of support programs and libraries. All of the major components described in this dissertation are fully implemented in QStream: the SPEG video format, the Mapper algorithm, the PPS protocol, and the PPM protocol. The SPEG format and Mapper algorithm were described in Chapter 3. Chapters 4 and 5 gave the conceptual descriptions of the PPS and PPM protocols. More details of their algorithms were covered in Chapter 6. QStream also includes a number of significant features that were not described in this dissertation, such as audio support (multiplexed transport and synchronized playback), and live streaming from a webcam source. QStream was used extensively in the evaluation part of this dissertation. We have used it to generate data for simulations, we have used it for controlled tests in our lab testbed, and we have used it in live tests and demonstrations

over the real Internet to home broadband (e.g., cable modems) and over 802.11b networks. In general, the system is stable enough and free of serious bugs, so that we use it frequently in live demonstrations and research talks to show the Priority Progress approach in action.

In addition to the core streaming software, we developed a substantial amount of support software in QStream, including the the `mxtraf` traffic generator, the QStream remote network monitor (Monitor), and a pair of libraries called `GAIO` (Asynchronous IO library) and `qsf` (Quasar Streaming Framework library). The `mxtraf` program is a scalable and efficient network traffic generator, used to inject realistic mixes of competing traffic into a network testbed. The QStream Monitor program collects a wealth of diagnostic data from the other QStream programs, and presents them for real-time visualization through a set of graphical views with a software oscilloscope, `gscope`. We also used the Monitor to store the data to a database for offline analysis and subsequent visualization through programs such as gnuplot[1].

We described the `GAIO` and `qsf` libraries in Chapter 6. They provide the infrastructure for the reactive programming model we adopted, and provide support for the network protocols in the QStream programs (PPS, PPM, `mxtraf` client-server protocol, and Monitor data collection protocol). We have publicly released all of the QStream software under open source terms (with a GPL license) to promote re-use of our framework by other researchers. The QStream software was used extensively in our evaluation of Priority Progress, which we summarize next.

8.1.4 Evaluation

The Priority-Progress framework was evaluated experimentally through measurements of QStream performance, including live and emulated network settings. We also used measurements from QStream to drive simulations that compared the performance of PPS to previous streaming algorithms. By using an emulated network in our lab testbed, we were able to verify the performance of PPS in demanding conditions where the network was saturated with competing traffic. Also, through carefully constructed multicast trees, we were able to use our testbed to verify PPM's ability to perform effective and efficient multi-rate multicast. The rest of this section will summarize our major results.

[1]This is how we generated most of the plots in this thesis

In Chapter 3, using SPEG to encode a sequence of real movies, we showed how SPEG's scalability can span a wide range of bitrates, as much as two orders of magnitude between the minimum and maximum, and the supported rates were spread well within this range. This first set of results demonstrated the first claim of our thesis statement: a) through informed dropping, it is possible to cover a very wide range of quality-rate combinations and with fine granularity. We also showed, using our implementation of the Mapper, that the translations from policy specifications to priority assignments are correct, i.e. the priorities assigned by the Mapper cause priority dropping to produce a mix of adaptations that accurately match the target policy. Hence we have verified the second claim of our thesis statement: b) tailorable adaptation policies can be used to control the mixture of adaptations to best meet content, task, and user specific requirements.

We presented the measured performance of the PPS protocol in Chapter 7. We tested PPS performance along a network path that emulated wide-area conditions (delay, bandwidth, etc.) and was saturated with competing traffic that we generated with `mxtraf`. In these conditions, we compared the performance of PPS over the course of a full length (two hour) movie to existing streaming algorithms, and showed that it was more effective and robust. In particular, we showed that, with the window scaling feature of PPS enabled, PPS delivered significantly better consistency than previous approaches. Hence we verified the next two claims of the thesis statement: c) this kind of video leads to an enhanced user experience when streaming over typical network lines, and d) video can be streamed over networks in a TCP-friendly way making it easier to deploy in the real world.

Our next set of experiments, in Chapter 7, measured the performance of multicast PPM in our testbed, using a sequence of carefully selected multicast topologies. First, using a simple tree, we showed that PPM correctly matches the rates of individual receivers to the available bandwidth. Then, using a larger tree, we showed how PPM was able to use only the bandwidth necessary to service a given set of receivers. In particular, we verified that the flow control mechanism in PPM avoids the waste of upstream bandwidth. Our final PPM experiment used a tree that tested both aspects together, showing that rate matching and bandwidth conservation functions did not interfere with each other. The overall performance of PPM verified the final claim of our thesis statement: e) TCP-friendly video streaming can be applied efficiently to multicast delivery, enabling large scale video broadcast distribution.

