Real-time Support for Interactive Multimedia Applications

by

Aiman Erbad

B.Sc., The University of Washington, 2004

M.Sc., The University of Essex, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

The University Of British Columbia

(Vancouver)

August 2012

Abstract

Emerging interactive multimedia applications, such as real-time visualizations, animations, on-line games, virtual reality, and video conferencing have low latency interactions and continuous high resource (*e.g.*, CPU processing and network bandwidth) demands. The combination of latency sensitive interactions and high resource demands is challenging for best-effort platforms, such as the Internet, general-purpose operating systems and Web browsers because these platforms have no timing or resource guarantees and tend to favor high utilization. When demands exceed available resources, it is impossible to process all computations and data in a timely fashion resulting in diminished perceived quality (e.g., frame rate) and brittle real-time performance. The mismatch between application demands and available resources is observed to varying degrees in all resources including network, processing, and storage.

To deal with the volatility and shortage of resources, we build upon and extend the Priority-Progress quality adaptation model. Our approach enables applications to scale demands (up or down) based on available resources and to utilize the limited resources in processing the computations and data with more influence over perceived quality. We develop enhancement layers to improve timeliness and guarantee more consistent quality using quality adaptation while maintaining the strengths of the existing best-effort transports and execution platforms. DOHA, our execution layer, extends the Priority-Progress CPU adaptation to work in games and across multiple execution threads. The modified game has better timing, higher perceived quality, and linearly scalable quality with a small number of cores. Our transport layer, Paceline, introduces low latency techniques over TCP and exposes Priority-Progress adaptation as an essential transport feature improving upon TCP's end-to-end latency while preserving its fairness and utilization.

Preface

This thesis draws heavily from three papers I have previously published:

- [21] Aiman Erbad, Norman C. Hutchinson, and Charles Krasic. DOHA: Scalable Real-time Web Applications through Adaptive Concurrent Execution. International World Wide Web Conference, Apr 2012.

- [20] Aiman Erbad, Norman C. Hutchinson, and Charles Krasic. Scalable Quality for Web-based Games. ACM SIGPLAN International Workshop on Programming Language And Systems Technologies for Internet Clients, Oct 2011.

- [19] Aiman Erbad, Mahdi Tayarani Najaran, and Charles Krasic. Paceline: Latency Management through Adaptive Output. ACM Multimedia Systems Conference 2010, Feb 2010.

This thesis consists of research I have contributed to during my PhD at UBC. The research have been in collaboration with colleagues at the Networks, Systems, and Security (NSS) Lab. My supervisor is Dr. Norman C. Hutchinson and I have work closely with Dr Charles Krasic, the lead investigator in the QStream project, during his time at UBC. I also worked with Mahdi Tayarani Najaran during his master work.

Paceline [19] was done in collaboration with Dr. Charles Krasic and Mahdi Tayarani Najaran. DOHA [20, 21] was done in collaboration with Dr. Norman C. Hutchinson and Dr. Charles Krasic. The three papers are under the copyright of the ACM and for which I have permission to reuse.

Otherwise, all writing and research work in this thesis was done by myself, with the helpful advice of my colleagues.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to thank the members of the NSS and MAGIC labs, especially Michael Blackstock, Nima Kaviani, Brad Penoff, Jean-Sebastien Legare, Mayukh Saubhasik, Anirban Sinha, Kan Cai, and Mahdi Tayarani Najaran for their help and support. My supervisory committee – Norman C. Hutchinson, Charles Krasic, Eric Wohlstadter and Rodger Lea – provided invaluable feedback on my PhD proposal, research papers, and most recently on earlier drafts of this thesis.

I would like to express my sincere gratitude and appreciation to my supervisor, Norman C. Hutchinson. Norm provided endless support, expert guidance and great patience. Without Norm's help this thesis could not be produced in its present form. I would also like to express sincere appreciation to Dr. Charles 'Buck' Krasic for his support and valuable advice during his time at UBC. Buck's work in the QStream project has provided the inspiration and solid technical foundations on which we built this thesis.

I have been gifted with great brothers that made my stay in Vancouver a memorable experience. I would like to specifically thank Tariq Al-Khasib, Aissa Ikhlef, Ali Al Shidhani, Samer Al-Kiswany, Mohammed Senousy, Hisham Ismail, Yehya Imam, Assem Bsoul, Mohammad Khouj, Abdullah Alsubaie, Safwat Rabae, Ahmad Ashour, Tamer Khattab, Amr Salem, Amr Ayad, Ahmed Maher, Mohammad Badran, Abdullah Gharaibeh, Mohamed Azab, Mohamed Osama, Ghassan Alsharif, Haitham Eissa, Walid Abdulrahman, Hudu Fusheini, Ali Nabi Duman, Ibrahim Gadala, Mohamed Zainal, Ahmed Kafafi, Shaheer Muhammad, Ahmed Atta, Khalid Ibrahim, Wajeeh Moughrabiah, and Mohammed Alrasheed for all the support and the wonderful times we spent together.

Finally, I would like to thank my family for their continuous help and support. And last but not least, I would like to thank God.

Dedication

Dedicated with love to

My parents: Fatima and Mahmood

My wife: Fayza

My beautiful kids: Mahmood and Fatima

Chapter 1

Introduction

Multimedia is becoming more popular with the increasing usage of media-rich social networks, gaming platforms, and video sharing services. Users are expecting multimedia applications with more features, better visual effects, and low latency interactions [45]. To have an engaging user experience and meet expectations, emerging multimedia applications, such as interactive visualizations, animations, on-line games, virtual reality, and video conferencing, have continuous high resource demands and low latency interactions. In addition to the high demands, the resource usage over time in multimedia is bursty and highly volatile [37, 40]. We refer to this ambitious class of multimedia applications as *interactive multimedia*.

Popular interactive multimedia applications use best effort platforms with no timing or resource guarantees, such as the Internet, general purpose operating systems, and Web browsers. Best effort platforms favor high utilization over timeliness because considering both concerns complicates sharing the platform with other applications locally (e.g., the CPU resource) and across the network (i.e., the Internet). The combination of low latency interactions and high resource demands in interactive multimedia is extremely challenging for best-effort platforms especially when demands exceed available resources. Due to the dynamic fluctuations in application demands and in available resources, demands inevitably exceed available resources making it is impossible to consistently process all computations and data in a timely fashion which leads to diminished quality and brittle real-time performance.

Our basic approach to deal with the volatility and shortage of resources in best effort platforms is based on Priority-Progress adaptation [37]. Our approach addresses resource volatility by enabling applications to scale demands (up or down) based on available resources. To efficiently utilize the limited available resources, the computations and data with more influence over perceived quality are given precedence. Unlike conventional multimedia adaptation techniques, Priority-Progress does not require estimation of resource requirements, drastically simplifying its usage. This quality adaptation technique was developed for multimedia video streaming and remains the most stable in terms of packet delay and jitter [42]. Priority-Progress adapts quality based on time and has three main principles: quality is incremental and improves with more iterations similar to the accuracy in iterative algorithms or the video quality in scalable coding; assigned priority is based on the contribution to perceived quality; and finally data is processed according to priority and low priority data is canceled when it becomes stale.

The mismatch between demands and available resources is observed to varying degrees in all resources including network, processing, and storage [37], depending on the application scenario (easy, complex) and the available resources. For ambitious interactive multimedia applications, any of the basic resources could potentially be the limiting factor of the real-time performance [40]. Without an end-to-end approach to performance engineering, it is challenging to improve performance since fixing a concern in one area can cause a new one to arise somewhere else [38]. To reduce the end-to-end delay and improve the overall perceived quality, our work spans multiple resources. We focus on the two most critical resources: CPU and network, leaving other resources, such as storage and memory to future work.

The context of our real-time research is interactive multimedia in the Web. Over time the Web is becoming the de-facto standard distributed application platform so our work can have an impact on the future of distributed multimedia. In addition, multimedia applications using standard Web technologies are easier to port across different platforms (mobile to high-end desktop, and wireless to gigabit network) necessitating an overall framework to handle resource volatility in heterogeneous environments. Even though our work sheds light on some of the unique performance challenges facing interactive multimedia in the Web, we be-

lieve our research contributions address the fundamental requirements of all interactive multimedia applications including server or client components, and native or Web applications.

The rest of the chapter is organized as follows. Section 1.1 describes the requirements of interactive multimedia. Section 1.2 lists the limitations of best effort platforms. Section 1.3 highlights the thesis statement and research contributions.

1.1 Requirements of Interactive Multimedia

Interactive multimedia is primarily driven by user interactions. Real-time interactions have an interaction threshold to maintain comfortable communication between users at both ends. To get an understanding of the stringent timing requirement of interactive applications, we list some of the known figures for interaction delay threshold. International Telecommunication Union G.114 [33] suggests that 150 ms is the ideal delay in most interactive applications, 150 to 400 as the tolerance range, and 400 as the cut-off acceptable delay. Miller's analysis [49] of the threshold levels of human attention suggests that a response time of 100 ms is viewed as instantaneous while a response time of more than 1 second causes users to lose the feeling of operating directly on the data. Finally, the interaction delays in the gaming domain should not exceed 100 ms for first person shooter games, 500 ms for role playing games, and 1000 ms for real-time strategy games [16].

In addition to low latency demands, interactive multimedia has high resource (e.g., CPU and network) demands that vary over time. Games are representative of processing intensive applications. Similar to desktop games [6], popular Web games [17, 68, 76] use most of the available processing power (between 80%-100% of a 2GHz core). For the network resource, multimedia applications have higher bandwidth requirements due to either high quality multimedia (e.g., HD video) or high frequency updates in large scale on-line games (FPS) [10].

The combination of low latency interactions and high resource demands leads to missed deadlines and poor quality for interactive multimedia running in the prevalent best effort platforms. We aim to improve the timeliness limitations and maintain the strengths, such as high utilization and fairness in best effort platforms. The next section presents limitations in best effort transports and execution layers.

1.2 Limitations of Best-effort Platforms

We start with the generic problem statement; and then we analyze the specific limitations facing interactive multimedia in execution and network communication.

1.2.1 Problem Statement

There is a conflict between interactivity and the best-effort nature of the communication and execution platforms of standard client machines. Current transports and execution platforms are optimized for high utilization and do not provide interactive multimedia applications with mechanisms to balance timing with other concerns, such as utilization and fairness when resources are limited. When application demands exceed available resources, the perceived quality (frame rate and jitter) diminishes because it is impossible to satisfy all application demands in a timely fashion. Balancing between these concerns becomes more challenging in concurrent software.

1.2.2 Execution Limitations

When the application demand exceeds available CPU resources, it is not feasible to execute all application computations in a timely fashion. The browser best-effort execution model does not provide any mechanism to balance between timeliness and utilization. One commonly used approach to run interactive applications with consistent quality is to hardcode the appropriate configuration settings, such as games' target frames per second [67, 76]. Static approaches become difficult to manage with the expanding number of platform combinations – browser versions, operating systems, and hardware platforms. More importantly, they can not handle the dynamic fluctuations over time in application demands or in available resources (due to sharing the CPU and network with other applications). In addition, ambitious interactive multimedia applications need more processing power than is available in one core especially in mobile platforms with low-end cores. HTML5 Web workers [30] introduce concurrent execution in browsers; however, workers do not support real-time software developers in addressing challenging issues, such as state management, load-balancing, and timing control across execution threads. Without a general solution that enables scaling demands based on all available re-

sources (including multi-core), the perceived quality of these applications will be brittle and sensitive to any change in the execution conditions.

1.2.3 Transport Limitations

The combination of high bandwidth bursty traffic and low latency interactions in interactive multimedia [77] is challenging to support in the best effort Internet. When demands exceed available network resources, all popular Web transport layers, such as HTTP [23], and SPDY [71] face significant delays. These latency limitations are inherited from TCP, the underlying transport in all existing Web transport layers.

TCP is the dominant transport in the Internet with more than 90% of the traffic volume [25]. TCP has several advantages for high bandwidth multimedia communication especially in regards to congestion-control and reliability [28]. However, high bandwidth communication puts pressure on TCP's best effort nature leading to delays at multiple levels. Firstly, TCP's latency shortcomings are primarily due to *queuing delay* – inside TCP send buffers and network queues. In many realistic conditions the queuing delay in the send side TCP socket buffer is the dominant portion of the overall delay [28].[1] Secondly, TCP retransmissions can add multiple roundtrips to the end-to-end delay. For this reason, TCP is commonly dismissed as unsuited for latency sensitive applications. In the common case, TCP's fast retransmit mechanism limits the retransmission-induced queuing delay to an RTT or two and only very congested networks face exponential back-off and back-to-back retransmission timeouts which degrade TCP's performance. TCP needs to improve its agility by resolving the limitations in the send-side buffers, and retransmission timeouts so it can provide consistent low end-to-end delay under heavy load.

TCP also lacks a data service model that can balance between timeliness, utilization, and fairness while considering application quality. Balancing between these concerns becomes more challenging across multiple concurrent streams because of the diverse requirements. Our solution should address latency at all levels of the transport and enable adapting demands to available resources.

[1] A full kernel socket buffer of size 64KB contributes 1700 ms of delay to a 300 Kbps video stream.

1.3 Thesis Statement and Contributions

This thesis addresses the limitations of best effort platforms (Section 1.2) and ensures higher and more consistent application quality even in concurrent software.

When execution and network demands exceed available resources, it is possible and practical to improve timeliness and ensure more consistent quality in real-time games and video streaming applications by enabling applications to adapt with available resources using Priority-Progress application-level adaptation.

Our approach acknowledges the immense strengths of best-effort platforms. We propose enhancement layers to maintain the strengths and mitigate the weaknesses of the existing communication and execution subsystems using application-level quality adaptation. The next two sections present more details about our approach and highlight the main research contributions.

1.3.1 Priority-Progress Adaptation (PPA)

Our approach adapts multimedia quality (a.k.a. application resource demands) based on the available resources. Quality-adaptation is not a new concept. Many researchers have proposed techniques for multimedia adaptation with much of the pioneering work tracing back to quality adaptive video in the Quasar Project [14]. Priority-Progress adaptation [37, 39] was inspired by several works on quality-adaptive streaming [22, 60, 65]. This line of research understands the importance of the best-effort nature of the Internet and aims to provide consistent multimedia quality by adapting to varying bandwidth availability.

Classic quality-adaptation techniques are based on a feedback loop that balances between application quality and resource usage [27, 44, 66]. Feedback control is formalized in control theory with successful applications in electronics, and is increasingly used in software. However, most classic quality-adaptation controllers assume that there is a reliable way to monitor progress and to estimate the correct control decisions based on monitored values. For network bandwidth in adaptive video, this means the adaptation mechanism should estimate the throughput of real-time video, estimate the network bandwidth that will be available, and from these provide the control decisions that will maximize video quality. Sim-

ilarly, the CPU adaptation mechanism needs to estimate the CPU time required to process data, the amount of CPU that will be available, and from these derive control decisions. PPA avoids two sources of complexity inherent in feedback-based adaptation mechanisms: developing a model for the resource requirements in multimedia applications and estimating resource availability in best-effort environments, drastically simplifying its usage. This adaptation technique adapts based on time [37, 40] with three main principles:

- *Incremental Quality:* The target application can be architected to produce results in an incremental fashion. The application quality improves with successive increments similar to the video quality using scalable coding (e.g., H.364 [64]).

- *Prioritized Data:* Data priority is assigned based on the influence on perceived quality. Priority assignment in application adaptation policies coordinates between different quality dimensions, such as the spatial and temporal dimensions.

- *Priority Data Dropping:* Timestamps and priority are used to adapt quality while maintaining timeliness. Timestamps subdivide time in adaptation windows. Data is being processed from high to low priority and at the end of each adaptation window stale data is canceled.

Prior to our work, the PPA model [37] was only applied to video streaming of stored video. PPA network support [39] was intermixed with the application code without a general API to expose the adaptation primitives in a familiar transport abstraction. The network support also lacked the techniques to minimize TCP's kernel socket buffering and back-to-back retransmissions making it not suitable for real-time scenarios, such as video conferencing. Finally, PPA CPU and network support [39, 40] had no support for concurrent software (e.g., multi-core and concurrent communication streams).

This thesis extends the PPA model to adapt quality in a second interactive multimedia application, games, and enhances the video streaming support to meet the requirements of real-time video conferencing. DOHA [20, 21], our execution layer, extends the PPA model to work across multiple threads with no shared memory.

DOHA is developed as part of this thesis to enhance the event-driven execution model in browsers and enable adaptation in HTML5 games. Paceline [19], our transport enhancement layer, factors out and extends the initial network support (in QStream [37]) providing a session-layer transport with a service model supporting adaptation within message streams and quality fairness across concurrent streams. Paceline also develops latency reduction techniques to mitigate TCP's shortcomings.

DOHA and Paceline extend and enable Priority-Progress adaptation in prevalent best effort transports and execution platforms (i.e., TCP and JavaScript engines). We evaluate Paceline with an experimental video conferencing application [37] and DOHA, our JavaScript enhancement layer, with a popular HTML5 game [76]. Even though DOHA and Paceline have a different codebase, they build upon and extend the Priority-Progress quality adaptation model, and together they provide an end-to-end solution to the real-time performance limitations facing interactive multimedia applications.

Execution Layer: DOHA

Multimedia applications need to scale their quality, and thereby scale processing load, based on the resources that are available. DOHA [21] defines scalable quality, based on the Priority-Progress quality adaptation model, as a necessary requirement to write HTML5 games once and run them with consistent quality everywhere. DOHA also extends Priority-Progress adaptation to work across worker threads. DOHA introduces explicit execution events and enables adaptation based on the following Priority-Progress adaptation principles.

- *Incremental Quality:* The modified game loop executes as many events as possible in each iteration. The perceived quality of a game increases if we can execute more events within the target frame rate.

- *Prioritized Data:* DOHA introduces event prioritization to provide timely execution of those events that have the greatest influence over quality. Priority is based on spatial or temporal indicators of quality, such as the distance from players or the time since last entity update.

- *Priority Data Drop:* DOHA executes events according to priority and introduces event cancellation to adapt the application rate based on available CPU resources. Events become stale and are canceled at the end of each iteration and new events are submitted.

To evaluate DOHA, we modified an HTML5 game, RAPT. The modified game has better timing and higher perceived quality when resources are scarce. More importantly, the overall quality of the parallel game scales linearly as we use more cores and the game is playable in larger scenarios beyond the scope of the original version.

Transport Layer: Paceline

Paceline [19] is an enhanced transport on top of TCP to support interactive, high-bandwidth applications. Even though the underlying service model is best effort, Paceline's latency reduction techniques improve the agility of the transport in responding to network conditions and ensure timeliness for important data. Paceline enables quality adaptation based the following Priority-Progress principles.

- *Incremental Quality:* Our video streaming application uses scalable video coding so quality improves if more data items (frames or enhancement layers) are transferred.

- *Prioritized Data:* Our service model introduces message prioritization to provide timely delivery of important data. Priority is assigned by the application policy using spatial and temporal indicators of perceived quality.

- *Priority Data Drop:* Our service model introduces message cancellation to adapt the application rate based on available bandwidth. Messages are sent according to priority and stale messages are canceled at the end of each adaptation window.

Paceline improves upon the end-to-end latency shortcomings (median and worst case) of using TCP while preserving TCP's fairness and utilization. While Paceline was initially developed to support traditional multimedia, Paceline was exposed in Firefox as a standard Netscape Plugin Application Programming Interface (NPAPI)

[51] browser plugin. Web applications can use Paceline with an API that resembles the Web socket API with the extra adaptation mechanisms.

1.3.2 Primary Contributions

This thesis has the following three primary contributions.

- Our enhancement layers show how to expose adaptation as a transport and execution feature without changing the best effort nature of the underlying platform (i.e., TCP and JavaScript engines). Both layers expose Priority-Progress adaptation mechanisms enabling application developers to implement the necessary policies, improving the latency profile and providing consistent quality when demands exceed available resources in HTML5 games and video conferencing. The adaptation mechanisms become part of the enhancement layers (DOHA and Paceline) while the policies become well-defined within the application code.

- We enhance the adaptation policies and enhancement layers to support concurrent software. To utilize multi-core resources for real-time software, DOHA augments HTML5 Web workers with mechanisms to ease the handling of challenging issues, such as state management, load-balancing, and quality-adaptation across workers. For concurrent communication, Paceline balances between timeliness and fairness among multiple concurrent streams using quality-based fairness.

- We develop adaptation policies inspired by Priority-Progress adaptation in a new application domain, HTML5 games. Our work in DOHA explored the use of Priority-Progress adaptation for CPU quality-adaptation in game loops.

1.3.3 Secondary Contributions

We have the following two secondary contributions.

- We examine the challenges and opportunities of using HTML5 Web workers and share our qualitative and quantitative observations.

- We contribute both Paceline and DOHA with their respective modified applications to QStream's [2] open-source repository to facilitate further research.

1.4 Dissertation Outline

We present background information about interactive multimedia in the Web and multimedia adaptation with a focus on the Priority-Progress model in Chapter 2. We then present DOHA in Chapter 3 and Paceline in Chapter 4. In Chapter 5 we conclude by reviewing the contributions of this dissertation and suggesting avenues for further research.

[2] QStream, located at http://qstream.org, is an experimental media streaming system that takes a comprehensive approach to the end-to-end communication path – both in terms of software layers (application, middleware, and OS) and resource types (network, processor, and storage).

Chapter 2

Background

This chapter provides a brief background about the growing importance of interactive multimedia in the Web and the need for multimedia quality adaptation. Section 2.1 discusses the quality of interactive multimedia in prevalent Web platforms. Section 2.2 introduces quality adaptation in interactive multimedia with an emphasis on Priority-Progress adaptation. Detailed related work analysis for each enhancement layer is presented in the corresponding chapter.