8.1.5 Summary of Conclusions

Our framework is broad in its scope in that it treats areas of video representation, quality of service specification, network protocols, and even the real-time programming model. However, we had a common theme joining these areas, that is, adaption through priority data dropping. Priority-data drop is the foundation of our vision of an encode once, stream-anywhere framework.

To summarize, the main contributions of this dissertation were the following:

- SPEG and the Priority Mapper: We showed how through proper framing and prioritization, a single video encoding can support a wide range of bitrates with fine granularity. Moreover, the mix of adaptations within the range is explicitly controllable so that user, content, task and device specific requirements can be optimally addressed.

- Priority-Progress Streaming (PPS): our adaptive streaming protocol achieves several important objectives, namely robustness, high utilization. Furthermore, the window-scaling feature of PPS provides a powerful mechanism to mitigate the conflict between consistent quality over time (characteristic of downloads) and low navigation latency (characteristic of streaming).

- Priority-Progress Multicast (PPM): We extended PPS to multicast distribution through an overlay approach. PPM supports multi-rate, quality-adaptive, multicast distribution, in a completely TCP friendly manner. To our knowledge, prior approaches have only been able to a subset of these characteristics simultaneously.

- QStream prototype: We have implemented a complete implementation of our system that was used extensively for our experimental evaluation. The prototype itself also constitutes interesting contributions toward programming for time-sensitive network applications, such as the use of reactive programming, support for scalable generation of competing traffic, and the use of remote, real-time, visualization techniques. We have made the whole framework publicly available.

Although we made contributions across the various sub-areas, the overall combination of the components into the framework, including their full implementation, is perhaps the major contribution. More than any of the parts, we believe it is the whole system that best demonstrates the

elegance of priority data drop as an adaptation strategy.

8.2 Future Work

In this section, we describe some of the future research problems that have emerged from the work in this dissertation. Where appropriate, we sketch out the potential solutions we have devised.

8.2.1 Better quality-calibration in scalable compression

The emergence of scalable compression was one of the main factors that inspired our investigation into priority-drop based adaptation. The SPEG format provides a sufficient demonstration of the essential advantages of a scalable approach: the ability to adapt quality with fine granularity over a wide range. The combination of scalable compression with priority-drop was quite potent, particularly in our framework where it retains the ability to control the mix of adaptations. We developed the SPEG part of this work mainly out of necessity, since we could not find publicly available scalable codecs. Our main research objectives were on the delivery side of streaming, with QoS specification and adaptive streaming. We were specifically not concerned with improving the compression efficiency of scalable coding, which is a research area in which others are making steady progress. However, our experience with QStream shows that SPEG, and scalable compression in general, needs more radical restructuring to better support our overall goal of "encode once, stream anywhere". For instance, scalable video compression should be more explicitly integrated with work from the emerging field of video quality metrics [70]. This area can be partitioned into two significant sub-areas: objective and subjective quality measurement.

Objective quality measures are those that can be automated (i.e., given some video data possibly with a reference video, the metric can calculate a measure of the video quality). In QStream, we simply use the number of SPEG layers as our measure of spatial quality, although this value is only indirectly related to the actual spatial quality. The most ubiquitous quality measure in the field of video compression is the pseudo signal-to-noise ratio (PSNR), which is based on the mean square error between each reconstructed image and the original. Within the video community, it is widely acknowledged that PSNR is a rather crude quality measure. At least conceptually, it would be preferable to have a measure based on a model of the human visual system (HVS). On the

other hand, PSNR is well understood and easy to implement, while the HVS based metrics remain an open research area. We think that our framework makes it obvious that scalable compression ought to calibrate various quality layers against one of these objective quality measures (perhaps PSNR initially, and an HVS measure when they become more practical), and that the quality values should be exposed as part of the compressed representation. We think the construction of a new scalable video encoder in this way would have the dual advantage of better compression efficiency, and in our framework it would have the advantage that it would facilitate more accurate priority assignments by our mapper.

Subjective quality measurements use human subject studies to better understand the psychology of human visual perception, which can help provide insight into the biological side of visual perception, as well as provide a means to verify the effectiveness of objective quality measures. In terms of our approach, there are important questions about the utility of different aspects of video quality, as well as the impact of changes in quality to users. Zink *et al.* have built a subjective quality evaluation system, which uses our SPEG implementation, and they have conducted a study to measure human assessments of various patterns of changing quality [97]. This type of study is important in relation to this dissertation, as the information revealed relates directly to a question we have left unanswered: "On what basis should utility functions be set?"