2.1 Interactive Multimedia in the Web

Browsers have become mature platforms enabling Web applications to rival their desktop counterparts. An important class of such applications is interactive multimedia: games, animations, and interactive visualizations. Interactive multimedia in the Web was limited by the lack of key technologies, such as rich graphics elements, bi-directional continuous network transport, and fast JavaScript engines. HTML5 [31] and related standards, such as offline storage, Web sockets, Web workers, and WebGL are enabling more interactive media-rich applications. Interactive multimedia is becoming an integral component of popular Web applications and is expected to become more important in the future.

An ambitious application that exemplifies what the Web stack needs to support in the near future is an HTML5 game which uses the state of the art animations, accurate physics and collision detection, advanced AI, and high quality video chats

for coordination between players. The network transport is required to transmit high quality video as well as player state updates in a timely fashion. Upon receiving the player updates, each player has to execute the client side physics, animation and AI logic and to perform the multimedia encoding/decoding with a consistent rate to achieve the desired game quality. Any slowdown in the network transport or in the execution of logic at the client can degrade perceived quality significantly. Multimedia Web applications are latency sensitive and have high resource demands making them in dire need for techniques to enhance real-time performance when resources are limited.

Interactive multimedia in the Web has high CPU and bandwidth requirements. For the CPU resource, games are great representatives for processing intensive multimedia applications. Popular Web games use most of the available processing power. According to the developers [17, 76], adding new features is limited by the processing capability of the execution platform – i.e., the browser and its version, the operating system, and the underlying hardware. For the network resource, high-definition (HD) video in multimedia Web applications have high bandwidth demands and contribute a significant fraction of the data streamed on the Internet [77]. As audio and video become more tightly integrated in browsers [32, 52, 58, 74], the usage of interactive multimedia in Web applications will increase. When demands for CPU processing and network bandwidth exceed the available resources in best effort Web platforms, the perceived quality and the real-time performance of interactive multimedia diminishes.

To address the limitations of the Web best effort platforms and have an impact on the future of multimedia, our research focuses on interactive multimedia in the Web context. Interactive multimedia is becoming an integral component of popular Web applications. To provide consistent quality and improve timeliness, it is imperative to develop a general framework to adapt the high application demands based on available resources. The next section describes the quality adaptation technique we used to scale demands based on available resources.

2.2 Priority-Progress Adaptation (PPA)

This section describes the state of the art in multimedia quality-adaptation with a focus on Priority-Progress adaptation using adaptive games and video streaming scenarios.

2.2.1 Adaptive Games

Games have high CPU and network demands in loaded servers [15]. A conventional approach to limit the resource demands in game servers is area-of-interest (AOI) geographical partitioning which limits updates only to nearby players within your zone [9]. Geographical partitioning works well when the distribution of players is controlled and player movements are limited. However, population density in real games follows a power law [54], and players move to only a small number of zones during each playing session. Thus, game designers restrict player clustering by partitioning the world into mini-worlds, thereby precluding certain classes of interesting game play, such as epic battles [18]. To meet the high resource (CPU and network) demands in game servers handling popular zones, researchers have designed dynamic load-balancing algorithms [15] which better handle transient crowding by adaptively dispersing or aggregating regions from servers in response to quality of service violations. Load-balancing algorithms are complementary to our work since they do not eliminate the need for instant quality adaptation when demands exceed available resources in a popular server.

To support fast-paced epic scale games, DonneyBrook [10] defines interest sets to reduce the bandwidth requirements of games. DonneyBrook has two priority levels: important and less frequent. Continuous priorities in the Priority-Progress adaptation model can better capture the range of players' interests instead of using two discrete types of updates. Moreover, the cancellation of expired updates in Priority-Progress streaming can enable rate adaptation based on the network conditions without using a complex reservation scheme for important updates. For the CPU resource, games are ambitious processing-intensive multimedia applications. Even simple client-side desktop games [6] or popular Web games [17, 68, 76] use most of the available processing power (between 80%-100% of a 2GHz core). An informal study we did on the architecture of multiple games and graphics engines

[1, 17, 68, 76] revealed that the core of these applications consists of one or multiple execution loops that perform the basic tasks of rendering and simulation. These loops executes 30 to 60 times a second depending on the target frame rate. Current game loops attempt to update all entities at each frame (loop iteration) leading to brittle application quality because adding one feature affects frame duration and can render the game unplayable.

For adaptive games, the focus is on adapting the execution of games in the browser environment. DOHA developed CPU adaptation policies inspired by the Priority-Progress adaptation model for Web-based games and extended the adaptation model to work across parallel threads with no shared memory [20, 21]. In our adaptive game, the execution loop adapts to available CPU at each game loop frame. At the beginning of each frame, the loop cancels the pending events from the previous frame and issues a new event for each game entity. Before submission to the execution layer, the priority policy method for each entity is called to calculate the event importance and then to assign the event priority. Our current policy defines the relative importance among different game entities based on the distance from active players. The relative importance (priority) among game entities dictates the order of event execution in each frame. Our framework easily handles other policies.

2.2.2 Adaptive Video

A video consists of frames, at a constant nominal number of frames per second. To reduce network bandwidth requirements, video frames are encoded at the sender and decoded at the receiver. Scalable video encoders encode each video frame into a *base layer* and a series of *enhancement layers*. The base layer contains vital information required by the decoder to reconstruct a low quality version of the original image. The quality of the decoded frame depends on the number of enhancement layers used, and will resemble that of the original frame if all enhancement layers are used.

Video encoders also try to exploit similarities between different frames, and impose temporal frame dependencies. Based on the type of dependencies, an encoded frame may either depend on no other frame but have other frames depend

on it (I-frame), depend only on previous frames and have frames depend on it (P-frame), or depend both on previous and next frames but have no other frame depend on it (B-frame). Dependencies between frames affect the priorities assigned to each frame/layer. A video application may adapt to available resources by either dropping different layers of a frame (spatial quality adaptation), or dropping an entire frame (temporal quality adaptation), or a combination of both.

Dropping decisions are made according to each frame's time-line, kept both by the sender and receiver. For example, a network frame time-line is initialized when the frame is released (either fetched from storage or captured from camera and encoded), and allows a maximum transmission period (defined by the application) for the frame data to be sent. During the transmission period, frames are sent based on their priority. At the end of the transmission period, i.e., at the transmission deadline, the sender cancels unsent data of the frame. The decoder on the receiving side, which consumes 80-95% of processing time, starts decoding a frame based on the time-line with whatever number of layers it has received for that frame. If no data has been received within the time window, the frame is skipped. This application scenario shows network and CPU adaptation of video streaming.

For adaptive video, the focus is on video streaming in the Internet. Paceline [19], our transport enhancement layer, is layered on top of TCP, the dominant component of Internet traffic volume (typically greater than 90% [25]). Paceline argues that adaptation mechanisms, such as Priority-Progress [37, 39] are essential transport features based on 20 years of multimedia transport research that provides quality of service through adaptation strategies [75]. PPA remains the most stable adaptation technique over TCP in terms of packet delay and jitter [42]. Paceline is a general purpose transport layer exposing a stream API with per-message priority and cancellation. Paceline was used in other applications, such as a cloud-based game prototype to scale the communication in an epic scale game scenario [69]. To address TCP's latency problems and improve timing for important data, Paceline develops the three following techniques: application-level rate control to reduce kernel queuing delay, failover among connections to handle extreme cases of congestion, and application data unit (ADU) fragmentation to reduce the granularity of pre-empting less important data. These techniques in Paceline allowed and verified that the PPA model can be used in real-time interactive video conferencing.

2.3 Summary

Interactive multimedia is an integral component of popular Web applications and is expected to become more important in the future. Interactive multimedia Web applications have high CPU and bandwidth requirements. Quality adaptation is necessary to provide consistent quality and improve timeliness in best effort platforms. Quality-adaptation mechanisms adapt quality (a.k.a. resource usage) based on the available resources. Priority-Progress quality adaptation works across different resources (e.g., CPU and network) and in different application scenarios, such as adaptive video streaming and adaptive games. This thesis extends the Priority-Progress adaptation model to work in a new multimedia application, Web games, and provide a transport and execution layers to enable adaptation in in prevalent best effort transports and execution platforms in the Web (i.e., TCP, JavaScript engines).

Chapter 3

Execution Layer: DOHA

One important area in which applications must adapt to the availability of resources is interactive applications on the Web. Web applications are executed in browsers which historically have focused on the downloading and rendering of mostly static content. However, browsers have recently become mature execution platforms enabling Web applications to rival their desktop counterparts. An important class of such applications is interactive multimedia: games, animations, and interactive visualizations. Unlike many early Web applications, these applications are latency sensitive and processing (CPU and graphics) intensive. Games are great representatives of ambitious processing-intensive multimedia applications in this class. Similar to desktop games [6], popular Web games [17, 68, 76] use most of the available processing power (between 80%-100% of a 2GHz core). Games and other ambitious applications are shifting the performance optimization focus from the download and parsing time of Web files to the run-time performance. The dynamic fluctuations and the scarcity of processing resources limit game features and lead to significant development effort to manage the resource demands [17, 76].

When demands exceed available CPU resources in interactive multimedia, it is not feasible to execute all application computations (callback functions) in a timely fashion. The browser best-effort execution model does not provide any mechanism to balance between timeliness and utilization. One commonly used approach to run interactive applications with consistent quality is to hard code the appropriate configuration settings, such as the games' target frames per second [67, 76]. This

static approach cannot keep up with the expanding number of platform combinations (browser version, hardware, and operating system). More importantly, it does not gracefully handle the dynamic fluctuations in application demands (common in multimedia applications) or available resources (due to sharing the CPU with other applications). Another major concern related to the prevalent processing model in browsers is single-threaded execution. Ambitious multimedia applications need more processing power than available in one core especially on mobile platforms with low-end cores. To deliver the available multi-core cycles to Web applications, we need to facilitate real-time concurrent software development. Although HTML5 Web workers [30] enable concurrent execution as seen in Figure 3.1, they do not help developers address challenging issues in concurrent software, such as state management, load-balancing, and timely execution across threads. Without a general solution that deals with the fluctuations and scarcity in processing resources, the perceived quality of these applications becomes brittle and sensitive to any change in the execution conditions.

DOHA is an execution layer written in JavaScript that enhances the browser execution model. Our basic approach in DOHA to deal with the volatility and shortage of processing resources is based on Priority-Progress adaptation [37]. DOHA defines scalable quality as a necessary requirement to write Web applications once and run them with consistent quality everywhere. Scalable quality addresses resource volatility by enabling applications to scale demands (up or down) based on available resources (including multi-core) and to efficiently utilize the limited available resources by giving precedence to important computations with more influence over perceived quality. Priority-Progress adapts based on time and uses the following three principles:

- *Incremental Quality:* The modified game loop executes as many events as possible in each frame. The perceived quality of a game increases if we can execute more events within the target frame rate.

- *Prioritized Data:* DOHA introduces event prioritization to provide timely execution of those events with the greatest influence over quality. Priority is assigned based on distance from the players which is one of the spatial indicators of quality in a game.

- *Priority Data Drop:* DOHA executes events according to priority and introduces event cancellation to adapt the application demands based on available CPU resources. When execution events become stale at the end of each game frame, they are canceled and a new set of events is submitted.

DOHA also extends Priority-Progress adaptation to work across worker threads utilizing the widely available multi-core processors. To utilize multi-core resources, DOHA augments HTML5 Web workers with mechanisms to ease handling challenging concurrency issues, such as state management and load-balancing. The modified game using DOHA has better timing and higher perceived quality when resources are scarce. More importantly, the overall quality scales linearly (up to 3 cores) and larger game scenarios, beyond the scope of the original game, are playable in the parallel version of the game.

The remainder of the chapter is structured as follows: Section 3.1 discusses DOHA's design and implementation details; Section 3.2 explains our evaluation results; Section 3.3 describes our qualitative lessons learned; Section 3.4 presents the related work; and Section 3.5 concludes.

3.1 Design and Implementation

While studying the architecture of a variety of Web-based games [1, 17, 68, 76], we observed that they have one or multiple execution loops to perform the basic tasks, such as rendering and simulation. As we see in Figure 3.2, current game loops have a global update that iterates over all game entities (e.g., players and enemies) in a pre-determined order (creation time order in RAPT [76] and z-axis ordering in the Render Engine [17]). The global update is called using a JavaScript timer 30 to 60 times a second depending on the target frame rate. Current game loops attempt to update all entities at each frame in a timely fashion. This architecture leads to brittle application quality because adding one feature affects frame duration and can render the game unplayable.

DOHA provides Web applications with abstractions and an execution layer to have better control over quality and have more access to available multi-core resources. DOHA consists of two major components: the event-loop which handles prioritized execution locally in each thread, and the concurrent execution module,

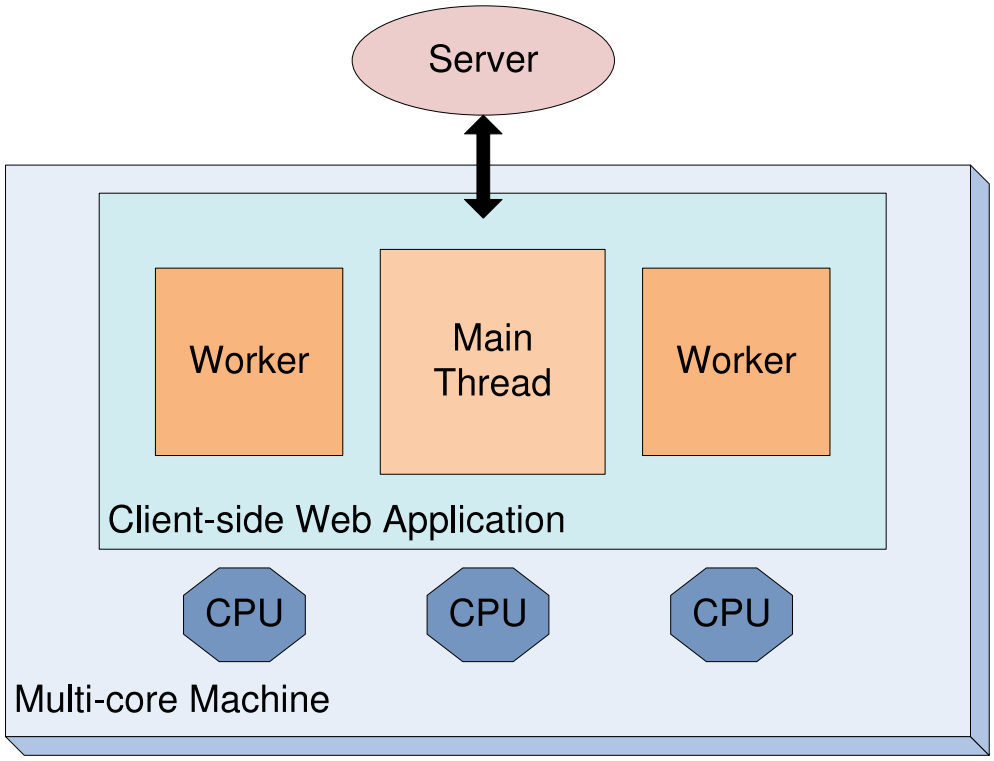


Figure 3.1: Web application with two HTML5 workers running in a multi-core platform

```
function update (time) {
  // Call update for all entities
  for(entity in game_entities){
      entity.update (time);
  }
}
```

Figure 3.2: Game loop global update function

MultiProc, which simplifies state management and scheduling of events on worker threads.

3.1.1 Event-loop

An event-based architecture is a natural fit for the asynchronous browser execution environment. DOHA's event-driven programming model is inspired by the principles of reactive programming [5] and aims to support the needs of interactive Web applications. Popular Web applications are event-driven with a large number of short callback functions [59]. DOHA introduces explicit execution events that specify the function to be executed and the call parameters. Inspired by Priority-Progress adaptation [39], the scalable but computationally intensive parts of games are broken into explicit events. Explicit execution events in DOHA give the underlying scheduler performance hints and define the granularity at which applications adapt (scale quality up and down).

Our key observation is that time-sensitive applications have some computations that are time synchronous (e.g., sound and game loop updates) and others that are best-effort (e.g., AI logic and the particle engine) and can be adapted. These two types of computations need to be clearly identified so that their needs can be met independently. DOHA provides an event-loop abstraction to ensure timely event execution [41]. The event-loop dispatches non-preemptively, prioritizing the time synchronous computations over the best-effort computations. Events can be dispatched with low latency because our event-based model should ideally have short-lived computations that avoid blocking.

The key primitives in the event-loop are: *submit* an event for execution (and start the execution loop if it was not active), *run* to start the execution loop, *cancel* to delete a submitted event before it is executed, and *stop* to pause the execution loop. Each event is given a type, a callback specifying the function that will be called, and an array of arguments. Explicit execution events have two types: *timer and best-effort*. In timer events, the release time specifies the time an event becomes eligible to execute. Once eligible, timer events take precedence over best-effort events. For best-effort events, execution is ordered according to priority. When application demands exceed available resources, it is not feasible

to dispatch all events in a timely fashion. Best-effort events with more influence over perceived quality are given high priority so they execute first. Less important events are canceled when they become stale (based on time), thereby matching demands to available resources. Priority and release time are assigned by an application-specific adaptation policy.

To order the execution of both event types, the event-loop has two internal priority queues. Timer events use a min-heap so events with earlier release times are closer to the heap root. Best effort events use a max-heap so higher priority events are closer to the root. At each event-loop iteration, we peek at the timer's heap root to examine the closest release-time. If it has been reached, we execute the root event. If it has not been reached, we execute the best-effort heap root. If the best-effort events heap is empty, we yield to the underlying JavaScript engine until the closest release-time. If the timers heap is empty, we yield execution of the event-loop until a new event is submitted.

When an event is canceled or executed, it is removed from the associated heap (while maintaining the heap property). Our event-loop is minimal and is designed to co-exist with the underlying JavaScript engine. We see our event-loop as an enhancement layer to add the essential adaptation mechanisms: priority and cancellation. Heaps in our design allow applications to queue events improving utilization while keeping full control over timing through prioritized execution and cancellation. To avoid blocking the underlying JavaScript engine, we can run DOHA's event-loop in a timed mode by setting a threshold (e.g., 200 ms) for the maximum duration of an event-loop iteration.

RAPT: Events and Policies

As a case study for DOHA, we choose the game Robots Are People Too (RAPT) [76]. RAPT won the most fun game award in Mozilla's Game On contest [72]. RAPT is an HTML5 platform game ported from C++. Players jump between moving platforms and coordinate their movements in order to pass game levels. The exit to each game level is blocked by enemies that roll, jump, fly, and shoot to prevent escape. RAPT uses 100% of a single core CPU (2GHz). To understand the time profile of different game components, we used the internal browser pro-

```
function update (time) {
  var evt;
  // Delete pending events
  for(evt in pending_events){
    eventloop.cancel(evt);
  }
  // Add events to event-loop with a priority
  for(entity in game_entities){
    evt = new Event(entity.update, [time], BestEffort);
    evt.priority = entity.getPriority();
    eventloop.submit(evt);
  }
}
```

Figure 3.3: Modified simulation loop update call

filer. The performance of RAPT is impacted by two major components: graphics and simulation (physics and collision detection). In Chrome, 50% of the time is spent rendering, 30% on the simulation update, and around 20% is spent inside the browser. The major components in terms of performance (graphics, simulation, and AI) are similar to traditional desktop games [6]. We focused our experiments on the simulation updates because it constitutes a large performance concern especially after the rendering in browsers becomes hardware-accelerated.

Our first task was to split the large monolithic simulation loop into small explicit update events reducing the granularity of adaptation to an execution event. The main simulation loop is now triggered by a timer event executing the global update function at a rate of 30 frames per second (33ms frame duration). Timer events triggering the global simulation update take precedence over best-effort events submitted within each frame. As shown in Figure 3.3, the modified global simulation update starts by canceling the pending best-effort events from the previous frame. Then, a separate update event per game entity is created and submitted to the underlying event-loop. Before an update event is submitted, the *getPriority* policy method for each entity is called to calculate the event importance.

Adaptation policies developed for the game define relative importance among different game entities in each game loop iteration (game frame). Relative importance (priority) among game entities dictates the order of event execution. Since players are at the heart of a game, their updates are the most critical indicator of perceived game quality. Our basic adaptation policy assigns priority based on dis-

tance from active players. Priority is a number between 0.0 and 1.0. Players get priority 1.0. The priority assigned to the update events of other entities is inversely proportional to their distance from the closest player.

Using distance only can lead to starvation for distant entities. These entities will not be updated if resources are limited which causes flaws in their physics updates. To minimize starvation and ensure correct simulation for all game entities, we defined a minimum update heuristic based on the time since last update. As the time since last update increases, the priority increases to reach 1.0 when we exceed a maximum time threshold between updates. Finally, game entities have some non-linear behaviors, such as gravity. These behaviors limit scalability because they require a high and consistent update rate. Our policy needs to detect and account for these behaviors while assigning priority.

All our entities sub-class two base classes: enemy and player. These base classes define the adaptation policies other entities inherit. This current policy can be customized at run-time with the appropriate thresholds, such as minimum update threshold and distance ranges. We can also override a policy to include other factors specific to an entity type. For example, we can increase the priority of a bullet proportional to its speed.

It is important to note that with simple modifications to the main simulation update loop, it was possible to scale quality using DOHA's event-loop. Web-based games have other places where scalability can help trade accuracy for performance, such as the particle engine (visual effects accuracy) and AI logic (algorithm accuracy). DOHA accompanied with the necessary adaptation policies can enable scalability in these places to provide better and more consistent quality.

3.1.2 MultiProc: Concurrent Execution

HTML5 Web workers are implemented using threads in major browsers and utilize multi-core hardware if available. Worker threads were envisioned to provide an API to run scripts in the background without locking the user interface [30]. Since their inception, Web workers have been used in computationally expensive demo applications to speed-up highly parallel algorithms. For example, our parallel factorial micro-benchmark gets around 10x speed-up with 16 cores for large

numbers ($3 * 10^9$ in Chrome).

We believe Web workers have a larger role in enhancing the performance of interactive multimedia Web applications especially in the mobile Web. Mobile platforms have low-end multi-core processors (e.g., 600 MHz) and browsing performance is the biggest barrier to entry for a large number of ambitious Web applications. Using one of the most challenging Web application domains, HTML5 games, we show that Web workers with appropriate support can significantly improve performance and perceived quality.

MultiProc API

MultiProc provides mechanisms to write concurrent Web applications with different architectures. As shown in Figure 3.4, we started with a central master/slave architecture that is tightly coupled. This architecture works well when the application state resides in the main thread and the computations have minimal shared state. The application computations along with their necessary state (i.e., events with parameters) are submitted to a central scheduler that dispatches them for execution on worker threads. *remote_submit* is used to submit a remote event to the central scheduler. The scheduler (in the main thread) decides where to execute each event based on worker load statistics (orders workers based on load). Before events are assigned to a worker, they can be canceled using *remote_cancel*. To inform the main thread that an event was executed successfully, a worker calls *done*. This call updates the worker load statistics (number of active events).

To address the high communication costs across workers (as shown in Table 3.2), we moved to a less central design where the code in workers is more independent. Concurrent Web applications with expensive communication are similar to distributed systems. To share state between application components across workers, MultiProc introduces a publish-subscribe communication API and RPC events. To send a direct RPC event to a specific worker bypassing the central scheduler, we use the *remote_direct_submit* call. Shared state can be published using *publish_state*. The state is transferred across worker boundaries and the method that subscribed for the state updates using the *subscribe_state* API call is notified.

```
//Central scheduler API
remote_submit(Event e);
remote_cancel(Event e);
done(Event e);


//RPC and state management API
remote_direct_submit(Event e);
publish_state(topic, msg);
subscribe_state(topic, worker_id, function_name);
unsubscribe_state(topic, worker_id);
```

Figure 3.4: MultiProc public API

Central Hierarchical Design

MultiProc started with support for a centralized master/slave Web application architecture. The main browser thread is the master dispatching events to slave workers. The main thread and workers each run their own event-loop to manage event execution. Worker creation, book-keeping, and scheduling decisions happen in the centralized scheduler. This central design is based on the observation that the main thread handles the Document Object Model (DOM) and that workers can only communicate with their parents (no direct communication between siblings). To extend our adaptation model across workers, events are queued in the main scheduler and sent according to their priority. DOHA's central scheduler allows a small fixed window of events in-flight per worker. Upon notification that an event was executed successfully, another event is dispatched to the same worker. Since the window size is small, the scheduler is agile in responding to load imbalance among workers.

When an event reaches a worker, it is passed to the application code. The application code at the worker adds the event to the local event-loop which respects its priority. Each level of DOHA orders events according to their priority to approximate a distributed adaptive event-loop across workers. To balance load across workers, events are assigned to the worker with the smallest load (number of events in progress). MultiProc uses two heaps to manage remote events and workers. Remote events are ordered according to their priority while workers are ordered according to the number of active events.

Percentage Of Unique Colors	0	25	50	75	100
Delay (ms)	501	393	342	320	310

Table 3.1: Average delay to render a frame using ray tracing

Assuming each event can be executed in any worker, the centralized scheduler will have perfect load balancing. However, some events have dependencies (e.g., manipulate the same data-structure). To handle these dependencies between events our framework uses event *coloring* [82]. Programmers color events and MultiProc adheres to the coloring constraints. Events with the same color execute in the same worker and events with different colors can execute in parallel (in different workers). Workers can generate events and assign them unique colors. Since our design is centralized, a worker delegates the event to the main thread (master) which assigns it to the appropriate worker. Coloring is an easy to adopt [82] yet powerful concurrency control mechanism. If all events have the default color, we have a serial program. Having more colors reduces the scheduling constraints which leads to better load balancing across workers.

To test dynamic load balancing in the central design, we used a 3D animation that renders a large frame using ray tracing. Ray tracing is computationally intensive and has minimal shared state. To simulate a loaded worker, we limit the CPU share of one worker (out of 4 worker threads) to be 25% of the CPU time. To simulate dependency between events, we vary the percentage of events with unique colors. 100% means each event has a unique color (no dependency) and 0% means each event is colored with one of the four major colors (25% of the total events per color to distribute work evenly across 4 workers). The original rendering application which uses round robin scheduling takes 309ms to render a frame when all the workers have enough resources and takes around 500ms when one worker is limited. In Table 3.1, we see that MultiProc central load balancing algorithm assigned events to other less loaded workers reducing the impact of the loaded worker and maintaining the same overall application execution time when each event has a unique color (100%). As the percentage of events with unique colors decreases (more dependencies between events), the rendering task gets delayed significantly. The load balancing logic was not able to run as many events in parallel and gets the same results as the original round robin version.

Our initial design assumed Web applications have a central design where all execution events pass by the MultiProc scheduler. Even though our results with a simple application were positive, in the central design all application state accessed during each computation and the generated results cross worker boundaries. The communication cost becomes prohibitively expensive in complex applications, such as games, with tightly coupled components sharing state. Table 3.2 shows the high costs of a ping-pong message in HTML5 Web workers as we vary the message size.[1] 3ms is a relatively high cost considering the 33.3ms frame duration (or 16.6ms with a rate of 60 frames per second).

Message Size (bytes)	10	100	1K	10K	100K
Firefox (ms)	3	3	2.3	3	4.5
Chrome 15 (ms)	3.4	4	4.5	6	46.9

Table 3.2: Average delay for a ping-pong message between workers

State Management and Publish-Subscribe

To address these high communication costs, we moved to a less central design where the code in workers is more independent. We re-structured the simulation loop of RAPT as a network of components running in workers. As we see in Figure 3.5, each worker has an event-loop to run local events. Instead of sending all events and their related state across worker boundaries, we send a few direct events (synchronization and control events) and necessary state updates between workers. For example, we send a game loop start iteration event from the main thread with minimum data parameters, such as the current time. Update events for entities assigned to the worker are generated locally and added to the local event-loop.

Without shared memory, workers cannot access the browser Document Object Model (DOM). The code for each game entity in RAPT had to be split into two parts: one for simulation which runs in worker threads and another for rendering which runs in the main thread. The modified game loop performs rendering in the main thread while the loop in each worker performs the simulation. To maintain

[1] Internet Explorer (IE) is not included because at the beginning of our study, IE did not support Web workers.

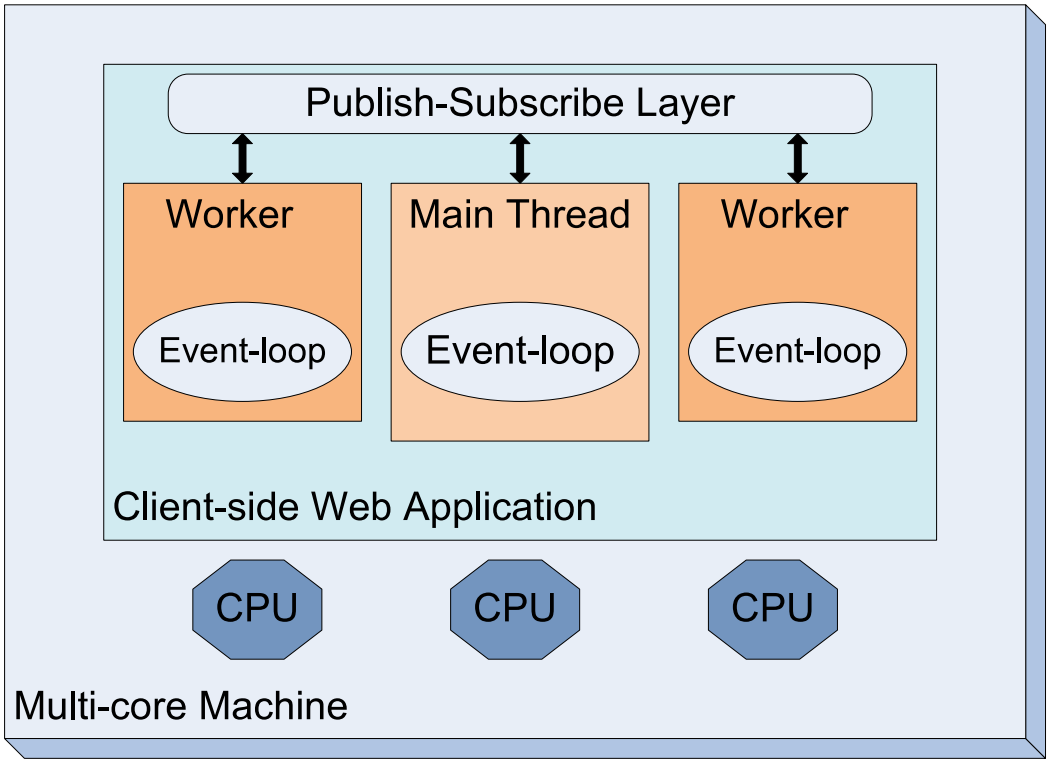


Figure 3.5: Web application using MultiProc with two workers

the game view, the rendering state of each entity is replicated. After each simulation update, each entity communicates the state needed for rendering back to the rendering replica in the main thread. This partial replication of the entity's state uses our publish-subscribe communication API, as we see in Figure 3.6. Partial replication transfers the minimum amount of state needed for rendering, such as the entity position (x, y), and orientation (angle).

Few key entities in RAPT are global. For example, players are accessed and modified by different types of enemies in multiple workers. Similarly, some entities at the boundary of partitions need to have their state shared between two workers. To perform correct simulation, the entire state of each global entity is replicated across multiple workers. One worker owns the primary (authoritative) copy of the entity and other workers have full replicas. We synchronize all replicas after each entity update. The primary publishes its state to the entity's topic which all replicas

```
//Publish state in the worker
Entity.prototype.publishState = function()
{
   var msg = [this.x, this.y, this.angle];
   worker.publish_state(this.id, msg);
};
//Update state in the main thread
Entity.prototype.updateState = function(args)
{
   this.x = args[0]; this.y = args[1];
   this.angle = args[2];
};
```

Figure 3.6: Entity sharing rendering state in concurrent RAPT using the publish-subscribe API

(partial and full) subscribe to.

To allow modifying global objects, each identical replica acts as a proxy. State mutation is only allowed in the authoritative version of an entity. When a mutator method in a replica is called, the call is published on the global object mutation topic which the authoritative version subscribes to. State management for entities heavily uses the publish-subscribe API for one-to-one (partial replication), one-to-many (full replication), and many-to-one (proxy forwarding) communication. These different communication patterns and the dynamic movements of entities to balance load across workers are the main motivations for our publish-subscribe communication API. Publish-subscribe provides a loosely coupled communication API that supports various communication patterns and allows the communication pattern to vary over time.

Our publish-subscribe logic is central. Web workers pass the communication API calls to the main browser thread. Our main publish-subscribe unit maps topics to a subscribers list. Each subscriber is a tuple of (worker ID, function name). When a message is received from the topic, it is forwarded to a function with the given name on the specified worker. In each worker, the application registers a list of public functions that handle state-update messages.

The topics used for publish-subscribe communication need to be unique. We built a distributed identity manager to provide each game entity (in RAPT) with a unique identity that is used as a topic for its communication. The primary identity manager in the main browser thread assigns each worker a limited range in the

identity space. When the identity range in a worker runs out, the remote identity manager asks the primary manager for a new range.

Load-Balancing

Our central scheduler implements load-balancing as we saw in Section 3.1.2. However, DOHA's state management support is agnostic to the way application components are distributed across workers. Building efficient distributed algorithms for games is an active area of research that is outside the scope of our work. We aim to provide the necessary mechanisms so application developers can implement their favorite distributed load-balancing algorithms on top of DOHA.

In the concurrent version of RAPT, we partition the game map geographically into a number of grids equal to the number of workers. Each worker handles a grid with all associated entities (enemies, and players). The state of each entity is updated in a single worker. This design respects data locality since each entity primarily interacts with other entities in its vicinity. Local interactions avoid expensive state transfer across worker boundaries. When entities move between grids, they migrate with all their state to a different worker.

Even though static geographical partitioning does not distribute work evenly across workers, our experience in a few popular state of the art Web-based games suggests that designers distribute game entities evenly across the game map. To help developers implement the load-balancing algorithms, DOHA provides:

- load information so developers can use it to decide when to migrate entities.
- a distributed identity manager which names entities uniquely, thus avoiding name conflicts upon migration.
- A loosely-coupled communication API to easily set-up and tear-down communication channels for frequent entity migration.

Developers need to develop the load-balancing policy and then use our communication layer to send the entity state.

DOHA aims to support applications with different concurrency requirements, ranging from simple applications that only need the computational benefits of Web

workers to the more demanding Web-based games. Simple applications without shared state can delegate load balancing and scheduling of remote events to the central scheduler. For more advanced applications with shared state across workers, DOHA provides a publish-subscribe communication layer to manage state. In our efforts to parallelize RAPT, we initially tempted to isolate a major game component such as the physics engine in a worker. This would have been easier and can probably enhance performance. However, it does not scale with the number of cores. Even though current mobile platforms have at most dual-core processors, RAPT and other Web applications should aim for scalable parallelism to improve performance with more cores.

3.2 Evaluation

We conducted a set of experiments with gaming scenarios of various computational demands. In the basic test map for RAPT, both players move inside a horizontal tunnel in one direction and the enemies move in a parallel tunnel above the players. We compare the following game versions: the original RAPT (RAPT), the modified RAPT using adaptation only (RAPT-A), and the modified RAPT using adaptation and concurrent execution with 2 Web workers (RAPT-C).

Our evaluation takes two views on performance: the first based on lower-level event-loop execution metrics, and the second based on higher-level application metrics. The low-level metrics include: number of events submitted per second and the ratio of canceled events. These low-level metrics show the throughput of the event-loop (events per second). To understand how these low level metrics affect game quality, we analyze the quality of the gameplay experience using high-level metrics, such as the simulation loop jitter profile (jitter median and jitter tail which is the 95th percentile of the jitter distribution) to quantify the average timeliness and the magnitude of execution glitches, and the average frames per second (FPS) versus priority for all entities to quantify the average game quality (scalable quality).

We performed our experiments on an AMD Opteron with 16 2GHz cores. Multi-core hardware allowed Web workers to run on different cores. The duration of each experiment is 80 seconds. To avoid start-up and shutdown effects, we

Table 3.3: Jitter profile

Jitter (ms)	Scenario	RAPT	RAPT-A	RAPT-C
Median	Easy	22	0	1
	Medium	47	0	0
	Hard	219	0	0
Tail	Easy	26	7	7
	Medium	55	8	17
	Hard	291	8	33

use the middle 60 seconds. We used Google Chrome 15.0.874.102 beta in Ubuntu 10.04 LTS. The two changing experiment parameters are the computational difficulty of the game scenario (which is controlled by the number and type of enemies) and the game version (RAPT, RAPT-A, and RAPT-C). We have three game scenarios: an easy scenario where all versions have reasonable quality; a medium scenario which is the hardest playable scenario by RAPT and RAPT-A (with the processing power of one core); and finally an extremely challenging game scenario with processing requirements beyond the capacity of one core.

3.2.1 Adaptive Execution

In this section we discuss the effects of our adaptation model on game performance. We analyze the low-level event-loop throughput, the timeliness of simulation loop updates, and the average overall game quality (scalable quality).

Timeliness

Table 3.3 shows the simulation loop jitter profile for all RAPT versions running all scenarios. The median jitter gives a measure of average timeliness and agility in responding to stimuli, such as input and collisions. To quantify glitches which affect quality negatively, we measure the jitter tail. The expected inter-arrival time between frames is 33.3ms since the target frame rate is 30FPS (frame duration 1000ms/30). We measure the offset for the expected arrival time and report its median and 95th percentile to capture the jitter distribution.

As seen in Table 3.3, the jitter median and tail in the original RAPT increases with the difficulty of the game scenario. The median jitter reaches 219ms which means RAPT executes 1 out of 7 frames (219/33.3=6.6) yielding a frame rate of

around 4 FPS. This increasing jitter is due to RAPT's game loop iterating over all game entities in each frame leading to a large delay in processing each frame.

RAPT-A has low consistent jitter profile (median=0ms and tail=8ms) for all game scenarios. Our reactive event-driven design gives timer events more importance than best-effort events in each game loop frame. When the timer event running the game simulation loop (global update) fires, we stop execution of best-effort events and delete all pending events from the previous frame. Finally, RAPT-C has low consistent median jitter. But the jitter tail in RAPT-C increases with the difficulty of the scenario to reach 33ms in the hard scenario (95% of the jitter values are less than 33ms). This increase in the jitter tail is mainly due to the communication and OS scheduling spikes for the two Web workers.

Low-Level Event-loop Statistics

Table 3.4: Event throughput statistics

Statistics	Scenario	RAPT-A	RAPT-C
Event Submission Rate	Easy	7417	7127
	Medium	11150	10749
	Hard	33110	31317
Cancellation Ratio (%)	Easy	2.2	0.1
	Medium	17	18
	Hard	88	66

Our low-level event-loop statistics help us understand the event throughput. As seen in Table 3.4, both RAPT-A and RAPT-C submit more events as the difficulty increases because of the increase in the number of entities. The cancellation ratio also increases because the frame duration is not enough to update all entities as the difficulty increases.

RAPT-A submits slightly more events per second in all scenarios than RAPT-C indicating a higher simulation rate. The ratio of canceled events in the easy and medium scenarios (RAPT-A and RAPT-C) is comparable. In the hard scenario, RAPT-C cancels less events (66%) than RAPT-A (88%). Moreover, RAPT-C has higher event rate due to having 2 extra cores to execute the simulation updates.

Priority Vs Quality

The medium scenario (hardest playable scenario for RAPT-A and RAPT) tests our capability to gracefully degrade quality when resources are scarce. RAPT in the medium scenario has an average quality of 12.3 FPS which is similar to the simulation rate. When resources are scarce, RAPT-A and RAPT-C cancel update events for stale low priority entities. Thus, the overall average game quality (in FPS) is not captured in the simulation rate (29 and 30 FPS).

To give more meaningful measure of the overall game quality (scalable quality), we measure the average frames per second for all game entities. We correlate this quality indicator with the priority assigned by our policy. As we see in Figure 3.7, the FPS of game enemies in RAPT-A ranges from 16 for low priority entities to 29 (maximum) for high priority entities. Similarly, the average jitter for all entities decreases from 17ms to 4ms as priority increases (average jitter is inversely related to average FPS). We notice that low priority entities never starve (have at least 16 FPS). This is due to our minimum update threshold which ensures that even low priority entities are updated at a lower frequency. When CPU is limited, our adaptation model in RAPT-A improves quality for important entities so quality has a strong correlation with priority.

RAPT-C has relatively higher quality for all entities (24 FPS). The quality does not have a strong correlation with priority because at each time instance game entities with the highest priority are concentrated in one worker (due to geographical partitioning of entities). Other workers at the same time instance are processing events with low priority leading average FPS to lose correlation with priority. In addition, the communication overhead in RAPT-C with three separate cores (2 workers and the browser thread) leads to lower FPS than RAPT-A in the medium scenario for high priority entities. The trade-off between the communication overhead and the parallel speed-up changes in the hard scenario as we see in Section 3.2.2.

Results Summary

RAPT-A and RAPT-C have better timing (lower jitter mean and tail) than the original RAPT. By canceling updates of less important entities at the end of each sim-

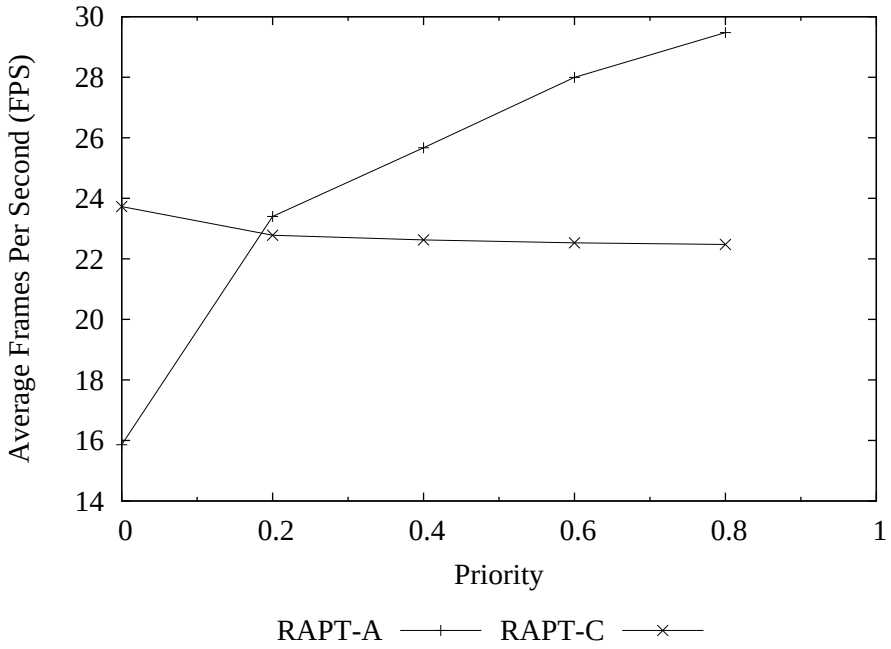


Figure 3.7: Priority versus quality (average FPS)

ulation loop, RAPT-A and RAPT-C can provide important entities (with more influence over quality) a higher update rate. This translates to better overall game quality in RAPT-A and RAPT-C.

3.2.2 Concurrent Execution

While playing the game, we noticed RAPT-A had much worse perceived quality than RAPT-C in the hard scenario. RAPT-A's main thread was overwhelmed by the extremely high load and it was not yielding execution to the browser engine (to perform the rendering). In this case, isolation between the two tasks (simulation and rendering) in RAPT-C provided much better perceived quality.

The hard scenario is not playable in either RAPT or RAPT-A. RAPT-A was overwhelmed by the load and the simulation rate in the original RAPT is extremely low (4 FPS). The hard scenario is only playable in RAPT-C. Even though the hard scenario had a large number of enemies, our design scales the communication costs. RAPT-C only executes and communicates state updates for as many events as the frame duration allows.