8.2.2 Quality adaptation for other resource types

Our investigation of the priority-drop approach in this dissertation was limited to the context of adapting to a single resource type, wherein our goals concerned how to match video bitrate with the available bandwidth in the network. As it does for the network, the commodity infrastructure typically provides a best effort service model for the other basic resources, such as CPU time and storage (disk) bandwidth. The PPS approach can be extended to these resources also. The basic approach of PPS, which subdivides the processing timeline into adaptation windows, and then processes the contents of the windows in priority order, forms a design pattern that is applicable to the other resources. For instance, video decoding could work in this fashion so that it adapts to available CPU, although it would require minor modifications to legacy software, such as the `ffmpeg` codec [3] from which we derived the SPEG codec in QStream. The `ffmpeg` API is typical of software based video decoders, in that it assumes that frames are decoded in MPEG's

natural order. To apply the PPS pattern, we would decode video in priority order. The context (state) management in the codec would need modifications to accommodate priority order decoding. We expect with these minor modifications, we could extend our player StreamPlay to perform Priority-Progress based CPU adaptation.

8.2.3 Quality adaptation for other application domains

In this dissertation, we have focused on video streaming applications, as they embody many of the critical challenges relating to supporting time-sensitive applications in the Internet. However, the general approach and parts of our existing framework are directly applicable to other data types and applications. For example, the Priority-Progress model should make sense for distributed graphics applications such as networked games and scientific visualizations, especially as the size of the full graphical environments become large and detailed enough to overwhelm the available bandwidth of some receivers. The transport of sensor data in sensor networks is another good match with our approach, as consumers of sensor data generally desire fresh data with as much fidelity as possible. Sensor networks usually depend heavily or entirely upon wireless links, so bandwidth variabilites are extreme. Also, control over bandwidth usage is critical for power management in these types of networks, as power availability is yet another source of resource variation (beyond available network bandwidth and video bitrate requirements). Variable power availability will be especially significant for networks that derive some or all of their power from unpredictable sources such as wind and solar energy.

8.2.4 Alternatives to TCP

Another important question left unanswered in this thesis is how Priority-Progress might perform on the many alternative protocols (and congestion control schemes) targeted specifically at media streaming such as HPF [54], TFRC [27], RAP [67], TEAR[68], or SCTP [64]. Such a comparison is not trivial however, as these protocols are unreliable, and so using them with PPS will mean that the effects of random data loss must be dealt with in some way at the application level. In our estimation this could represent a significant amount of effort. SCTP might be the most promising choice in this regard, because it offers a partial reliability model (bounded retransmission time) that might be the most straightforward to accommodate in PPS.

8.2.5 Improving TCP's support for streaming applications

We have used TCP as our transport for streaming, which is notoriously difficult. The high effectiveness of PPS over stock TCP might come as quite a surprise to many. That said, there is certainly room for improvement.

8.2.5.1 AQM and ECN

Although not discussed at detail in this dissertation, we expect that Active Queuing Mechanisms (AQM) in routers in combination with Early Congestion Notification (ECN) could result in significant benefits to TCP based streaming applications. In brief, one of the principle goals of an AQM+ECN combination is to eliminate packet dropping due to congestion through the use of explicit congestion notifications. The AQM mechanism's job is to anticipate congestion (at routers) and to select the packets that belong to the best candidate for congestion avoidance. ECN implements the actual mechanism for routers to signal these conditions to TCP endpoints. Although there are serious deployment issues with AQM and ECN (as with any mechanism that targets the core of the Internet), in our network testbed, we have observed significant improvements to TCP's latency and sharing performance when AQM (NISTnet's DRED mechanism) and ECN were enabled. For future work, comprehensive measurement studies that investigate and quantify the potential benefits of AQM+ECN to multimedia would help to make the case for AQM deployment, and might also shed further light on the strengths and weaknesses of TCP relative to other transport protocols we mentioned in Section 8.2.4.