To evaluate the effects of adding more cores on game quality, we run RAPT-C using the hard scenario while varying the number of cores. As we see in Figure 3.8, the average FPS for all game entities (scalable quality) increases as we add more cores. RAPT-C with 1 worker gets an average of 3.5 FPS. With 3 workers, the average FPS is between 12 and 14. Average jitter also drops from 260ms with 1 worker to around 42ms with 3 workers. In the hard scenario, we get linear improvement in quality with each worker added up to 3 workers. As we see in Figure 3.8, using 4 workers does not improve the game quality. With 4 workers, only 3% of the execution events are canceled which indicates the application load in the hard scenario is small relative to the available cores. In this case, RAPT-C pays additional concurrency costs but does not benefit from the extra core.

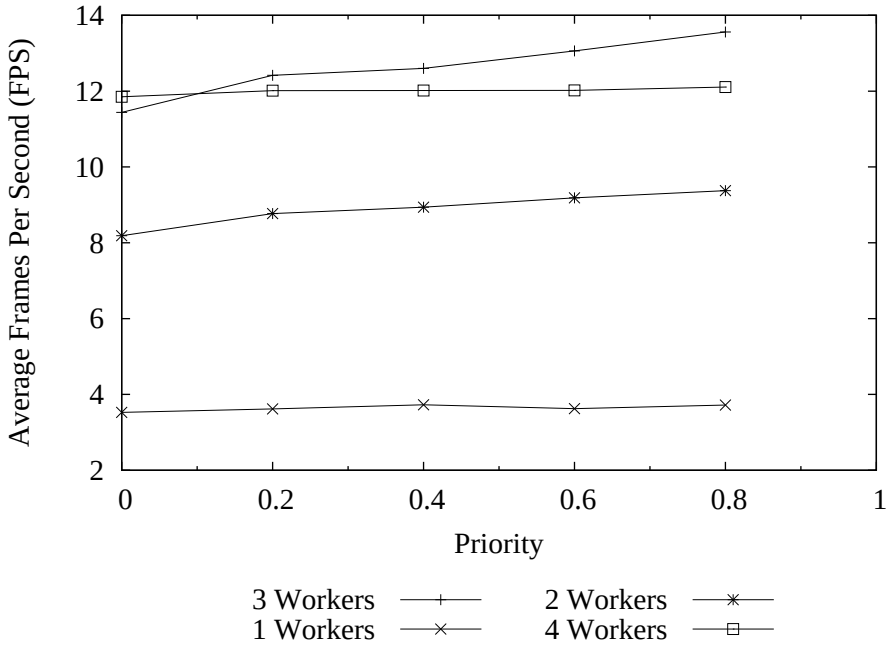


Figure 3.8: RAPT FPS in the hard scenario with 1, 2, 3 and 4 workers

To look at timeliness as we add more cores, we measure the jitter distribution of the simulation loop frames in the hard scenario. RAPT has by far the worst jitter profile. As we see in Figure 3.9, only 20% of execution frames have jitter less than 210ms. RAPT's jitter tail extends to around 291ms causing significantly bigger execution glitches. RAPT-A and RAPT-C (with 1 and 2 workers) have the same

low average jitter profile and RAPT-C has a relatively worse jitter tail.

We also observe that latency increases as we add more cores. With 3 workers, the mean jitter is 14ms and the jitter tail is 64ms. The jitter increase is partly because the simulation loop in worker threads is triggered by a periodic update event sent from the main thread. When the number of workers increase, the communication load on the main thread increases and the loop update events are delayed.

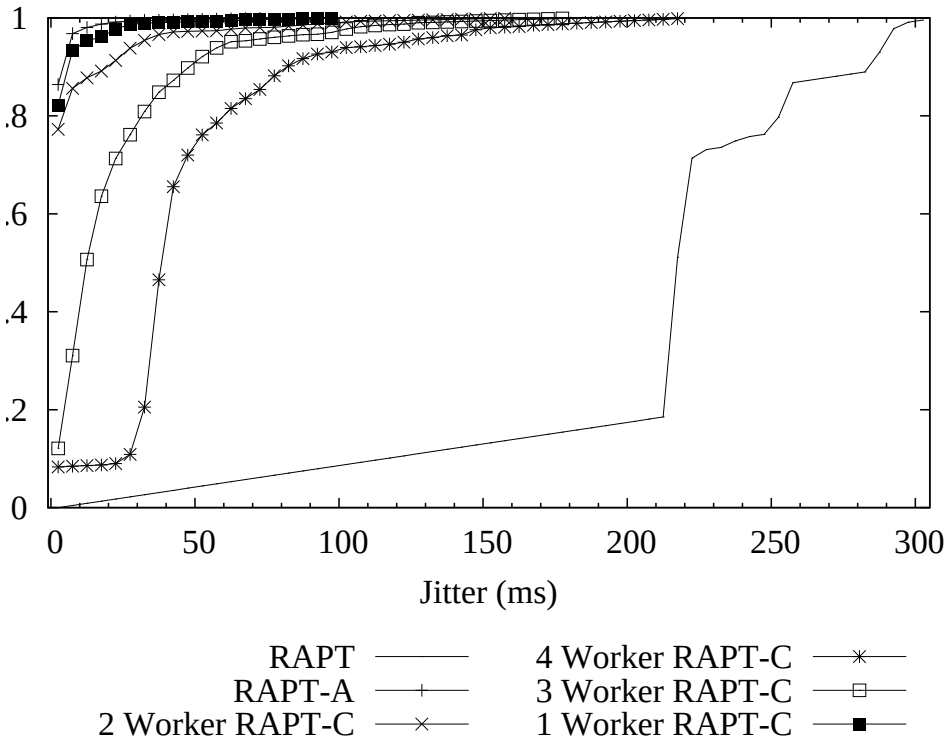


Figure 3.9: Jitter cumulative distribution in the hard Scenario

Less Cores Than Workers

To test what will happen if we have more workers than cores, we ran the medium scenario in a single core machine (a 2.80GHz Intel(R) Pentium 4).[2] As we see in

[2] All of the other experiments reported in this thesis were performed on a multicore machine.

Table 3.5: Jitter profile in the easy scenario (with one core)

Jitter (ms)	RAPT	RAPT-A	RAPT-C
Median	25	5	17
Tail	28	8	28

Table 3.5, RAPT-C has lower FPS and higher jitter tail than RAPT-A (but comparable quality to RAPT). We also noticed in the low-level event-loop statistics that RAPT-C submitted less events and canceled more. This performance gap is due to the overhead of communication between and scheduling of the worker threads and the lack of any parallel speed-up using the one core machine.

Ideally we should have one worker per core. Degradation in performance is expected if we use more or less workers than necessary. To help applications choose the appropriate number of workers, browsers can have an API to expose the number of cores (user agent information) or JavaScript library developers can detect it (using micro tests).

Results Summary

DOHA demonstrates the potential to help scale quality linearly as we use more cores in a challenging game scenario. It is essential to choose the appropriate number of workers for the execution and communication load and the underlying hardware. Using more workers than needed (4 workers case in Figure 3.9) or using less workers than cores (Table 3.5) can reduce performance due to concurrency overheads (without getting any parallel speed-up).

3.3 Lessons Learned

This section includes few of the subjective lessons learned which can shed more light on DOHA and Web-based game development. We noticed that:

- Performance engineering inside browsers is challenging. Browsers have primitive debugging and performance monitoring tools. Web workers have even less support. To conduct a rigorous experimental study and quantify performance, we had to build a number of performance analysis tools.

- Game adaptation in RAPT required minimal code changes. To introduce priority and cancellation of events we only had to change the core game loop and add the policies as described in Section 3.1.1. The changes affected 1% of the code base (including comments, counted using wc -l). Using the central scheduler was relatively easy having already used explicit events because most of the distribution tasks are delegated to the central scheduler. However, using Web workers with a distributed application design requires re-structuring existing code in a major way. For example, RAPT was modified to have a distributed game loop and we used replication to manage shared state as described in Section 3.1.2. Introducing the publish-subscribe layer improved the abstraction but developers still need to write complex distributed algorithms.
- Parallel and adaptive execution are independent and can be adopted separately. Applications can use DOHA's publish-subscribe communication layer for state management without adopting explicit events and the other way around. Both approaches are introduced together within the framework of scalable quality because adaptation scales application demands (up and down) based on the limited available resources while concurrency expands the pool of available resources allowing scalable parallelism. RAPT-C uses both adaptive and concurrent execution to get better performance when CPU demands exceed the resources available in one core. However, RAPT-A avoids the development costs and the concurrency run-time overheads of RAPT-C (Figure 3.9) making it appealing when the application can provide adequate quality with one core. Developers need to examine the advantages in quality improvements versus the run-time overheads (and development costs) when deciding what is the level of scalability (e.g., RAPT-A, RAPT-C with 2 workers, or RAPT-C with 3 workers) most appropriate for the range of target platforms.
- HTML5 Web workers expose an elegant shared-nothing concurrency abstraction. Explicit message-passing is a good fit for asynchronous event-driven browser execution. It also allows Web developers to use familiar distributed computing abstractions (from their experience with server compo-

nents). The main limitation is the high communication costs in implementations. Current communication cost in browsers will enable client-side applications to scale quality in multi-core platforms with few cores (e.g., 4 cores in RAPT – 3 workers and the main browser thread). To scale in many-core platforms with tens of cores, the communication costs needs to be reduced significantly. Browsers need to optimize the communication channels and expose optimization mechanisms (e.g., use immutable objects with ownership transfer to pass large objects across workers). JavaScript frameworks similar to DOHA can also perform communication optimization (batching and pipelining) to reduce messaging costs.

- DOHA is applicable in other Web multimedia applications, such as video applications, visualizations, and animations. We observed the same event-driven architecture in the few animation and visualization platforms we studied. To extend our support to server-side game components, we ported DOHA to node.js [35], a popular JavaScript server framework. Our future work aims to use scalable quality in other application domains and perform an in-depth study of the adaptation and load-balancing policies required.

3.4 Related Work

DOHA builds upon the event-driven nature of popular Web applications, which have a large number of short handler functions [59]. Event-driven programming is a natural fit for JavaScript, a single threaded programming language, supported by the asynchronous browser APIs. Event-driven reactivity in DOHA has its roots in the concepts of reactive programming [5]. DOHA introduces execution event classes and specifies timing information at the event level similar to the application model in Cooperative Polling [41]. Explicit execution events are used for all computations and the underlying scheduler adheres to hints in events to run important computations in a timely fashion.

DOHA developed CPU adaptation policies for Web-based games inspired by Priority-Progress adaptation [37, 39]. DonneyBrook's [10] interest sets use distance, aim, and recency from the player's perspective to decide which entities are more important. Similar to interest sets, our CPU adaptation policy uses distance

from the player to determine the importance of game entities, but our priority scheme has a continuous spectrum (between 0.0 and 1.0) allowing smooth scalability instead of the two priority levels in DonneyBrook.

Recently, developers used workers to separate the physics engine of a simple animation [43] improving the animation's frame rate. However, offloading functional units to workers will limit scalability to the number of independent units within an application. DOHA helps developers address challenging concurrency issues, such as state management and load-balancing in order to provide scalable parallelism with more cores.

Parallel architectures in games use techniques, such as Synchronization via scheduling (SvS) [7] and Software transactional memory (STM) [46] to manage state. SvS uses results of the static and dynamic code analysis to manage potential shared state conflicts by exposing the data access pattern to the scheduler. Lupei et al. [46] show that STM can provide better performance than the state of the art multi-threaded lock-based game server. Even though these techniques are suitable for parallel games, they assume shared memory while Web workers have no sharing and use message-passing. The Multikernel [4] investigates a new OS structure that treats a machine as network of independent cores with no shared memory and move traditional OS functionality to a distributed system of processes that communicate using message-passing. Similar to the Multikernel, we embraced the network nature of concurrent systems and re-structured our experimental Web-based game as a network of distributed components. We use replication to share state using ideas from the distributed architecture of interactive multi-player games in Colyseus [9].

Coloring [82] was introduced as a coarse grain concurrency management technique for event-driven Web servers. Events with the same color execute in the same worker while events with different colors can execute in parallel. Coloring was used to manage concurrency in the central scheduler. To improve the browser performance, the parallel browser project [47] re-writes the bottlenecks (parsing and rule matching) in a parallel fashion. Application-level concurrency (in JavaScript) is equally important especially since applications are faster to change and adapt than browsers.

Native Client [81] allows Web applications to execute native code inside a

browser sandbox and improve performance (by using hand-coded assembler and native threads). DOHA aims to improve the performance of applications written in JavaScript, the de-facto language for Web applications. Using native code in the browser is complementary to our work especially since JavaScript engines are becoming more mature. The exokernel browser architecture in Atlantis [48] defines a narrow API for basic services and allows Web applications to extend their execution environments. Atlantis' run-time language Syphon supports a full threading model. Even though the performance of threads with shared memory is arguably superior to Web workers with message-passing, the performance gains come at the high cost of introducing a concurrency model that causes most system errors [63].

3.5 Conclusions

Browsers are becoming mature platforms. Ambitious Web applications with high computational demands and low latency interactions, such as games, animations, and interactive visualizations are pushing the limits of available processing resources. In overload conditions, the best-effort execution model of current browsers lacks the necessary mechanisms to help these demanding applications control quality and balance between timeliness and utilization. In addition, ambitious multimedia applications need more processing power than available in one core especially in mobile platforms with low-end cores. Even though HTML5 Web workers provide a concurrency model to utilize multi-core resources, Web developers still need more programming support in hard concurrent software development issues, such as state management, load balancing, and timely execution with multiple threads.

DOHA deals with the volatility and shortage of processing resources based on Priority-Progress quality adaptation. Scalable quality in DOHA addresses resource volatility by enabling applications to scale demands based on available resources (including multi-core) and to utilize the limited available resources to execute important computations with more influence over perceived quality. Our evaluation shows that when CPU resources are scarce, the modified game using DOHA had better timing and higher overall quality. More importantly, the quality scales linearly with a small number of cores. Scalable quality enables ambitious Web applications to explore more challenging scenarios without the fear of brittle quality.

Chapter 4

Transport Layer: Paceline

Beyond interactive Web applications as addressed by Doha in Chapter 3, another area in which application adaptation is critical is streaming delivery and display of multimedia content over the network. Because the data of many applications and users are multiplexed over a shared common media, network performance is highly variable both over the short-term (millisecond or second granularity) and longer term (10s of seconds or minutes granularity). The primary motivation of our work is to enable adaptation in applications with high bandwidth (hundreds of Kbps or more) and latency sensitive (tenths of a second or less) network communication. Our example application is HD video conferencing which is part of a growing real-time collaboration market. Other multimedia Web applications in this class are large scale high speed online multiplayer games [10], and online virtual worlds [2]. In these applications the volume of data is large (i.e., HD multimedia with a large number of participants). In addition, there are stringent interactivity requirements so applications need to keep end-to-end latency down at all times, for effective response and comfortable communication. These demanding applications simultaneously require high bandwidth and low end-to-end latency, a conflicting combination that is poorly supported in existing best-effort transports.

Paceline introduces adaptation mechanisms as essential transport primitives to resolve the conflict between timeliness and best-effort transports. Paceline enhances the transport service model with mechanisms for quality adaptation. Our quality-adaptation model is based on Priority-Progress adaptation [37], which re-

mains the most stable adaptation technique over the Transmission Control Protocol (TCP) in terms of packet delay and jitter [42]. Priority-Progress adapts quality based on a timeline that specifies when each message is relevant. Priority-Progress uses three principles:

- *Incremental Quality:* The video conferencing application used in the evaluation supports spatial and temporal scalability.
- *Prioritized Data:* Paceline introduces message prioritization to provide timely delivery for important data with more influence over quality. Priority is assigned by an application policy using spatial and temporal indicators of perceived quality.
- *Priority Data Drop:* Paceline introduces message cancellation to adapt the application rate to match available bandwidth. Messages are canceled when they becomes stale according to the application's timeline.

Using Priority-Progress adaptation mechanisms, Paceline enables applications to scale quality with available resources and to use the limited available bandwidth in transferring data with more influence over quality. Up to this point, we are considering adaptation in a single stream of messages to the varying availability of network resources. Paceline also enables Priority-Progress adaptation across multiple high-bandwidth low-latency streams in a fair fashion. Different streams have varying requirements in terms of latency, bandwidth, and high-level application quality metrics. For example, a game transfers several kinds of streams, such as player status updates, player video coordination chats, advertisements, and game control messages. The frequency of advertisements might be relaxed if necessary to help ensure player updates are sent promptly. Similarly, a distance learning session can have voice, video, and slides from different users as separate streams multiplexed over the same communication channel and have different quality metrics. For fair and timely communication across concurrent streams, Paceline supports for two notions of fairness across streams sharing a link: quality and resource fairness. Resource fairness guarantees fair bandwidth across streams, while quality fairness ensures fair application level quality. Quality is defined at the application level as the frames per second in video conferencing, or the updates per second in online

games. Our service model defines a generic notion of quality to suit any type of continuous communication.

Contrary to conventional wisdom, Paceline has not been implemented over the User Datagram Protocol (UDP), nor does Paceline propose changes to TCP. Paceline is layered *entirely* above TCP, the "narrow waist" of the Internet. TCP constitutes the dominant component of Internet traffic volume, typically greater than 90% [25]. The shortcomings of TCP with regard to end-to-end latency are well known with full Internet standards (IETF) (i.e., Stream Control Transmission Protocol *SCTP* [56] and Datagram Congestion Control Protocol *DCCP* [36]) and mature research transports (Structured Stream Transport *SST* [24]) proposing to modify or replace TCP. However, the basic characteristic of TCP has not changed and no alternative transport has yet to gain any appreciable adoption. In this chapter, we explore the idea of improving latency without replacing or even modifying TCP. To address TCP's latency problems and minimize the end-to-end latency for important data, we use the three following techniques:

- Application-level rate control, to reduce queuing delay due to excessive socket buffering.
- Failover among connections to handle extreme cases of congestion leading to latency spikes.
- Application data unit (ADU) fragmentation to prevent head of line blocking and reduce the granularity of pre-empting less important data.

The contribution of Paceline is in the combination of the above techniques, in a way that mitigates TCP's weaknesses. Even though the underlying service model is best effort, Paceline's techniques improve the transport agility to ensure that the most important data have good timeliness. Our evaluation shows that Paceline improves upon conventional end-to-end latency shortcomings of using TCP, by a factor of 3 in median latency and a factor of 4 in worst case latency. Meanwhile, Paceline is able to preserve TCP's performance in terms of fairness and utilization. We also compare Paceline with the Structured Stream Transport (SST) [24]. SST is one in the class of several protocols designed to provide better control over timing than TCP, such as SCTP and DCCP. SST is the most recent, it has comparable features and builds upon many ideas common to the others, we chose SST

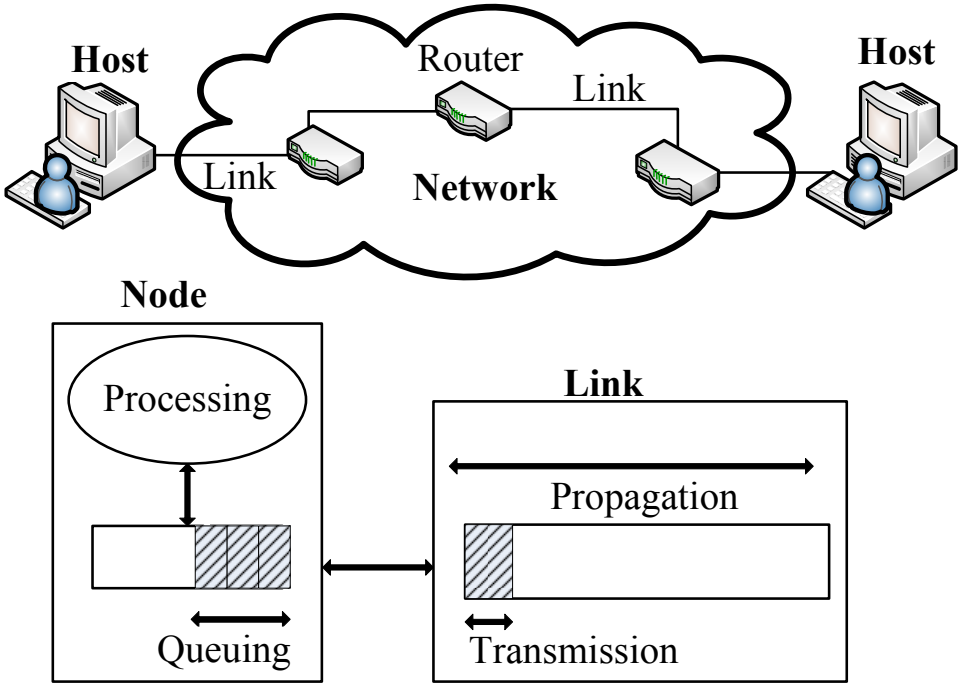


Figure 4.1: Components of end-to-end latency

for our evaluation because we believe it represents the current state of the art. Using application level metrics from a video streaming system, our evaluation shows Paceline's performance to be very competitive with that of SST.

The rest of the chapter is organized as follows. Section 4.1 provides an end-to-end analysis of TCP delays. Section 4.2 describes the transport service model and Section 4.3 explains the architecture of the low-latency techniques. Section 4.4 presents the evaluation results. Finally, Section 4.5 summarizes the related work and Section 4.6 concludes.

4.1 End to End Latency Analysis

Since interactivity and transport latency are a key focus of this work, we now briefly characterize the sources of latency and set the context for our approach in Paceline. As depicted in Figure 4.1, end-to-end latency is commonly broken down into four components of 1) *processing* delay, due to processing speed, 2) *queuing* delays in

nodes (hosts and network routers and switches), 3) *transmission* delay due to the bit-rate of transmission, and 4) *propagation* delays due to physical distances. When one or more of those delays becomes large, interactivity (application to application message delivery) will suffer. As we will show later in Section 4.4, the total end-to-end latency of TCP can be several seconds. In the following analysis, we explain that of the four latency components, *queuing delay* (inside TCP send buffers and network node queues) is the dominant cause of latency for high bandwidth TCP applications.

Firstly, *processing delay* is generally negligible due to fast CPUs and careful design of transport algorithms. Second, if we assume for the moment that application data units (ADUs) fit within transport segments up to a maximum size (*mss*), then *transmission delay* will be bounded to $delay_{transmit} = mss/link\_rate$. With common values of $link\_rate$ (Mbps or Gbps) and *mss* (e.g., 1500B), $delay_{transmit}$ will be a small value (e.g., sub-millisecond). This leaves propagation delay and queuing delays as the dominant contributors to latency. One-way *propagation delay* has lower bounds set by the laws of physics. Typical Internet path round-trip-time (RTT) values are in tens of milliseconds for intra-continental distances, or around one or two hundred milliseconds for distances that cross oceans or traverse satellites. In addition, TCP provides reliability via retransmissions that can add extra queuing delay (multiples of the propagation delay) to the total. For this reason, TCP is commonly dismissed as unsuited for latency sensitive applications. However, in the common case, TCP's fast retransmit mechanism should limit the retransmission-induced queuing delay to an RTT or two. Only in the case of very congested networks will back to back retransmission timeouts lead to a series of exponential backoffs which degrade TCP's performance. On the other hand, TCP's socket buffer is often large enough that it can cause queuing delays in the order of seconds. In a previous study, it was shown that in many realistic conditions, the queuing delay (specifically due to the send side TCP socket buffer) is the dominant portion of the total delay because of large kernel socket buffers employed by TCP implementations [28].[1] To address this, that study proposed and implemented a modification in the Linux kernel to dynamically tune the socket buffer size to

[1]For example, with a typical TCP send buffer size of 64KB, and a 300 Kbps video stream, a full send buffer contributes 1700 ms of delay.

avoid unnecessary queuing delay, bringing the end-to-end delay within 2 RTTs most of the time, while leaving TCP's congestion control unchanged. In this paper we build upon that work, but our current solution is designed to avoid altogether the need for kernel modifications.

A kernel approach has several limitations that motivate us to investigate user-space solutions. First, the need to modify the kernel is a fairly major obstacle to deployment, as new implementations of TCP can take years to percolate widely to existing systems. Second, under unusual duress of back to back losses and retransmission timeouts, TCP can effectively become stalled. To address this challenge, we enhance our transport service with a failover mechanism external to TCP. Third, in circumstances where transparent proxies are employed (by ISPs, CDNs, etc.), splitting the end-to-end flow across multiple TCP connections, an in-TCP based approach looses effectiveness. For example, if the TCP connection to a proxy (first hop) is local, and the proxy to server connection (second hop) spans a slow wide-area path, then the first connection will mistakenly send at a fast rate and allow the proxy to queue up an undesirable amount of data. Considering these issues together, we were inspired to explore a user-level approach.

4.2 Data Service Model

Paceline provides a transport service targeted to applications with both tight timing and high bandwidth requirements. These applications are increasingly designed to support diverse environments from gigabit broadband networks to congested wireless links. Network resources (e.g., bandwidth) can be saturated either due to variability in an application's own demands (applications can have different resource requirement over time) or variability in resources (when sharing resources with other applications). Adaptive applications fine-tune the quality of their outcome depending on the available resources. This section describes the service model integrating Priority-Progress adaptation mechanisms to support target applications.

In contrast to the reliable byte-stream service model of TCP, Paceline provides a reliable message-based service model. Like TCP, the Paceline service is full-duplex, but for simplicity of presentation, we will describe the two endpoint applications and Paceline in terms of application sender or receiver roles. We chose a message based model because low-latency is a primary goal, and messages provide

a natural explicit means for the application to inform the transport about latency preferences as well as representing an application data unit (ADU).

Paceline's programming interface allows the application to specify message importance on a per-message basis, and Paceline delivers messages in order of importance, which is not necessarily first-in first-out (FIFO).[2] The ability to queue messages ahead of time is essential to achieve high bandwidth, but the ability to prioritize messages is necessary to prevent head-of-line blocking[3] and the attending loss of responsiveness. Since re-ordering is implicit in Paceline's model, the message send primitive provides an option (per-message) for the sender to be notified when the message has been delivered to the receiver. Unlike the byte-stream service model, Paceline allows the sender to cancel a pending message, this feature is motivated by the goal of responsiveness because the old data will slow down new messages and waste bandwidth. At the receiver, Paceline passes messages directly to the application. Applications need to handle out-of-order delivery and missing data introduced by message priority and cancellation.

In conjunction with congestion control, cancellation is used by the application to adapt the rate of message delivery to the underlying network conditions — such adaptation is an essential requirement to reconcile the inherent conflict between the application's need for control over timing and the best effort nature of Internet service. Informed cancellation maintains reliable delivery semantics while allowing applications to cancel stale messages. This provides an alternative to random dropping of messages (e.g., UDP) under congestion.

Paceline's delivery service model comprises two request primitives and two notification primitives (callbacks). The requests are *msg_write* and *msg_cancel*, and the notifications are *sent*, and *recv*. One notable absentee from this model is any explicit notion of time, deadline, expiration, *etc.*. Given Pacelines's objective of supporting low-latency applications, we could have expanded the model to have Paceline assist directly in expiring messages deemed too late for useful delivery (e.g., as in SCTP's partial reliability option). However, we rejected such an approach as the cancel primitive is sufficient for the application to expire messages itself (as in the example below), and it provides a means for application specific

[2] Paceline does enforce FIFO ordering among messages of equal importance.

[3] A delay that occurs when a line of packets is held-up by the first packet.

```
send_video_frame (player, stream, frame) {

  /* Set message data and length */
  msg_init.data           = frame.data;
  msg_init.length         = frame.data_len;

  /* Set message importance */
  msg_init.importance        = server_get_utility(frame);
  msg_init.marginal_utility  = server_get_marginal_utility(frame);
  msg_init.sent              = video_frame_sent;

  /* Sending a frame with cancellation */
  stream.msg_create(msg_init, &frame.msg_handle);
  stream.msg_write(frame.msg_handle);
  frame.expire_event = expire_video_frame;
  add_timer(frame.deadline,
            frame.expire_event);
}
expire_video_frame (frame, stream) {
  stream.msg_cancel(frame.msg_handle);
}
video_frame_sent (player, frame) {
  cancel_timer(player,frame.expire_event);
}
```

Figure 4.2: Adaptive video conferencing client

canceling policies. Furthermore, an expiry mechanism would not subsume the need for a cancel primitive, because there would still be *sporadic* events (such as user initiated seek to a new position in video streaming) that require the ability to cancel immediately.

To help illustrate Paceline's service model, Figure 4.2 contains a pseudo-code example of the logic that an adaptive real-time application might employ, in this case an adaptive video conferencing client. The client calls the *send_video_frame* function to send a video frame message. This function sends the message with an importance specified using an application-specific utility measure, reflecting the relative importance of individual frames to perceived quality. If congestion control restricts the rate of the stream, messages of low importance will be canceled by the client when their utility has expired while high importance messages will be sent.[4]

[4]For messages of equal utility, Paceline breaks the tie according to position.

Paceline's service model provides a clean interface for rate adaptation to match application demands with network conditions, instead of committing messages to the network transport (i.e., socket buffer) and then suffering from head of line blocking.

To benefit from Paceline's data service model, applications have to develop domain-specific adaptation policies. In HD video conferencing, we have two dimensions for adaptation: spatial and temporal quality. For each application data unit (ADU), the application calculates a utility value to estimate its contribution to the video quality. Each ADU represents a video layer and uses a Paceline message. Our example application, QStream [39] video conferencing, adaptation policies favor maintaining temporal quality (frame rate per second) over spatial quality (e.g., peak signal-to-noise ratio PSNR) to have less interruption even if the spatial quality per frame is minimal. In addition to these two quality dimensions, we can incorporate higher-level indicators, such as the active tab, mouse clicks, and position of scroll bar to derive our adaptation policies.

In the gaming domain, first person shooter (FPS) games have limited upload bandwidth to send frequent updates to all game players especially in epic fights with a large number of players that are concentrated in one area. Therefore, recent research [10] has introduced the idea of interest sets to leverage limits of human cognition in reducing bandwidth requirements. Interest sets are measured using three criteria: proximity, recency, and aim. *Proximity* is important because players are most likely interested in players near them. However, proximity is not the only spatial locality indicator because players have an orientation. They are more likely to be interested in players they are aiming at. *Recency* indicates temporal locality so players who have recently interacted are more likely to pay attention to each other. Interest sets can be used to derive the adaptation policies in epic-scale first person shooter games to adapt bandwidth requirements[5] and maintain timeliness.

Each application message sent in Paceline is part of a full-duplex transport instance we refer to as a stream. Multimedia applications using Paceline can perform the following operations on streams: creation, sending a message, canceling a message, and deletion. Even though streams are decoupled from the underlying

[5]Each peer receives 10 Mb/s in a 900-player game [10]

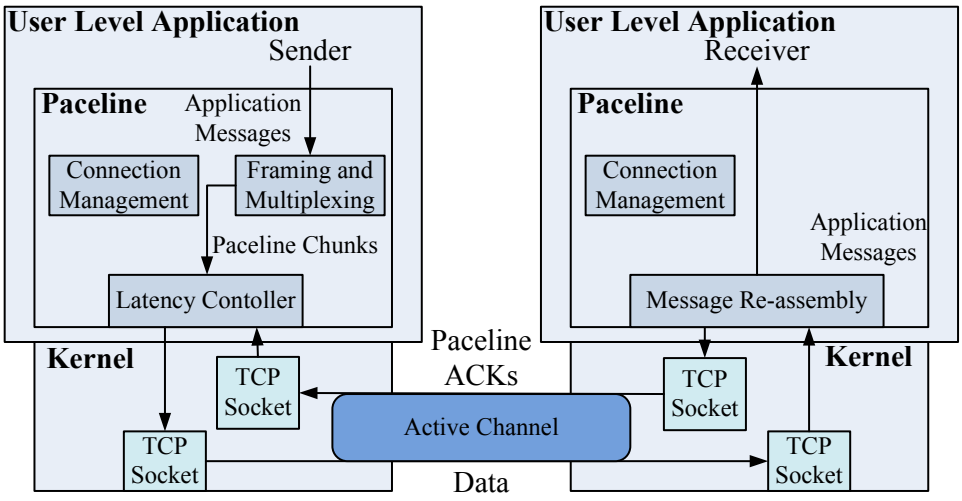


Figure 4.3: Paceline architecture

communication channels, all streams with the same host address and port number are multiplexed over the same persistent channel. *Channels* are the underlying communication primitive, identified by the host address and port number.

To summarize, Paceline's service model provides applications with the ability to prioritize and cancel messages. Paceline messages are part of a stream. Interactive applications with high bandwidth requirements can use these primitives to develop domain-specific adaptation policies to maintain timeliness in best effort networks (i.e., the Internet).

4.3 Architecture

Paceline is implemented as a user-level library and is layered above standard TCP implementations as depicted in Figure 4.3. Paceline's architecture consists of four main subsystems: message framing and multiplexing, a latency controller, connection management, and a stream layer. The stream layer is introduced at Section 4.3.4. We describe the subsystems in the remainder of this section.

4.3.1 Framing and Multiplexing

Fragmentation is the first of several techniques employed by Paceline to improve transport latency. Paceline allows application level messages of arbitrary size. To decouple transmission delay of potentially large application messages from lower

level queuing delays, the data transfer mechanism of Paceline supports sender-side fragmentation of application messages into Paceline chunks, and receiver-side reassembly of chunks back into the original application messages. Paceline chunks are bounded to a small size, typically a fraction of TCP's maximum segment size (MSS). Paceline includes application level message queues. Unlike lower level queues that operate in FIFO order, Paceline's message queues are priority queues, so that chunks of newly-arrived important messages may quickly preempt older less-important ones.[6] Therefore, chunks of messages with high importance are released to the network faster and observe minimal queuing inside Paceline as well as minimum application level transmission delay. Cancellation allows the application to abort a low importance message if its overall transmission delay is too large.

4.3.2 Latency Controller

In order to give applications more agility in adapting data delivery, Paceline reduces the amount of committed data in TCP's outgoing buffer and keeps data in its own message queues. The latency controller is the component that monitors the progress of the underlying TCP flow and regulates the rate of application data (chunks) delivered to the sending side TCP. The goal of this controller is to send chunks into TCP fast enough to allow the congestion control of TCP below to claim the flow's fair share of available bandwidth, but not so fast as to cause unnecessary amount of FIFO queuing to accumulate in TCP's outgoing socket buffer. We have devised two distinct schemes: kernel-assisted and purely user-level approach, each having specific advantages.

Kernel-Assisted

The first scheme, which we call the PaceK controller, utilizes information directly from the kernel TCP via the socket API. PaceK regulates the writing of application data to TCP in a way that dynamically matches the buffer fill level to a value close to the size of TCP's congestion window (cwnd), namely $cwnd + 3 \times MSS$. This design implements at user level the same strategy that was implemented inside

[6] SCTP provides similar support to avoid head of line blocking, but there the focus is blocking between sessions rather than individual messages.

the kernel in a previous study [28]. In that previous study, it was shown through extensive experimental evaluation that this strategy strikes the best balance between latency and throughput.

While this scheme is simple and effective, it requires information that only some implementations of TCP make available. The PaceK controller we have implemented in Paceline is Linux based, and it uses the Linux specific TCP_INFO getsockopt to query TCP's congestion window (cwnd) size, and SIOCOUTQ ioctl to query the TCP socket buffer fill level. In Windows, the overlapped IO feature of the WINSOCK2 API can provide similar information. Hence, this approach is more portable than the previous modification to the Linux kernel. However, to the best of our knowledge, the TCP socket API's of other popular OS's such as MacOSX, Solaris, Symbian OS, and BSD Unix do not provide access to such information, hence our PaceK controller is not fully portable. Also, transparent proxies in the network path would likely defeat the PaceK controller's ability to regulate queuing delay, as the TCP socket buffers in the proxies operate independently, and can easily become points of major queuing delay if they precede the path bottleneck.

Purely User-Level Approach

The second latency controller available in Paceline is called PaceA. Unlike the PaceK controller, it uses only the common TCP socket API available on all major operating systems. PaceA is designed to be layered above TCP, which entails both advantages and obstacles not applicable in true transport level solutions. The main advantages are portability (no need to extend or modify kernels), ease of deployment (e.g., in relation to firewalls), and avoiding problems due to intermediate proxies.

As with the kernel-assisted method, the user-level latency controller regulates writing of new chunks to TCP, so as to keep the value of TCP buffer fill level close to $cwnd + 3 \times MSS$. In this way, the amount of FIFO queuing, which is the root cause of head of line delays, is minimized. However, at the user-level the value of *cwnd* is not available. Hence, the primary goal of PaceA is to derive $\widehat{cwnd}$, an estimate of TCP's *cwnd*, as accurately as possible using only information that

is available to the application. Throughout the development of our algorithm, we used time series traces from our prototype to compare our algorithm's estimates against TCP's true behavior. Section 4.4 describes the experimental setups under which these traces were taken. Using this experimental approach, we iteratively developed our latency controller. The rest of this section describes the derivation of our algorithm to accurately calculate $\widehat{cwnd}$ from estimates of network latency ($\widehat{rtt}$) and available bandwidth ($\widehat{bw}$).

Paceline utilizes application-level acknowledgments (P-ACKs) to measure latency and bandwidth as follows.[7] Paceline generates a unique sequence number for every Paceline chunk. When a chunk is written by Paceline to TCP, it enters that chunk into a FIFO queue (since TCP delivery is FIFO) and stores the current time with the chunk. A value *unacked* is maintained that totals the size of all chunks written but as yet unacknowledged, hence reflecting the TCP socket buffer fill level.[8] When reading chunks, the Paceline receiver will generate a P-ACK containing the sequence number of the last chunk received (often multiple chunks are received at a time). Upon receiving a P-ACK, the Paceline sender scans the queue for the chunk matching the sequence number. When found, that chunk and all prior chunks are de-queued and considered acknowledged (P-ACK'd). For each chunk P-ACK'd, the round-trip time is computed and the chunk size is counted for use in a periodic bandwidth calculation of $\widehat{bw}$.

TCP keeps CWND bytes in flight. The simplest form of $\widehat{cwnd}$ would be the product of latency and bandwidth. Being a user-level algorithm, Paceline can not directly know about TCP events such as retransmit, which leads to significant measurement noise in the latency and bandwidth values provided from P-ACKs. Therefore, we applied simple smoothing to the measurements of latency ($\widehat{rtt}_{ewma}$) and bandwidth ($\widehat{bw}_{ewma}$), using exponentially weighted averaging (EWMA).[9]

After implementing a naive controller based on $\widehat{cwnd} = \widehat{rtt}_{ewma} \times \widehat{bw}_{ewma}$ (Figure 4.4a), we observed that $\widehat{cwnd}$ is able to follow the general trend of the real value of *cwnd*, but lacks the fine details where TCP exhibits rapid changes. This slow

[7] P-ACKs are also essential to the failover component of Paceline to be discussed later in this section.

[8] We verified this using time-series traces of socket buffer data.

[9] $ewma(avg, sample, \text{alpha}) : avg = avg \times \text{alpha} + sample \times (1 - \text{alpha})$.

responsiveness of the controller impairs performance in two respects. First, when TCP drops its rate by resetting cwnd to the initial value (i.e., when a TCP retransmission timeout occurs), our controller does not respond immediately, causing the socket buffers to fill and latency to increase. Second, at stream start-up or when there is a major increase in available bandwidth (e.g., due to reduction of competing traffic), the controller's slow response impairs TCP's ability to claim available bandwidth.

To improve the controller, we devised a hybrid estimate that treats TCP's decrease and increase modes separately. To identify TCP's mode of operation, we use an indicator of the trend in TCP rate which we call *pressure*:

$$pressure = \frac{\widehat{bw}}{2 \times \widehat{bw}_{ewma}}$$

Pressure normalizes the ratio between $\widehat{bw}$ (the short term estimate of bandwidth) and $\widehat{bw}_{ewma}$ (a long term estimate of bandwidth). If necessary the value of pressure is clipped, so it fits in the range $[0,1]$. During rapid decreases in bandwidth, pressure drops to zero, while during rapid increases it rises to one.

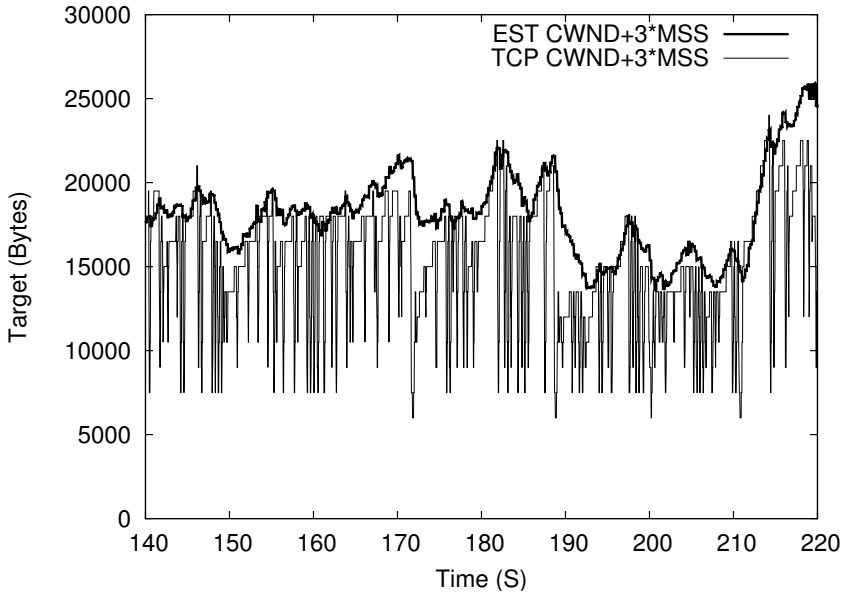
Sudden drops or increases in bandwidth measurements can be early signals of congestion or available bandwidth. This is similar to TCP Vegas congestion control [13], which uses increases in RTT measurements as a signal of queue buildup.