8.2.5.2 TCP segment-size tuning

In a previous study [31], we described our `TCP_MINBUF` socket option, which makes great improvements to latency properties of TCP in practice, notably without changing the "on the wire" part of the TCP protocol. One of the areas we looked at in this dissertation was the performance of PPS in the presence of competing network traffic. Because PPS is layered above TCP, the basic sharing properties of TCP are of interest to us. TCP is known to converge to a fair sharing when flows compete for the same link. In [57], Mathis *et al.* suggest the rate behaviour for each TCP flow can be modelled by the following equation: $BW = \frac{MSS}{RTT}\frac{C}{\sqrt{p}}$ where BW is the bandwidth,

MSS is the maximum segment size, RTT is round-trip time, C is a constant ($\sqrt{\frac{3}{2}}$) derived from TCP's congestion control, and p is the packet drop probability. Holding values on the right side of the equation constant (for a group of flows sharing a common bottleneck), the equation confirms the rule of thumb that n TCP flows will converge on a $1/n$ bandwidth share. However, we have noticed that this behaviour breaks down when the number of flows becomes large relative to the total capacity of the bottleneck. The reason for the breakdown stems from the relationship between packet size and RTT in TCP's congestion control. In view of the Matthis equation, we notice that, under high multiplexing, the bandwidth share for each flow, in terms of number of packets per round trip, drops below the minimum required to maintain AIMD. The range in which this effect holds increases with larger values of RTT and packet size. For instance, with RTT of 40ms and flows using path MTU discovery (PMTU), the minimum per flow rate required to sustain TCP's fair sharing is in the range of 500kbps[2]. This constrains the low end of the range of effectiveness of TCP for video streaming, and is a major concern for audio-only streaming using TCP. We plan to investigate modifications to the OS protocol stack that would do *packet size tuning*, to adapt packet sizes at low rates in order to promote better sharing (and consistency of throughput). We think this technique could lead to important performance gains not only to media streaming, but for any application that does bulk transfer over TCP.

[2]PMTU typically leads to a TCP MSS of about 1440 bytes. We're assuming an minimum average of 2 packets per round trip are required to maintain AIMD.

Bibliography

[1] AMIR, E., MCCANNE, S., AND ZHANG, H. An application level video gateway. In *Proceedings of the ACM Multimedia Conference* (1995), pp. 255–265.

[2] BANERJEE, S., BHATTACHARJEE, B., AND KOMMAREDDY, C. Scalable application layer multicast. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2002), pp. 205–217.

[3] BELLARD, F., NIEDERMAYER, M., ET AL. The FFMpeg Project. `http://ffmpeg.sf.net/`. *Date viewed: January 2003.*

[4] BERRY, G., AND GONTHIER, G. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming 19*, 2 (1992), pp. 87–152.

[5] BIRNEY, B. Intelligent Streaming. `http://msdn.microsoft.com/`, *Date viewed: October* 2000.

[6] BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. An Architecture for Differentiated Services. RFC 2475, December 1998.

[7] BYERS, J., FRUMIN, M., HORN, G., MICHAEL LUBY, M. M., ROETTER, A., AND SHAVER, W. FLID-DL: Congestion Control for Layered Multicast. In *Proceedings of the Second Int'l Workshop on Networked Group Communication (NGC 2000)* (Stanford, CA, November 2000), pp. 71–81.

[8] CASNER, S., AND DEERING, S. First IETF Internet Audiocast. *ACM Computer Communication Review 22*, 3 (July 1992), pp. 92–97.

[9] CASTRO, M., DRUSCHEL, P., KERMARREC, A., NANDI, A., ROWSTRON, A., AND SINGH, A. Splitstream: High-bandwidth content distribution in a cooperative environment. In *Proceedings of (IPTPS'03) (February 2003).* (2003), pp. 98–106.

[10] CEN, S., PU, C., STAEHLI, R., COWAN, C., AND WALPOLE, J. A Distributed Real-Time MPEG Video Audio Player. In *Network and Operating System Support for Digital Audio and Video* (1995), pp. 142–153.

[11] CHU, Y.-H., RAO, S. G., AND ZHANG, H. A Case for End System Multicast. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (2000), pp. 1–12.

[12] CLARK, D. D. The design philosophy of the DARPA internet protocols. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (Stanford, CA, Aug. 1988), ACM, pp. 106–114.

[13] CONKLIN, G., GREENBAUM, G., LILLEVOLD, K., AND LIPPMAN, A. Video Coding for Streaming Media Delivery on the Internet. *IEEE Transactions on Circuits and Systems for Video Technology 11*, 3 (March 2001), pp. 269–281.

[14] CROVELLA, M. E., AND BESTAVROS, A. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking 5*, 6 (1997), pp. 835–846.

[15] CROWCROFT, J., HAND, S., MORTIER, R., ROSCOE, T., AND WARFIELD, A. QoS's Downfall: At the bottom, or not at all! In *Proceedings of the Workshop on Revisiting IP QoS: Why do we care, what have we learned? (RIPQOS)* (Karlsruhe, Germany, August 2003), pp. 88–97.