To treat TCP increases separately from decreases, we replace $\widehat{bw}_{ewma}$ with two separate terms $\widehat{bw}_{+}$ and $\widehat{bw}_{-}$. The $\widehat{bw}_{+}$ term responds immediately to increases in $\widehat{bw}$ but smooths decreases, using asymmetric EWMA:

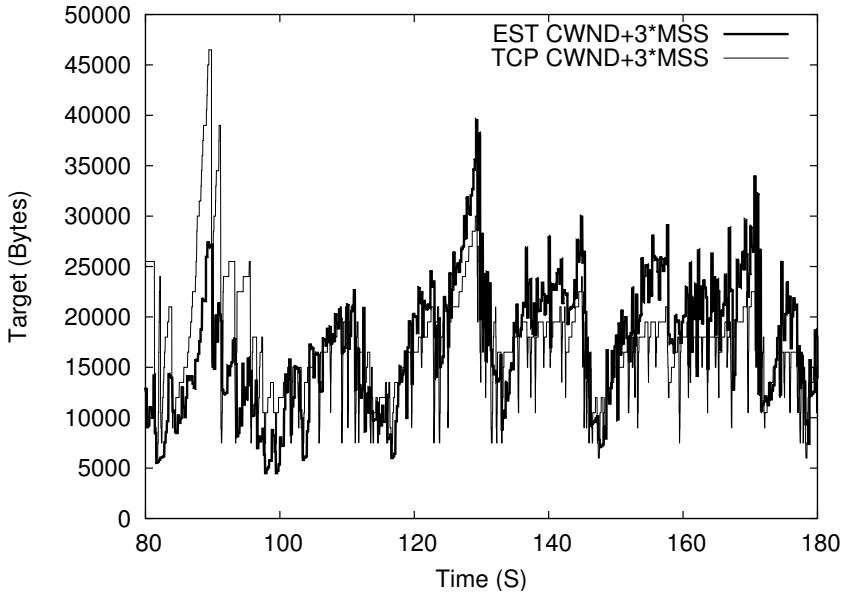
```
ewma(avg,sample,α):
  if(sample > avg)
    avg = sample
  else
    avg = ewma(avg,sample,α)
```

The objective is to favor $\widehat{bw}_{+}$ when TCP's rate is increasing, as *pressure* approaches one. $\widehat{bw}_{-}$ works in precisely the opposite fashion to $\widehat{bw}_{+}$, and is favored as pressure approaches zero. Thus, our hybrid estimate is as follows:

$$\widehat{cwnd} = rtt_{ewma} * ((1 - pressure) * \widehat{bw}_{-} + pressure * bw_{+})$$



(a) Naive estimation



(b) Hybrid estimation

Figure 4.4: User-level estimation of CWND compared to instantaneous values obtained from kernel

This estimate succeeded in responding more accurately to the true TCP *cwnd* under most network conditions (Figure 4.4b), however, under very heavy load we observed it still overshoots. In these conditions, the relatively small adjustments to the rate can have large effects on queuing delay. This causes a positive feedback loop, where a small overestimate of BW leads to an increase in rtt_{ewma}, which results in further over estimation, manifesting as a persistently rising rtt_{ewma}. To detect this situation, we introduce a new stability check in the algorithm, it uses the ratio of rtt_{ewma} to rtt_{-}, where rtt_{-} is based on asymmetric filtering of rtt. The rtt_{-} responds immediately to decreases in latency, smoothing otherwise. From our traces, we found it to be representative of the minimum transport latency defined by TCP's RTT measurement. We expect that a heavily smoothed average of rtt_{ewma} should remain close to the average TCP RTT. We take a ratio of the two to be a conservative sign that a correction is necessary.

Finalizing our estimation of *cwnd*, we have:

$$
\begin{aligned}
&\texttt{if } \tfrac{rtt_{ewma}}{rtt_{-}} \geq 2 \\
&\qquad rtt_{signal} = rtt_{-} \\
&\texttt{else} \\
&\qquad rtt_{signal} = rtt_{ewma} \\
&\widehat{cwnd} = (1 - pressure) \times rtt_{signal} \times bw_{-} + \\
&\qquad pressure \times rtt_{ewma} \times bw_{+}
\end{aligned}
$$

Encouraging Fast Retransmit

One of the fundamental components of TCP congestion control is the fast-retransmit algorithm, which normally allows TCP to recover from a lost TCP segment within a single extra RTT, instead of waiting for the retransmission timeout. Retransmission timeouts are highly undesirable as they cause an application to experience a dead zone, typically hundreds of milliseconds with no data transfer. Fast retransmit requires four segments to generate the necessary duplicate ACKs. If the application sends less than 4 $\times$ MSS bytes at a time, it may impair TCP fast retransmit. Paceline includes heuristics to prevent this, *i.e.,* it promotes bursting of at least four segments at a time. In some cases, if the available data is less than 4 MSS's,

Paceline interleaves sends with calls to the TCP_NODELAY socket option, to encourage TCP to generate four sub-MSS segments.

Implementation of P-ACKs

As described above, the receiver notifies the sender of the most recently received sequence number via P-ACKs. We constrain this process so that the total P-ACK bandwidth remains a modest fraction of the sender to receiver bandwidth (between 30 and 50Kbps in our implementation).

It should be possible to eliminate P-ACKs entirely in the PaceK controller, based on knowledge of the amount of data written and the socket fill level to infer which data has been ACK'd by TCP.

4.3.3 Failover and Connection Management

In Paceline, the message framing and latency controller components are the basic means to limit the latency experienced by the application, and we evaluate their effectiveness in Section 4.4.1. However, we foreshadow the results of our evaluation here in order to motivate Paceline's failover component. Briefly, we see that message fragmentation and application pacing can generally improve latency (in some cases more than a factor of three), but the distribution of latencies retains a prominent tail and there is a wide gap (e.g., more than a factor of eight) between median and worst case latencies. We diagnosed the worst case latencies through a combination of instrumentation in Paceline and packet trace analysis using tcpdump [70] and Wireshark [73]. Under heavy congestion, TCP can experience back to back losses leading to one or more retransmission timeouts. Our diagnosis confirmed the worst case latencies were correlated with such episodes of exponential backoff.[10] To reduce their impact, Paceline further includes a failover mechanism to supplement its basic latency limiting mechanisms.

One can think of Paceline's failover analogously to the scenario where a user presses the stop/reload buttons in their Web browser upon encountering slow response. Automated failover may sound quite radical, but our evaluation shows that

[10] Although it is outside the scope of this work, we have also noticed similar problems when testing Paceline over wireless links (WiFi) with poor signal strength.

our implementation achieves significant reductions in worst-case latencies while preserving bandwidth fairness (see Section 4.4.1). We already discussed further justification and rationale for failover in the related work analysis, for now we focus on the design of failover in Paceline.

Failover Implementation

Paceline's failover mechanism works by maintaining a pool of channels between the pair of communicating applications. Each channel contains two TCP sockets. Paceline uses just one of these channels for data delivery (in each direction) at a time. However, if a Paceline sender detects that a chunk has not been acknowledged within a time threshold, namely $failover_{thresh}$, it migrates data delivery away from the current active channel to one of the available standby channels. Paceline implements failover in a manner that is fully transparent to the application. The migration may include retransmission of chunks on the new active channel so Paceline introduces receiver-side logic to suppress duplicate chunks that may result. Also, concurrent to activation of the new active channel, Paceline's connection manager terminates the old one, and initiates a new replacement channel that, once established, enters the pool of available standbys.

The failover threshold is set dynamically in Paceline. The threshold setting is subject to a trade-off between latency and fairness. Since a replacement channel starts in TCP slow-start, frequent failover will inhibit Paceline's ability to attain a fair share of bandwidth, possibly resulting in under-utilization of the network. Paceline calculates the failover threshold as follows:

$$failover_{thresh} = MAX(threshold_{min}, rtt_{-} + failover_{factor} \times rtt_{var}) \qquad (4.1)$$

rtt_{var} is the maximum variance in round-trip times calculated for every Paceline chunk and $threshold_{min}$ sets a lower bound on $failover_{thresh}$, which we set to 225ms, similar to what is set as the minimum RTO for TCP connections inside the Linux kernel. We also intended to set $failover_{factor}$ to be the same as what is used in RTO, which is four. However, since our measurements are at the application level and subject to greater noise than measurements done inside the kernel, we observed that a factor of four results in too many false positives, i.e., Paceline

would failover even when TCP is not in an exponential backoff state. Thus adding a safety margin and using a $failover_{factor}$ of five significantly reduced the number of false positives.

When replacing a failed channel, the connection mini-protocol sends a *failover* message from the sender to the receiver on the newly selected active channel sockets to ensure both sides are synchronized. This message includes a counter value, maintained by the sender side, incremented each time the sender fails the current active channel. If very severe congestion causes more than one failover to occur back to back, then it is possible that standby channels will be established in an order different than how they were initiated. The counter is used to ensure that the sender and receiver remain correctly synchronized on the active channel.

When replacing a channel in this way, Paceline does not know if chunks that were in-flight on the failed channel were delivered or not. These chunks are returned to Paceline's sending queue, and possibly retransmitted on the new active channel. To maintain transparency, the Paceline receiver has to detect and suppress duplicate chunks. This is complicated somewhat by the fact that Paceline continuously sorts messages according to importance specified by the application. Thus, the sequence of chunks outstanding when failover occurs may not be equal to the initial sequence of chunks sent on the newly selected active channel. Some new high importance chunks may have arrived, and also some of the old low-priority chunks may be canceled (by the application). To cope with this, Paceline receiver maintains a lookup table containing sequence numbers of chunks received. When a chunk arrives, the table is used to detect duplicates. To prevent this table from growing unboundedly, the sender periodically sends a Paceline message to indicate the maximum and minimum sequence numbers active on the sender side. The receiver uses this information to clear out sequence numbers outside the given range. In this way, the set of sequence numbers in the receiver table will always be bounded, and all sequence numbers will eventually be removed from the table.

Connection Manager

The connection manager is responsible for managing the transport instances used by Paceline, including the standbys used for failover. For each *session*, meaning a

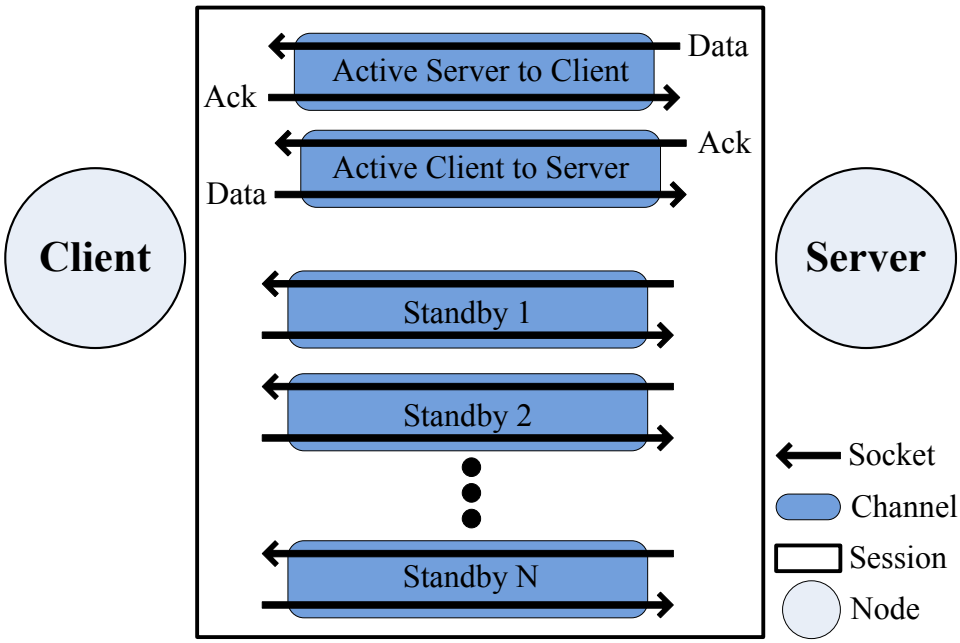


Figure 4.5: Paceline session

pair of processes communicating via Paceline, Paceline may employ several TCP sockets. The precise number of sockets and their roles depends on configuration settings, such as the number of standbys used for failover as depicted in Figure 4.5. Prior to the flow of application data, Paceline executes a mini-protocol to establish the initial set of session sockets. This protocol has two phases. The first phase exchanges a *configuration* greeting and response between client process and server process, using the first socket. The client greeting sets the session-wide configuration parameters, such as a globally unique identifier (UUID) for the Paceline session, and the number of standbys. The second phase of the protocol creates the remaining sockets, and exchanges a *bind* greeting and response on each of them. The bind greeting contains the UUID of the session allowing the server to associate the socket to the correct session state. After the initial set of sockets are established, the connection manager may also remove and replace failed sockets, as part of the failover functionality described above.

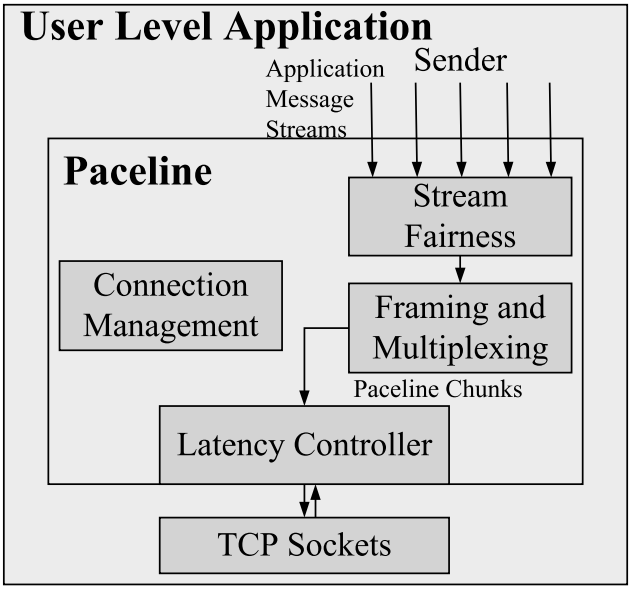


Figure 4.6: Paceline's architecture with stream fairness

4.3.4 Stream Fairness

Figure 4.6 illustrates the added stream fairness layer from the Paceline sender side view. The fairness design in this chapter is between streams sharing the same underlying communication channel. One important decision we had to make while implementing the stream fairness layer was to choose an appropriate policy to multiplex data of different streams over the underlying channel. While a simple FIFO or round-robin policy is simple to implement, timeliness necessitates a better notion of fairness among concurrent streams especially when bandwidth is limited. Thus, we implemented a fair sharing policy inspired by weighted fair queuing. In essence, our policy shares the available resources among *active* streams in a fair manner. An active stream is a stream that has data available to be written. If the send buffer of a stream is empty, we refer to that stream as *idle*. Each stream is assigned a *virtual time*, a counter quantifying the resources a stream has used since it was created. We use the term "time" due to the invariant that we never allow this counter to decrease, it can only increase.

The stream layer maintains a priority queue of active streams, sorted in ascending order of virtual times. Each time the channel is ready to write data to the underlying communication channel, it removes the stream with the lowest virtual time from the queue, writes a message from that stream and updates the stream's virtual time. If the stream is still active, i.e., still has more messages to be written, it is re-inserted in the queue. Otherwise, the stream is marked as idle and will re-enter the queue when it has more data available. Using this mechanism we can multiplex different streams at message-level granularity. The important factor regulating how streams are multiplexed is how we initialize and adjust the virtual times of the streams.

Virtual Time Initialization

Virtual time initialization is based on the following two rules:

- *Rule 1 (fair start):* when a stream is created its virtual time is set to the minimum virtual time of all the active streams. If no active streams exist, the virtual time of the newly entered stream is set to the maximum virtual time of all idle streams, or zero if this is the only stream.
- *Rule 2 (use it or loose it):* if a stream becomes active after being idle, the stream's virtual time is set to the maximum of its virtual time and the minimum virtual time of all active streams.

Rule 1 ensures that when some active streams have non-zero virtual times and a new stream X joins, we cannot set X's virtual time to zero, since it implies that all other streams would have to starve while X uses all the resources to catch up with the rest of the streams. If stream X is created when all other streams are idle, setting X's virtual time to the minimum is no longer fair. We set X's virtual time to the maximum virtual time of all idle streams. Now no stream has an advantage over the other because rule 2 guarantees that no stream can save its share of resources and use it at a future time. If stream Y goes idle for some time and all other streams use D units of resources, we adjust Y's virtual time once it reactivates to penalize it for not using its share.

Virtual Time Updates

In this section, we will give a clear definition of *fairness*. Fairness should have a definition that is independent from the type of application using the stream API. For example defining fairness between two video streams as an equal number of frames per second depends on the video application, and may not even be valid for multiplexing two videos with different encoding.[11]

Fair sharing is governed by how virtual time in each stream is updated. When a stream transmits a message over the underlying channel, its virtual time is updated according to the fairness criteria. We implement two fairness policies: resource fairness and quality fairness, using message size and marginal utility respectively. Conventional resource fairness is implemented by incrementing the virtual time based on the size of each message transmitted by the stream. On the other hand, quality fairness allocates resources based on the application's quality of experience. Quality fairness may cause un-fair resource allocation, for example, video quality depends on a number of factors such as temporal fidelity (frames per second) and spatial fidelity (image size or PSNR). Two streams with equal quality in those terms may have drastically different resource (bandwidth) requirements as we show in our evaluation.

To implement quality fairness consistently across concurrent streams, we propose a neutral model of quality based on a generic utility. Each application ADU will have a utility to the user, which can be expressed in normalized units (in the range [0,1], from the least acceptable quality to the maximum) between the application and the transport. Our service API exposes the message *marginal utility* value for this purpose. The virtual time of the stream is increased by the marginal utility of each message. The intuition is the following, since the application is continuous, we can think of the cumulative utility as the sum of instantaneous utilities. For the virtual time of each stream to be meaningful across heterogeneous streams, the marginal utilities should be computed such that $\int vt\,dt = t$, for the full sequence of messages having marginal utilities vt in a time interval t. If some messages are canceled, then virtual time will advance more slowly. Control messages are not adaptive so they use a default utility value of zero and importance of one.

[11] Sending one frame in an MPEG encoded video may be useless due to inter-frame dependencies.

By scaling virtual times of streams with different factors, we can allocate different shares to different streams, providing weighted fair sharing. In the next section, we evaluate fairness across streams.

4.4 Evaluation

We evaluated Paceline experimentally within an Emulab network testbed [80]. Our implementation of Paceline is written in C, the size of the Paceline codebase is around 10,000 lines (including comments, counted using `wc -l`). Our evaluation consists of three main sections: the first based on lower transport level metrics, and the second based on higher application level (multimedia) metrics. The first two sections evaluate adaptation in one stream while the third section evaluates fairness across concurrent streams. The low level evaluation concerns metrics of latency, fairness, and utilization in a range of network conditions. The high level part of our evaluation uses our video streaming framework, QStream [39], to relate the impact of low-level performance gains to metrics more applicable to the user experience. At the transport level, we use latency to mean the time to deliver chunks end-to-end with Paceline, and by fairness we mean the sharing of bandwidth among competing flows. At the application level, latency refers to the one-way delay tolerance of the application.

Our measurements are compared against two points of reference. The first is TCP, which we use to quantify the improvements due to Paceline. The second is the Structured Stream Transport (SST), which is implemented over UDP [24]. SST provides a rich service model including reliable messaging and congestion control, but unlike TCP it provides direct support for application priorities without head of line blocking. We chose it because we feel it includes the full range of capabilities one might expect from any realistic clean-slate replacement for TCP. Thus, we use SST to approximate a best-case reference point against which to compare Paceline.

4.4.1 Transport Level Performance

We compare Paceline (PaceA and PaceK modes) to TCP. In the TCP mode, our application still uses the service API of Paceline but the latency controller is disabled,

hence we send data via TCP as fast as it will allow.[12]

Our network setup uses the common dumbbell topology, where a set of servers on a LAN connect through a single bandwidth-delay constrained link, emulating a wide-area (WAN) path, to a set of clients on a remote LAN. For the WAN path, we emulate a 30ms RTT delay with a bandwidth limit of 16Mbps. The WAN bottleneck uses drop-tail queuing with a queue size of twice the bandwidth-delay product of the WAN adding 60ms when the bottleneck becomes congested. Each experimental run lasts more than 6 minutes. To control the amount of workload in each experiment, we vary the number of flows sharing the path. All of the nodes run the Ubuntu 9.04 Linux distribution with the Linux 2.6.28 kernel, and the default TCP Reno congestion control. We configure the number of clients and servers to ensure that the WAN path is the bottleneck (not some other client or server resource). To eliminate experimental start-up and shutdown effects, our measurements exclude the first and last 60 seconds of each run. Each experiment was executed 10 times and the average is reported. Confidence intervals were used to check the consistency of results but we do not report them due to lack of space.

Our first set of experiments focus on transport latency. Our aim is to quantify Paceline's low level latency improvements over plain TCP. In the remainder of this evaluation, our experiments are set up to reflect rather harsh conditions, where the bottleneck WAN link is persistently saturated. We focus on these conditions because we expect that if high-bandwidth low-latency applications, such as video conferencing, online games, and virtual reality applications become mainstream, they will make the network saturated, much as other high-bandwidth (but high-latency) applications such as peer to peer file sharing do now. These are the conditions where normal TCP's latency leaves much to be desired. On the other hand TCP's abilities to utilize the network and divide bandwidth fairly (in a decentralized fashion) while avoiding congestion collapse, have been critical to the ongoing success of the Internet.

[12] All of the flows are actually video flows [39] that cancels low-priority messages based on the flow rate. We mention this to confirm that the cancellation feature of Paceline is exercised in all our experiments.

Latency

To identify the settings at which TCP's latency begins to suffer, we streamed a variable number of video files over our testbed network. Each video flow has a constant maximum rate of 4Mbps. As described in Section 4.3.2, Paceline measures the application-level round-trip time for each chunk it transmits. Throughout the course of the experiment, the senders record a conservative measure of latency we call Oldest Un-Acked. The value is sampled periodically, as either the oldest outstanding chunk still in flight, or the maximum RTT measured (for each acknowledged chunk) in the interval, whichever is greater. This value provides a conservative estimate of the delay the application can experience for its most important data, although it explicitly excludes extra possible transmission delay due to large application messages. Through the remainder of this section, when we refer to latency, the measurement used is Oldest Un-Acked. We do not consider the latency of SST in this section, because its implementation eliminates transport queuing delay, however we will consider SST's performance and application level delays generally in Section 4.4.2.

Flows	2	4	8
Median Latency	1.6	7.1	7.4
99.9 Percentile	2.5	17.9	18.4
Worst Case	2.9	20.4	22.8

Table 4.1: Latency measurements for TCP (normalized to path RTT)

To understand the latency distribution of data delivery, Table 4.1 shows the median, 99.9% percentile and worst-case latencies, where network load is varied between two and eight flows. Each value in the table is normalized against the average of TCP's measured RTTs for the corresponding run.[13] In our setup, these averages were usually around 88 ms. We believe the TCP RTTs are quite accurate and give a faithful representation of network level propagation and queuing delays, hence the normalized ratios give a clear view of the additional delay due to the transport level. According to the data of Table 4.1, the only setting in which TCP has acceptable latency, below typical human interaction threshold, is a run with

[13]We used the Linux TCPINFO socket option to query the actual RTT measured by TCP.

2 flows, which is the only run where the bandwidth requirements of the flows does not fully saturate the bottleneck link. With 8 videos, TCP's performance has median latency over 650 ms (7.4 RTTs) and worst case latency around 2 sec (22.8 RTTs) which is not suitable for interactive applications.

Flows	Median Latency		
	TCP	PaceK	PaceA
8	7.4	2.0	2.0
16	9.3	2.0	2.1
24	8.8	2.1	2.7
32	9.5	2.7	3.0

(a) Median

Flows	99.9 percentile			Worst Case		
	TCP	PaceK	PaceA	TCP	PaceK	PaceA
8	18.4	5.4	5.7	22.8	9.4	9.4
16	27.3	9.4	9.8	35.7	13.8	15.0
24	35.5	13.4	14.0	49.6	24.1	23.1
32	45.3	18.0	20.8	73.6	34.6	37.7

(b) Tail

Table 4.2: Latency measurements for different latency controllers (normalized to path RTT)

Using 8 flows as a starting point, we consider a range of traffic loads, consisting of 8, 16, 24, and 32 Paceline flows (with aggregate bit-rates of 32, 64, 96, and 128 Mbps respectively compared to our 16Mbps bottleneck). Table 4.2 shows the latency measurements for TCP, PaceK and PaceA. To focus on the performance of the latency controllers, these results are with the failover feature disabled. For each number of flows, PaceA and PaceK consistently improve on TCP by a significant margin in median (improvement factor 3–4.5), 99.9% latency (improvement factor 2–3), and worst case latency (improvement factor 2–2.5). In all modes, the worst case latency measure is significantly higher than the median.

In Table 4.3, we show Latency results for PaceK and PaceA when we enable failover. Enabling failover doesn't make noticeable changes in the median, but improves the 99.9% latency and the worst case noticeably. The improvement over the non-failover case is up to a factor of 2 in congested settings. For the remainder

Flows	Median Latency	
	PaceK	PaceA
8	2.0	2.0
16	2.1	2.1
24	2.2	2.7
32	2.8	2.9

(a) Median

Flows	99.9 percentile		Worst Case	
	PaceK	PaceA	PaceK	PaceA
8	4.4	5.1	6.5	7.2
16	7.4	8.4	10.8	11.0
24	9.4	10.1	13.9	14.4
32	11.4	12.7	18.2	18.3

(b) Tail

Table 4.3: Latency measurements for different latency controllers with failover (normalized to path RTT)

of the evaluation, all results quoted for Paceline are with failover enabled.

It is informative to notice that PaceA with 32 videos (with failover enabled) has significantly lower latency profile (median, 99%, and worst case) compared to TCP with 4 videos. Thus, Paceline can maintain TCP's current latency profile with 8 times the number of flows. In addition, the worst case latency in Paceline with failover enabled is more predictable and consistent with a maximum confidence interval of 1. However, the worst case latency without failover varies considerably in congested networks with confidence intervals reaching 10.

Fairness

The previous experiments showed that Paceline significantly improves latency relative to TCP. We examine bandwidth fairness for PaceK, PaceA, SST and TCP. We do this in two steps, first where all flows are of the same type, and second when mixing Paceline or SST flows with TCP flows. We use two metrics to quantify bandwidth fairness. These two metrics were calculated using application level data to allow fair comparison with SST.

The first metric is the Jain fairness index [34] (results in Table 4.4), which is defined by the following equation: $fairness = \frac{(\sum x_i)^2}{(n\sum x_i^2)}$, where n is the number of flows and x_i is the bandwidth allocated to flow number i in a given time slot. This index ranges from $1/n$ (worst case) to 1 (best case), we scale it to percentages. SST flows has relatively better fairness than TCP flows when the link is extremely congested (i.e., 32 videos). More importantly, Paceline generally matches TCP's standard level of fairness in addition to improving upon TCP's latency profile.

Flows	Jain fairness index			
	TCP	PaceK	PaceA	SST
8	92.5	90.5	89.9	85.8
16	90.7	88.5	88.6	88.3
24	85.1	83.0	82.8	85.6
32	79.0	79.4	77.3	85.5

Table 4.4: Fairness measurements for different latency controllers

To understand fairness in greater detail than allowed with the Jain index, we also convert bandwidth measurements from the same experiments into a form of CDF for the sharing ratio between different flows. We computed the fairness CDF as follows. We subdivided the measurement time-line into uniform time slots (e.g., every 500ms). For each time slot, we compute the sharing ratio of each flow's bandwidth to the fair bandwidth share (i.e., total bandwidth / number of flows). We then plot the CDF of the sharing ratio for each transport mode. We prefer these CDF's to a single metric because their shapes convey unfairness both in terms of per-flow bandwidth (x position) and degree of affected flows (y position). In the case of perfect fairness, the CDF would appear as a single vertical step at $1/n$, i.e., all flows get equal bandwidth in every time slot. In general, as fairness decreases, so will the slope of the CDF line. Also, if the left y-intercept is non-zero, as seen in Figure 4.8 with the extremely congested scenario of 32 flows, it indicates that some of the flows experienced total starvation. This fairness measure was instrumental in refining our rate control and failover algorithms to meet our goal of maintaining fairness equivalent to TCP.

We computed these fairness CDFs for all of the cases of the previous experiment: with the number of Paceline flows ranging from 8 to 32, for each case of

TCP, PaceK, PaceA, and SST. In Figures 4.7 and 4.8, we show results when using 500ms intervals. We show only the 8 and 32 flow CDFs since the general trend across all the link load configurations (i.e., 8, 16, 24, and 32 videos) is consistent. The shape of the CDFs show that Paceline is able to preserve TCP's fairness. SST's fairness is the same, which is not surprising since SST's congestion algorithm is based on that of TCP. Also as expected, comparing Figures 4.7 and 4.8 shows that fairness degrades as more flows share a link.

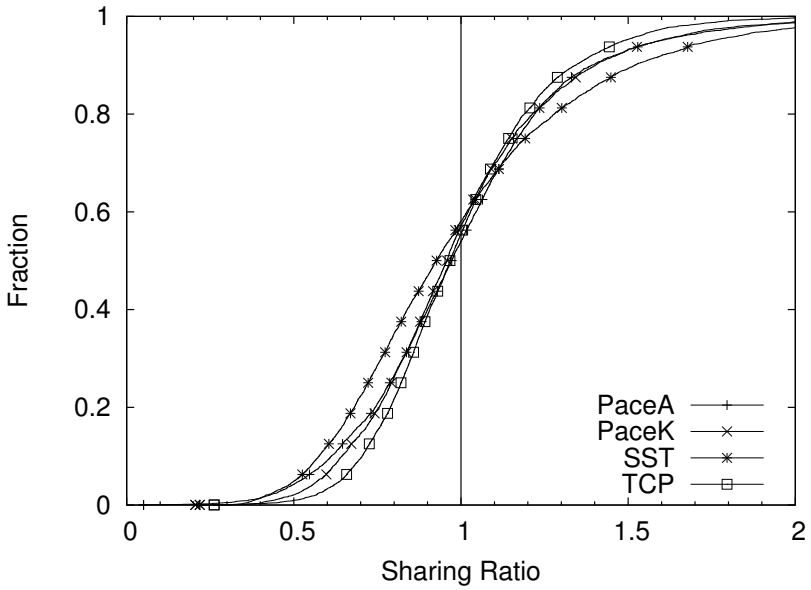


Figure 4.7: Fairness CDF of SST, PaceA and PaceK compared to TCP using 8 flows

Incremental Deployment

In the fairness experiments above, the flows in a given experiment were of the same type. Since TCP is the dominant transport in the Internet, it is important to see how a new transport such as Paceline or SST can be incrementally deployed and share bandwidth fairly and safely with TCP flows. In addition, it is important to verify that the latency advantages of Paceline still exist when Paceline flows compete with normal TCP flows. Paceline ideally should deliver its latency advantages even in

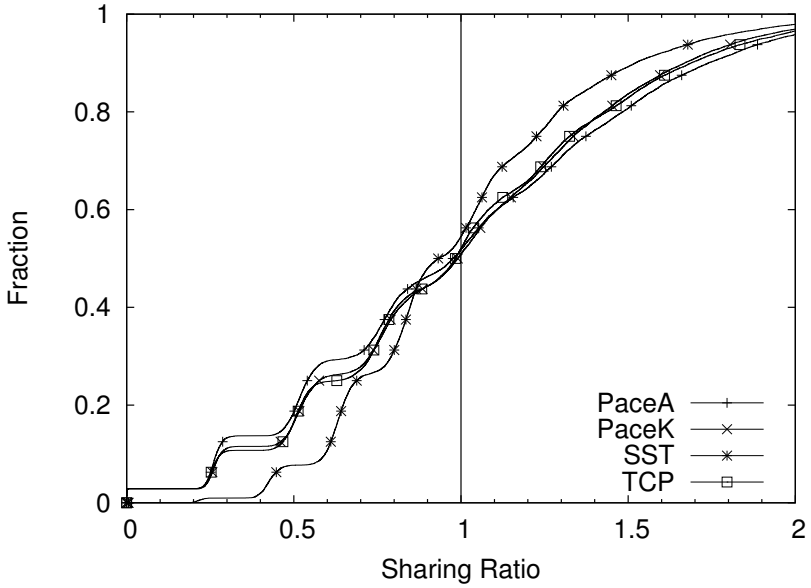


Figure 4.8: Fairness CDF of SST, PaceA and PaceK compared to TCP using 32 flows

mixed deployment without harming existing TCP traffic.

We conduct a series of experiments with a mixture of TCP and PaceK, PaceA or SST flows, with 16 video flows total. We vary the mix of flows using TCP from 0-75 percent. Table 4.5 shows the median latency, for PaceK and PaceA, for each mixture of flows. As the fraction of TCP flows in the mix increases, the median, 99.9%, and worst case latency of PaceA and PaceK are not affected.

TCP %	Median Latency		99.9 percentile		Worst Case	
	PaceK	PaceA	PaceK	PaceA	PaceK	PaceA
0	2.1	2.1	7.4	8.4	10.8	11.0
25	2.1	2.2	7.2	8.4	10.0	11.3
50	2.0	2.2	6.9	8.2	8.7	10.2
75	2.0	2.3	6.8	8.4	9.1	10.1

Table 4.5: Median latency measurements (normalized to path RTT) for different latency controllers and mixed TCP with 16 flows

Considering fairness, we show the result for one case, PaceK and TCP with

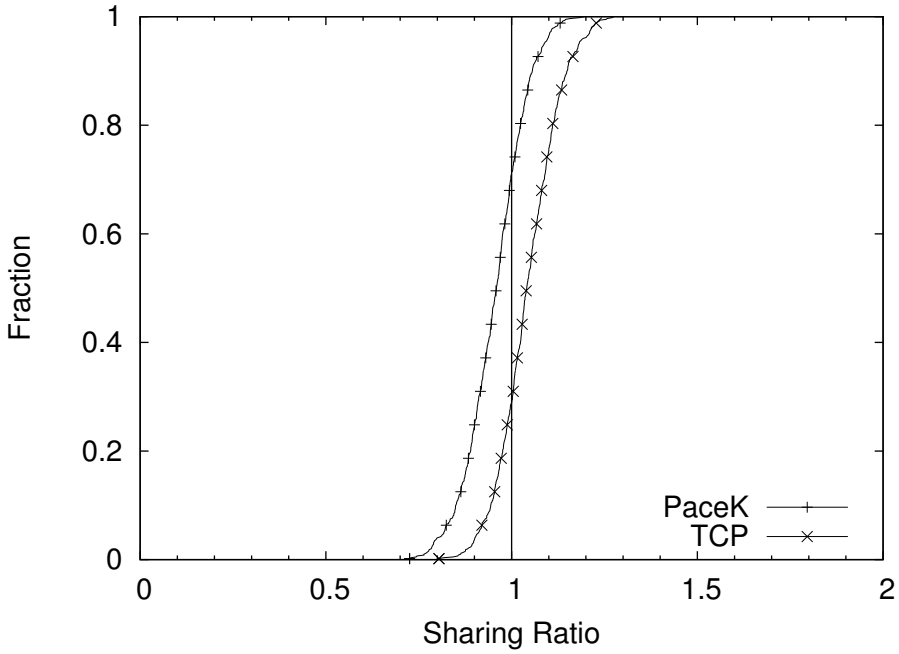


Figure 4.9: CDF of fairness with 8 PACE-K flows 8 TCP flows

8 flows each (50% TCP), in Figure 4.9. The general fairness trend is preserved, although TCP gets more than its fair share with constant factor, on average about 10% with PaceA. The other cases exhibit the same pattern, but the factors differ, with PaceA we see 8% and with SST we see 9%. We can see that neither Paceline nor SST represent a threat to TCP traffic.

Utilization and Wire Overhead

The general premise of adaptive delivery is that available bandwidth is dynamic, and that the application should use its full fair share to maximize quality. Hence a basic requirement for the transport is that it be effective in finding available bandwidth. We showed that Paceline shares bandwidth fairly (Table 4.4). While fairness concerns division of bandwidth between flows, it is also important that the aggregate bandwidth utilize the capacity of the bottleneck. We measured raw utilization using packet traces collected via tcpdump on the bottleneck link of our experiment. Also, to understand how much of the raw utilization is consumed by transport level

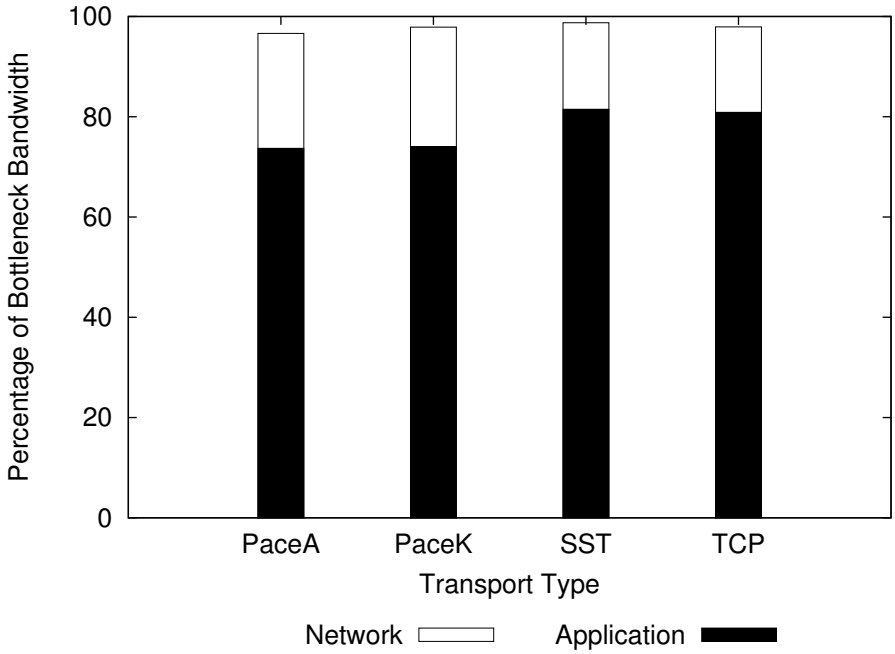


Figure 4.10: Network utilization versus application-level throughput

overhead, we measured the data rate delivered at the application level on the receiver. This rate is lower than raw utilization due to lower layer packet header overheads (Paceline, SST, TCP, IP, etc.). Note this throughput does not account for data that is delivered to the receiver, but is dropped at the application layer because of high latency. We consider that issue in the next section.

The difference between tcpdump utilization and the application throughput represents the basic wire overheads of the transport (and lower) layers. From Figure 4.10, all transports in our experiment achieve high network level utilization, the numbers are between 98.7% (SST) and 96.6% (PaceA).

Transport Level Performance Summary

Compared to TCP, Paceline's PaceA and PaceK algorithms reduce the median end-to-end latency by a factor of 3–4x. With failover, the 99.9% and worst case latency improves by a factor of 3–4x (in Tables 4.2 and Table 4.3). PaceA and PaceK have similar bandwidth fairness to TCP while SST has better fairness in congested

settings. In addition, Paceline shares bandwidth fairly with TCP flows while retaining all latency improvements so it is incrementally deployable in the Internet. All transports we examined have high network utilization and reasonable wire overhead.

4.4.2 Application Level Performance

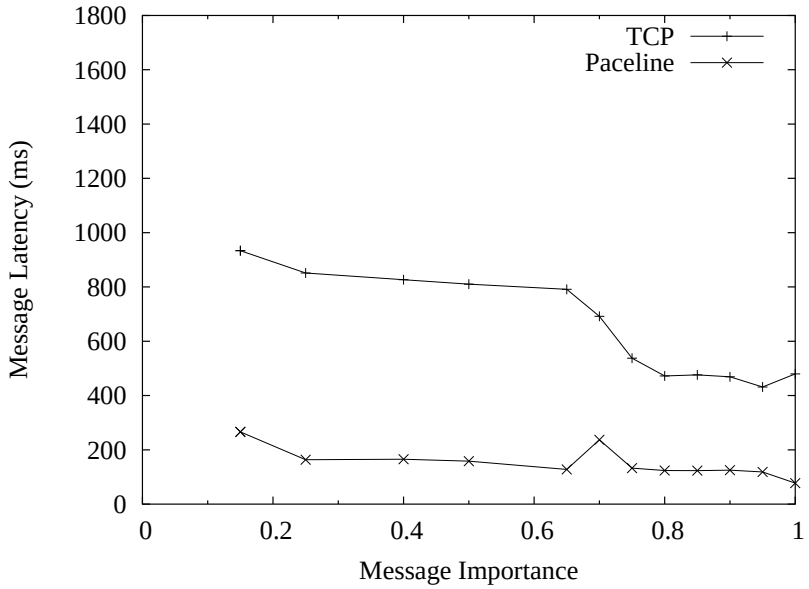
In Section 4.2 we gave an overview of how adaptive applications could work over best effort transport, by prioritizing ADUs and canceling low importance ADUs if they can not be delivered before an expiry time. We now evaluate the performance of such an application with Paceline, in terms of application level quality metrics. We show the message latency with respect to assigned priority; and then we shed light on the nature of the tradeoff between overall quality and interactivity.

Latency Effects on Quality

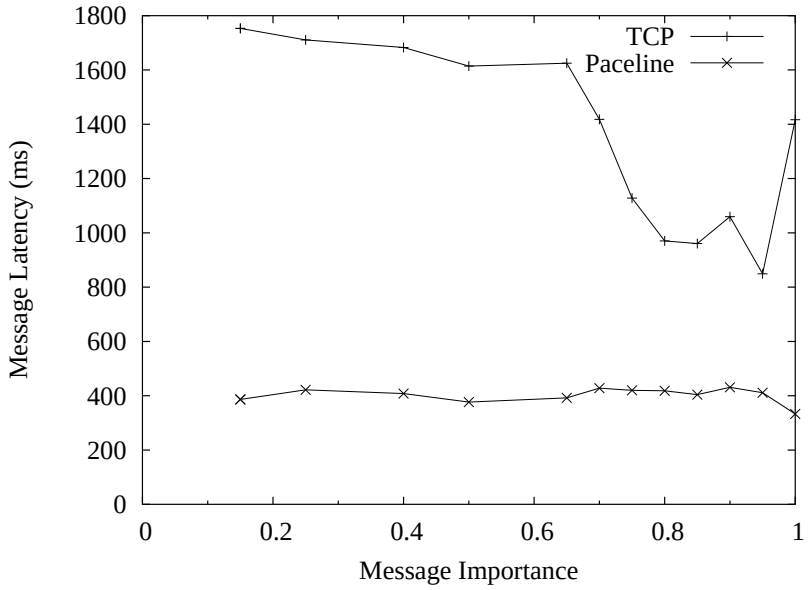
In this section, we conduct a simple experiment to support our claim that Paceline provides low latency for important data. The bottleneck link in this experiment allows 12Mbps of traffic in each direction with a 30ms round-trip time between the two LANs. We conduct this experiment with two servers and eight clients. In each run, the servers stream a single video to their clients using either TCP or Paceline. We measure one-way message delay in relation to the message importance. Every message given to the stream layer is timestamped with its write time and the receiver compares it against the arrival time.[14] The network is highly congested forcing the application to drop some messages. We expect some low importance data to be dropped (or sent late), while higher importance data be delivered with low latency. Figure 4.11 shows one-way end-to-end latency of delivered messages. Messages are spread into buckets, according to their importance. Figure 4.11a presents the median latency, while Figure 4.11b is the 99.9th percentile latency.

The servers enforce strict timing on outgoing messages. If a message has not been transmitted 200ms after it has been given to the stream layer, the application cancels it. The one way delay from a server to a client is 15ms, and the constrained

[14] We synchronize times using ntp. We do this only for experimental purposes, in normal operation QStream adjusts server and client side expiry times independently.



(a) Median



(b) 99.9th Percentile

Figure 4.11: End-to-end message latency based on message importance

link imposes an additional two bandwidth delay product (BDP) queuing delay, for a total of 75ms one-way delay. We would expect to see messages delivered with end-to-end latency of 275ms or less. As can be seen in both figures, TCP has high latency, where the median is well above the 275ms threshold. Paceline, on the other hand, manages to keep the median latency very close to the one-way delay (75ms) for more important data (importance equal to 1). Both TCP and Paceline exhibit higher median latency for less important messages.

Large application messages that are spread over multiple TCP packets may be transmitted over the network in different round trips, increasing the overall end-to-end delay of the message. The situation is aggravated if any of the packets have to be retransmitted, adding more round trips to the message delay. Thus, we expect the 99.9th percentile message delays to be more than the 275ms threshold allowed by the application. We can see that Paceline has a consistent 99.9th percentile latency, compared to TCP, which is close to 400ms for all messages. For TCP, the 99.9th percentile is above a second for the majority of messages, almost reaching 1.8 seconds in some cases.

In summary, Paceline reduces latency significantly compared to TCP. Median latency is relatively lower for important data (minimum one-way delay for Paceline) and higher for less important data. Paceline has a more consistent worst case latency compared to TCP. These results support previous results about Paceline's performance.