[16] DAI, M., AND LOGUINOV, D. Analysis of Rate-Distortion Functions and Congestion Control in Scalable Internet Video Streaming. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (June 2003), pp. 60–71.

[17] DE CUETOS, P., GUILLOTEL, P., ROSS, K., AND THOREAU, D. Implementation of adaptive streaming of stored MPEG-4 FGS video over TCP. In *Proceedings of the International Conference on Multimedia and Expo (ICME)* (Lausanne, Switzerland, August 2002), pp. 156–168.

[18] DEERING, S. *Multicast Routing in a Datagram Internetwork.* PhD thesis, Stanford University, 1991.

[19] DIOT, C., LEVINE, B. N., LYLES, B., KASSEM, H., AND BALENSIEFEN, D. Deployment issues for the IP multicast service and architecture. *IEEE Network 14*, 1 (/ 2000), pp. 78–88.

[20] ERIKSSON, H. The multicast backbone. *Communications of the ACM 8* (1994), pp. 54–60.

[21] FALL, K. Network emulation in the VINT/NS simulator. *Proceedings of the fourth IEEE Symposium on Computers and Communications* (1999), pp. 244–255.

[22] FEAMSTER, N., BANSAL, D., AND BALAKRISHNAN, H. On the Interactions Between Layered Quality Adaptation and Congestion Control for Streaming Video. In *Proceedings of the International Packet Video Workshop* (Kyongiu, Korea, April 2001), pp. 128–139.

[23] FENG, W., CHOI, J., CHANG FENG, W., AND WALPOLE, J. Under the Plastic: A Quantitative Look at DVD Video Encoding and Its Impact on Video Modeling. Tech. Rep. CSE-03-004, OGI, Feburary 27 2003.

[24] FENG, W., LIU, M., KRISHNASWAMI, B., AND PRABHUDEV, A. A Priority-Based Technique for the Best-Effort Delivery of Stored Video. In *Proceedings of the SPIE Multimedia Computing and Networking Conference* (San Jose, California, January 1999), pp. 129–139.

[25] FENG, W., AND REXFORD, J. A Comparison of Bandwidth Smoothing Techniques for the Transmission of Prerecorded Compressed Video. In *Proceedings of the IEEE Infocom* (1997), pp. 58–66.

[26] FENG, W., AND SECHREST, S. Critical Bandwidth Allocation for the Delivery of Compressed Video. *Computer Communications (Special Issue on Systems Support for Multimedia Computing) 18*, 10 (October 1995), pp. 709–717.

[27] FLOYD, S., HANDLEY, M., PADHYE, J., AND WIDMER, J. Equation-based congestion control for unicast applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2000), pp. 43–56.

[28] FLOYD, S., JACOBSON, V., LIU, C.-G., MCCANNE, S., AND ZHANG, L. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking 5*, 6 (1997), pp. 784–803.

[29] GARRETT, M. W., AND WILLINGER, W. Analysis, modeling and generation of self-similar VBR video traffic. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (1994), pp. 269–280.

[30] GOEL, A., ABENI, L., KRASIC, C., SNOW, J., AND WALPOLE, J. Supporting time-sensitive applications on a commodity OS. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (Dec. 2002), pp. 165–180.

[31] GOEL, A., KRASIC, C., LI, K., AND WALPOLE, J. Supporting Low Latency TCP-Based Media Streams. In *Proceedings of the International Workshop on Quality of Service (IWQoS)* (May 2002), pp. 156–164.

[32] GOEL, A., AND WALPOLE, J. Gscope: A visualization tool for time-sensitive software. In *Proceedings of the Freenix Track of USENIX Technical Conference* (June 2002), pp. 133–142.

[33] GROUP, S. M. W. Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. Tech. rep., World Wide Web Consortium, 1998. `http://www.w3.org/TR/REC-smil`. *Date Viewed: June 2000.*

[34] HANDLEY, M., SCHULZRINNE, H., SCHOOLER, E., AND ROSENBERG, J. SIP: Session Initiation Protocol. RFC 2543, March 1999.

[35] HASKELL, B. G., PURI, A., AND NETRAVALI, A. N. *Digital Video: An Introduction to MPEG-2.* Chapman & Hall, 1997, ch. 9.

[36] HE, D., MULLER, G., AND LAWALL, J. L. Distributing mpeg movies over the internet using programmable networks. In *Proceedings of the International Conference on Distributed Computing Systems* (July 2002), pp. 161–170.

[37] HE, Y., WU, F., LI, S., ZHONG, Y., AND YANG, S. H.26l-based fine granularity scalable video coding. In *ISCAS 2002* (Phoenix, USA, May 2002), pp. 548–551.

[38] HUANG, J., KRASIC, C., WALPOLE, J., AND FENG, W. Adaptive Live Video Streaming by Priority Drop. In *Proceedings of the IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)* (Miami, July 2003), pp. 342–354.

[39] IANNACCONE, G., MAY, M., AND DIOT, C. Aggregate Traffic Performance with Active Queue Management and Drop from Tail. *Computer Communication Review 31*, 3 (July 2001).

[40] ISO/IEC. 13818-2 Information technology — Generic coding of moving pictures and associated audio information: Video . International Standard, 1993.

[41] ISO/IEC. 11172-2 Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 2: Video. International Standard, 1994.

[42] ISO/IEC. 14496-2 Information technology — Coding of audio-visual objects — Part 2: Visual. International Standard, December 1999. First edition.

[43] ISO/IEC. 61834 Helical-scan digital video cassette recording system using 6,35 mm magnetic tape for consumer use (525-60, 625-50, 1125-60 and 1250-50 systems). International Standard, 1999.

[44] JACOBS, S., AND ELEFTHERIADIS, A. Streaming Video using Dynamic Rate Shaping and TCP Flow Control. *Journal of Visual Communication and Image Representatio 9*, 3 (January 1998), pp. 211–222. (invited paper).

[45] JANNOTTI, J., GIFFORD, D., JOHNSON, K., KAASHOEK, M., AND O'TOOLE, J. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (San Diego, CA, October 2000), pp. 197–212.

[46] KALMANEK, C. A Retrospective View of ATM. *ACM Computer Communication Review 32*, 5 (October/November 2002), pp. 13–18.

[47] KANG, S. H., AND ZAKHOR, A. Packet Scheduling Algorithm for Wireless Video Streaming. In *Proceedings of the International Packet Video Workshop* (Pittsburgh, April 2002), pp. 121–133.

[48] KAR, K., SARKAR, S., AND TASSIULAS, L. A scalable low-overhead rate control algorithm for multirate multicast sessions. *IEEE Journal on Selected Areas in Communications 20*, 8 (October 2002), pp. 127–146.

[49] KELLER, R., CHOI, S., DECASPER, D., DASEN, M., FANKHAUSER, G., AND PLATTNER, B. An active router architecture for multicast video distribution. In *Proceedings of the IEEE Infocom* (2000), pp. 1137–1146.

[50] KIM, J.-W., KIM, Y.-G., H.-J. SONG, T.-Y. K., CHUNG, Y.-J., AND KUO, C.-C. J. TCP-friendly Internet Video Streaming employing Variable Frame-rate Encoding and Interpolation. *IEEE Transaction on CSVT 10*, 7 (October 2000), pp. 1164–1177.

[51] KOSTIC, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: high bandwidth data dissemination using an overlay mesh. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (Bolton Landing, NY, USA, 2003), pp. 282–297.

[52] KRASIC, C., AND WALPOLE, J. QoS scalability for streamed media delivery. CSE Technical Report CSE-99-011, Oregon Graduate Institute, September 1999.

[53] KWON, G.-I., AND BYERS, J. Smooth Multirate Multicast Congestion Control. In *Proceedings of the IEEE Infocom* (April 2003), pp. 653–766.

[54] LI, J., DWYER, D., AND BHARGHAVAN, V. A Transport Protocol for Heterogeneous Packet Flows. In *Proceedings of the IEEE Infocom* (1999), pp. 543–550.

[55] LI, W., LING, F., AND CHEN, X. Fine Granularity Scalability in MPEG-4 for Streaming Video. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2000)* (Geneva, Switzerland, May 2000), IEEE, pp. 657–769.

[56] LIU, J., LI, B., AND ZHANG, Y.-Q. Adaptive video multicast over the internet. *IEEE Multimedia 10*, 1 (January/February 2003), pp. 22–31.

[57] MATHIS, M., SEMKE, J., AND MAHDAVI, J. The Macrosopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM Computer Communication Review 27*, 3 (July 1997), pp. 994–1010.

[58] MAYER-PATEL, K., AND ROWE, L. Design and Performance of the Berkeley Continuous Media Toolkit. In *Proceedings of the SPIE Multimedia Computing and Networking Conference* (San Jose, CA, February 1997), M. Freeman, P. Jardetzky, and H. M. Vin, Eds., SPIE, pp. 194–206.

[59] MCCANNE, S., BREWER, E., KATZ, R., ROWE, L., AMIR, E., CHAWATHE, Y., COOPERSMITH, A., MAYER-PATEL, K., RAMAN, S., SCHUETT, A., SIMPSON, D., SWAN, A., TUNG, T. L., WU, D., AND SMITH, B. Toward a common infrastucture for multimedia-networking middleware. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (St. Louis, Missouri, May 1997), pp. 39–49.

[60] MCCANNE, S., AND JACOBSON, V. vic : A flexible framework for packet video. In *ACM Multimedia* (1995), pp. 511–522.

[61] MCCANNE, S., VETTERLI, M., AND JACOBSON, V. Low-Complexity Video Coding for Receiver-driven Layered Multicast. *IEEE Journal on Selected Areas in Communications 16*, 6 (August 1997), pp. 983–1001.

[62] MILLS, D. L. Internet time synchronization: The network time protocol. In *Global States and Time in Distributed Systems,*. IEEE Computer Society Press, 1994, pp. 91–102.

[63] NIST. The NIST Network Emulation Tool. `http://www.antd.nist.gov/itg/nistnet`. *Date viewed: May 2001.*

[64] ONG, L., CORPORATION, C., YOAKUM, J., AND NETWORKS, N. RFC 3286, May 2002.

[65] PADMANABHAN, V. N., WANG, H. J., CHOU, P. A., AND SRIPANIDKULCHAI, K. Distributing streaming media content using cooperative networking. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (Miami Beach, FL, May 2002), pp. 41–51.

[66] REJAIE, R., HANDLEY, M., AND ESTRIN, D. Quality Adaptation for Congestion Controlled Video Playback over the Internet. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (Cambridge, MA, October 1999), pp. 189–200.

[67] REJAIE, R., HANDLEY, M., AND ESTRIN, D. RAP: An end-to-end rate-based congestiong control mechanism for realtime streams in the internet. In *Proceeedings of IEEE Infocomm* (March 1999), pp. 1337–1345.

[68] RHEE, I., OZDEMIR, V., AND YI, Y. TEAR: TCP Emulation at Receivers – Flow Control for Multimedia Streaming. Tech. rep., NCSU, April 2000.

[69] RIZZO, L. pgmcc: a TCP-friendly single-rate multicast congestion control scheme. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)* (2000), pp. 17–28.

[70] ROHALY, A. M., ET AL. Video quality experts group: Current results and future directions. In *Proceedings SPIE Visual Communications and Image Processing* (Perth, Australia, June 2000), vol. 4067, pp. 742–753.

[71] RUBENSTEIN, D., KUROSE, J., AND TOWSLEY, D. The Impact of Multicast Layering on Network Fairness. *IEEE/ACM Transactions on Networking 10*, 2 (2002), pp. 169–182.

[72] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (Nov. 1984), pp. 277–288.

[73] SAROIU, S., GUMMADI, K. P., DUNN, R., GRIBBLE, S. D., AND LEVY, H. M. An analysis of Internet content delivery systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 315–328.

[74] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, January 1996.

[75] SCHULZRINNE, H., RAO, A., AND LANPHIER, R. Real Time Streaming Protocol (RTSP). RFC 2326, April 1998.

[76] SCHULZRINNE, H., RAO, A., AND LANPHIER, R. Real Time Streaming Protocol (RTSP). RFC 2326, April 1998.

[77] SEN, S., AND WANG, J. Analyzing Peer-to-Peer Traffic Across Large Networks. In *Proceedings of the 2nd Internet Measurement Workshop* (Marseille, France, November 2002), pp. 56–68.

[78] SISALEM, D., AND SCHULZRINNE, H. The Loss-Delay Based Adjustment Algorithm: A TCP-Friendly Adaptation Scheme. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (Cambridge, UK., 1998).

[79] SUBRAMANIAN, L., STOICA, I., BALAKRISHNAN, H., AND KATZ, R. H. OverQoS: Offering QoS using Overlays. In *First Workshop on Hop Topics in Networks (HotNets-I)* (October 2002).

[80] The TCP-Friendly Website. http://www.psc.edu/networking/tcp_friendly.html.

[81] TRAN, D., HUA, K., AND DO, T. ZIGZAG: An Efficient Peer-to-Peer Scheme for Media Streaming. In *Proceedings of the IEEE Infocom* (San Francisco, CA, April 2003), pp. 264–278.

[82] UCB/ISI, ET AL. The Network Simulator ns2. `http://www.isi.edu/nsnam/ns/`. *Date viewed: July 2003.*

[83] UNKNOWN. Fast-start vs Streaming. http://www.apple.com/quicktime/. *Date viewed: June 2002.*

[84] VANDALORE, B., FENG, W., JAIN, R., AND FAHMY, S. A Survey of Application Layer Techniques for Adaptive Streaming of Multimedia. *Real-Time Imaging 7*, 3 (2001), pp. 221–235.

[85] VICISANO, L., RIZZO, L., AND CROWCROFT, J. TCP-like congestion control for layered multicast data transfer. In *Proceedings of the IEEE Infocom* (San Francisco, March 1998), pp. 996–1003.

[86] VICKERS, B. J., ALBUQUERQUE, C., AND SUDA, T. Source-adaptive multilayered multicast algorithms for real-time video distribution. *IEEE/ACM Transactions on Networking 8*, 6 (2000), pp. 720–733.

[87] WALPOLE, J., KOSTER, R., CEN, S., COWAN, C., MAIER, D., AND MCNAMEE, D. A Player for Adaptive MPEG Video Streaming Over the Internet. In *Proceedings 26th Applied Imagery Patter Recognition Workship AIPR-97* (Washington, DC, October 1997), SPIE.

[88] WEN, S., GRIFFIOEN, J., AND CALVERT, K. L. Building multicast services from unicast forwarding and ephemeral state. *Computer Networks (Amsterdam, Netherlands: 1999) 38*, 3 (2002), pp. 327–345.

[89] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation* (Boston, MA, December 2002), pp. 255–270.

[90] WILLIAMS, T., KELLEY, C., ET AL. gnuplot. `http//www.gnuplot.info`. *Date viewed: July 2003.*

[91] WILLINGER, W., TAQQU, M. S., SHERMAN, R., AND WILSON, D. V. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking 5*, 1 (1997), pp. 71–86.

[92] WROCLAWSKI, J. The Use of RSVP with IETF Integrated Services. RFC 2210, September 1997.

[93] WU, D., HOU, Y., ZHU, W., ZHANG, Y., AND PEHA, J. Streaming video over the internet: Approaches and directions. *IEEE Transactions on Circuits and Systems for Video Technology 11*, 1 (Feb. 2001), pp. 1–20.

[94] YANO, K., AND MCCANNE, S. The Breadcrumb Forwarding Service: A Synthesis of PGM and EXPRESS to Improve and Simplify Global IP Multicast. *ACM Computer Communication Review 30*, 2 (April 2000), pp. 41–49.

[95] YAVATKAR, R., GRIFFOEN, J., AND SUDAN, M. A reliable dissemination protocol for interactive collaborative applications. In *Proceedings of the ACM Multimedia Conference* (1995), pp. 333–344.

[96] YEADON, N. *Quality of Service Filters for Multimedia Communications.* PhD thesis, Lancaster University, Lancaster, May 1996.

[97] ZINK, M., KUENZEL, O., SCHMITT, J., AND STEINMETZ, R. Subjective Impression of Variations in Layer Encoded Video. In *Proceedings of the International Workshop on Quality of Service (IWQoS)* (Monterey, CA, June 2003), pp. 225–234.

Biographical Note

Charles "Buck" Krasic was born in Hamilton, Ontario on December 25, 1968. He graduated from Cardinal Newman High School in 1987. His undergraduate degree was a B.Math in Computer Science at the University of Waterloo, completed in 1992. As part of his undergraduate program, he did Co-op work terms with four different companies in Hamilton, Ottawa, Toronto, and Vancouver. After his undergraduate he joined the Programming Language Group at the University of Waterloo as a programmer. In late 1993, Buck moved to Los Angeles to work for SHL Sytemhouse. In 1995, he completed a M.Math in Computer Science from Waterloo under the supervision of Professor Gordon Cormack. His M.Math thesis title was "Parametric Overloading in ML". Buck then moved to Portland to take a position as research programmer at OGI.

At OGI, Buck worked on the Synthetix project on developing tools for specialization of operating systems, under the supervision of Calton Pu, Jonathan Walpole, and Crispin Cowan. As part of that work, Buck spent two months with the Compose team of Professor Charles Consel in Rennes, France. After working on Synthetix for three years, Buck decided in 1999 to join the Ph.D program at OGI, with the goal of researching adaptive multimedia under Professor Jonathan Walpole. In the summer of 2000, Buck took a brief hiatus from his Ph.d studies to pursue commercial application of his video research with a startup company called Digital Mercury Inc. (DMI) in Portland. At DMI, Buck had the lead role for a team developing a software solution for adaptive streaming of feature movies to the home. Buck returned to OGI in late 2000 to resume his Ph.d. In winter of 2003, Buck taught Advanced Topics in Networking course at OGI.

Currently, Buck has joined the Department of Computer Science at the University of British Columbia as an assistant professor.