Quality and Interactivity Tradeoff

We use the same experimental setup as in the transport level evaluation (16Mbps). One of the main issues we wish to shed light on in this part of our evaluation is the nature of the tradeoff between overall quality and interactivity—better interactivity (lower latency) generally comes at the expense of video quality (e.g., spacial detail). In the following experiments, we fix the number of flows at eight videos, but we vary the level of interactivity, using a configuration parameter we call *latency threshold*. The latency threshold is the amount of time each ADU is given before it expires. In our experimental setup, we synchronize the expiry times between client and server, so that latency threshold exactly determines the application-to-

application transport latency.

To quantify video performance close to the level of user experience, we use three metrics. The first two metrics are the average temporal quality (fps) and average spatial quality (number of spatial layers per frame). The application used is QStream adaptive video streaming application. Adaptation in QStream prioritizes ADUs according to two dimensions of video quality, namely temporal quality (frame rate) and spatial quality (PSNR of frames). The video format used in QStream is called SPEG (Scalable MPEG). In SPEG, each video frame consists of eight ADUs with one base spatial layer ADU and seven (progressive) enhancement spatial layer ADUs. QStream prioritizes ADUs according to a configurable policy that describes the utility of temporal and spatial quality. The default policy is biased toward temporal quality, that is as the bit-rate of a video stream drops, spatial enhancement ADUs are dropped, and when the spatial quality nears minimum, then further reductions in bit-rate will cause dropping of base ADUs which will result in dropping entire frames (lower temporal quality).

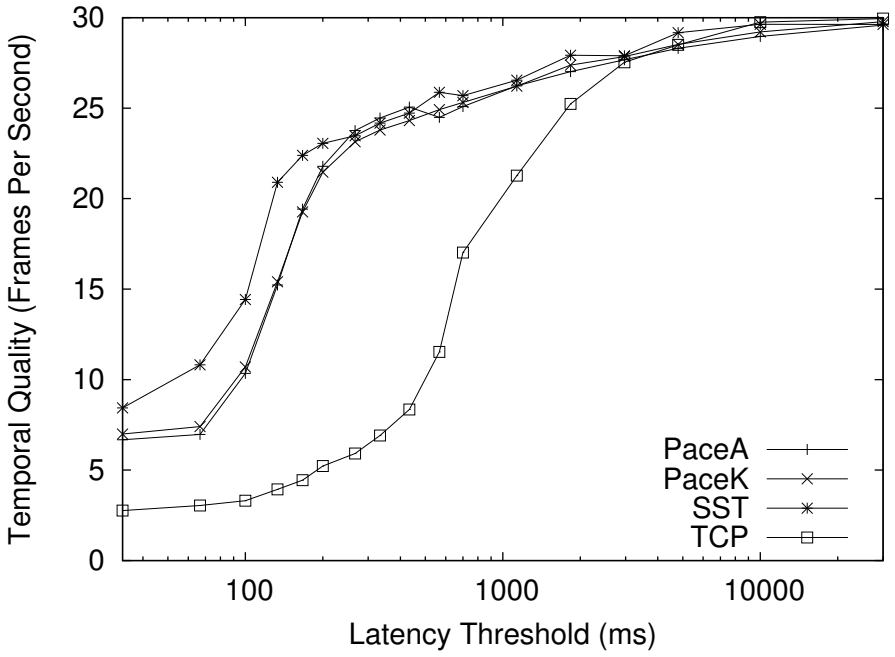


Figure 4.12: Latency threshold versus temporal quality

Figure 4.12 shows the average temporal quality (fps) as the latency threshold

is varied. Notice that in the rightmost side of the graph with highest latency thresholds (10s of seconds), all transports allow QStream to achieve full temporal quality of the video (30fps). The temporal quality when using TCP drops much more rapidly moving leftward (with lower latency thresholds). Despite the fact that TCP delivers high throughput (see previous section), the high transport latency with TCP (see Section 4.4.1) causes frequent head of line (HOL) blocking between low priority ADUs (spatial enhancements) and high priority ADUs (base layers), translating to dropped frames and much lower fps. The trend between SST and Paceline are very similar to each other. Recall that SST's implementation completely avoids HOL blocking. Comparing temporal qualities of Paceline and SST, we see that Paceline also eliminates most of the HOL blocking. Our testbed has a link delay of 15ms and bottleneck link queuing delay of approximately 60ms, for total sender to receiver network delay of about 75ms. Hence, it is not surprising that the temporal quality is very low for all transports as the latency threshold drops below 75ms. However, we can also notice the knee of the Paceline and SST curves in the 100-200ms zone. This indicates that even in this heavily congested network, it is possible to keep within the zone of reasonable interactivity for an application such as video conferencing with a modest impact on quality. On the other hand, using TCP as the transport results in quality not increasing substantially until well over the 500ms point, which is probably not acceptable for comfortable interaction.

To quantify spatial quality, we measured the average number of spatial enhancement layers per frame. There are some similarities with temporal quality, but also some notable differences. Firstly, in the rightmost region, we notice that spatial quality actually drops slightly relative to the peak in the middle. The reason for this has to do with the relative size of spatial layers in our SPEG video format, notably the fact that the base layer ADU is larger than the enhancement layer ADUs, so each base layer ADU (frame) that is dropped actually leaves room to transmit a slightly larger number of enhancement layer ADUs. Since QStream's default policy is biased to temporal quality and the larger latency threshold allows it to send more base layers (higher fps), this in turn causes the mild decrease spatial quality. In the leftmost zone, we see an odd effect where TCP has high spatial quality. Again this is due to HOL blocking, and is not desirable, the small gain in spatial quality comes at severe expense to temporal quality. Although not shown,

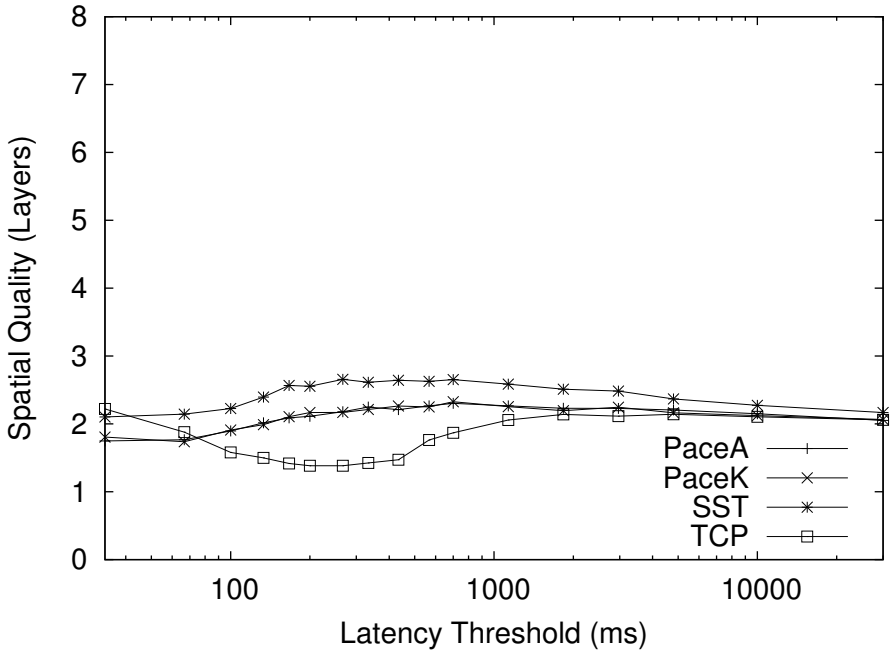


Figure 4.13: Latency threshold versus spatial quality

we observed a very similar result in a version of SST that used round-robin delivery rather than strict priority order.

Although averages capture the general level of quality a user may perceive, they can obscure transient problems that may be very noticeable to a user. To characterize the impact of transient problems (visible glitches or stutters), we use a third metric that we call *display jitter* which is the measured time between each frame displayed. Nominally, display jitter would be the inverse of frame rate, e.g., 30fps translates a display jitter of 33ms. However, because frames are uniformly spaced, dropped frames cause display jitter values that are multiples of the base rate (e.g., 66ms, 99ms, *etc.*), For example, a 20 fps average yields a mix of 33ms and 66ms display jitters.

Noting the knees in the average video quality in Figures 4.12 and 4.13, we choose the case of latency threshold of 200ms and analyze the distribution of all display jitter values. From Figure 4.12, we saw the average temporal quality for Paceline and SST are in the range of 21-23 fps, so we expect most frames to have

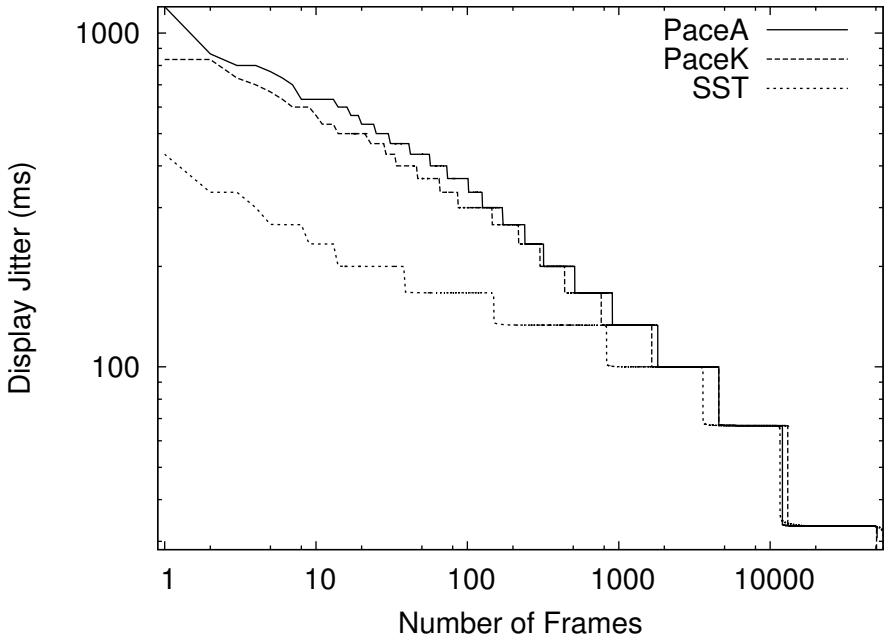


Figure 4.14: Cumulative number of frames displayed at or above level of display jitter

either 33ms or 66ms of display jitter. We do not consider plain TCP here as it's performance is unacceptable. The graph in Figure 4.14 plots the cumulative number of displayed frames, totaled over all videos (around 50000 frames) according to display jitter. The bottom right hand zone shows display jitter to be the same for the vast majority of frames whether Paceline or SST is used, which was already shown in Figure 4.12. Notice that both axes are log scale, which helps clearly see the tail of the distribution. To put these number of frames quantities in perspective, Table 4.6 provides a mapping between number of frames exhibiting a glitch and median time between occurrences. For example, the 100 largest display jitters (glitches) occur with mean period of 23 seconds and with glitch magnitude of 167ms in SST, 300ms in PaceK and 380ms in PaceA.

Number of Frames	Mean time between occurrences
1	39 minutes
10	3.9 minutes
100	23 seconds

Table 4.6: Mapping from number of frames to mean time between occurrence

Application Level Performance Summary

Quality improves if we can tolerate more latency so an application needs to balance between the target latency threshold and the quality level. Using SST as our reference transport with minimum head of line blocking, PaceK and PaceA have a comparable temporal and spatial quality as well as frequency and magnitude of glitches. Therefore, we believe that Paceline is within the similar zone of responsiveness to protocols such as SST. Plain TCP performs very poorly with small latency thresholds, which effectively defeats the application adaptation policy.

PaceA and PaceK have identical application-level performance. PaceA is more portable across different operating systems and more robust in the face of transparent proxies. However, PaceK has less run-time overhead and can eliminate the need for acknowledgments (P-ACKs) simplifying the design of Paceline. PaceA is the more general algorithm while PaceK can be used as a performance optimization in the supported platforms.

4.4.3 Stream Fairness Evaluation

The setup in this experiment consists of four client machines and two server machines. The link allows 12Mbps of traffic in each direction with a 30ms round-trip time between the two LANs. The shaper employs drop-tail queue management with a queue size of twice the bandwidth-delay product.

The real-time video adapts based on a timeline and the policies are implemented through proper selection of message importance and marginal utility. The marginal utility is calculated similar to priority, taking into account the improvement in utility (combined from spatial and temporal quality), and the number of frames affected. We start by evaluating two policies of fairness across streams: re-

source and quality fairness. Then, we investigate weighted fare sharing using both policies.

Resource Fairness

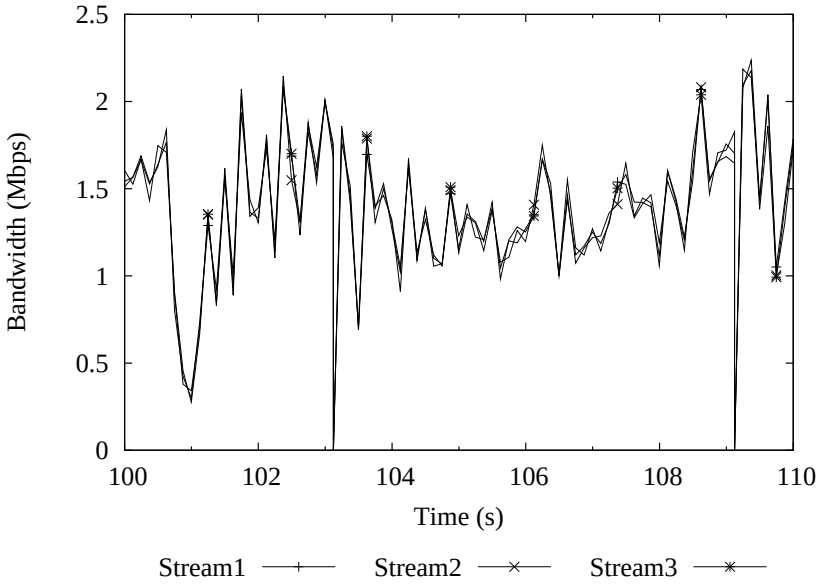
TCP utilizes available bandwidth efficiently and shares bandwidth fairly among concurrent streams. Being layered on top of TCP, independent Paceline channels share bandwidth fairly. In addition, Paceline provides tighter control over resource sharing by performing fair resource sharing across streams multiplexed over the same underlying channel. In the following experiment, we compare TCP's resource sharing (if each stream were to have its own channel) to Paceline's resource sharing when streams share a single channel. We ran the experiment with two servers and four clients. In each run, every server streams three different videos to each client. For TCP fairness, every video was transmitted over its own TCP connection. To demonstrate Paceline's fairness, all the streams used the same underlying channel.

We quantify the fairness of bandwidth sharing between the videos of a single client using the Jain fairness index [34]. We measured the Jain indices of TCP and Paceline bandwidth sharing to be 95.47% and 99.98% respectively. This implies both transports are able to fairly share resources in the long run. To verify the fairness behavior in small time scales, we measured the bandwidth of each video by one client using a 125ms sampling period in Figure 4.15.[15] We see an almost ideal sharing between Paceline streams (Figure 4.15a), whereas the bandwidth of the TCP streams seem less correlated (Figure 4.15b). In small time scales, Paceline can have tighter control over bandwidth sharing.

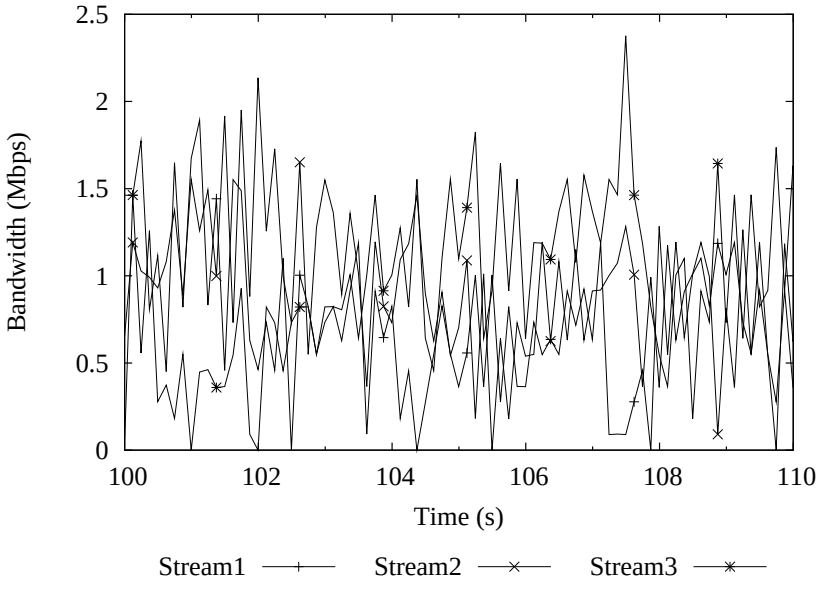
Quality Fairness

We ran the same Paceline fairness experiment using the quality fairness policy instead of resource fairness. Quality of the video is defined by temporal quality (frames per second) and spatial quality (spatial enhancement layers). Figure 4.16a plots the frame rate and Figure 4.16b plots the number of spatial layers of the 3

[15]For clarity we only present a ten second snapshot. We verified that the entire logs exhibit the same pattern.

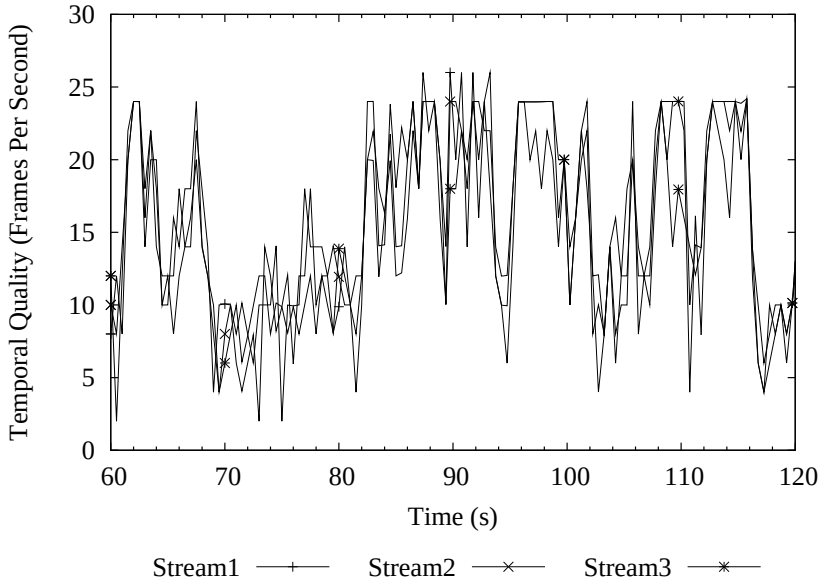


(a) Paceline bandwidth sharing

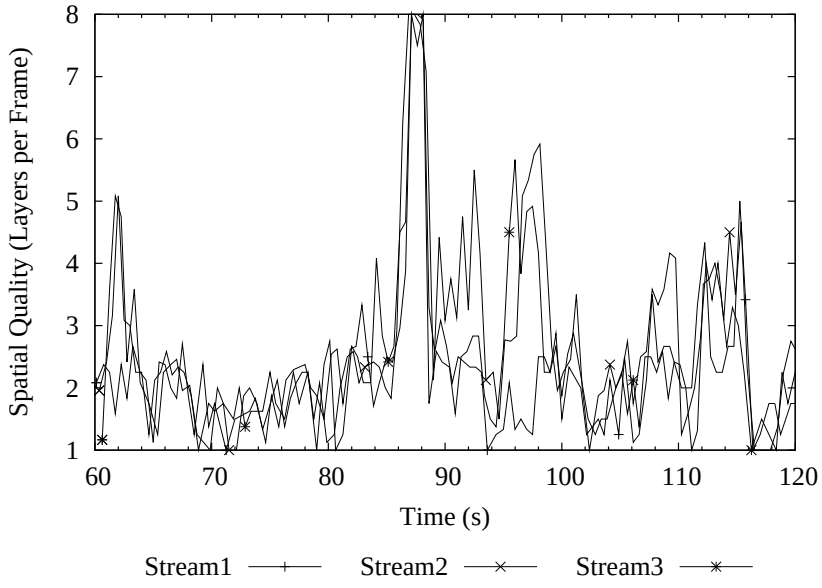


(b) TCP bandwidth sharing

Figure 4.15: Resource fairness policy



(a) Temporal quality



(b) Spatial quality

Figure 4.16: Quality fairness policy

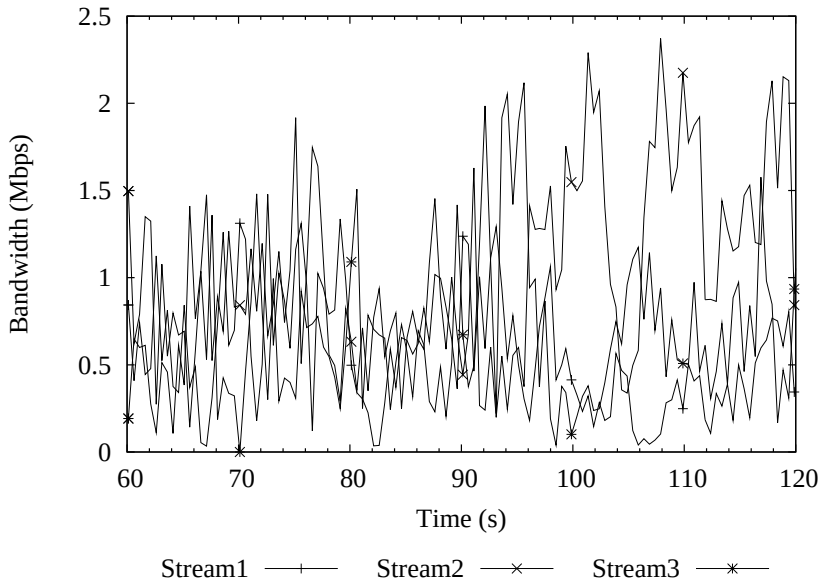


Figure 4.17: Bandwidth with quality fairness

videos over time. The videos (transferred over 3 streams) displays with identical quality (frame rates and spatial layers), which changes based on network conditions, according to quality fairness. Even though the video quality (spatial and temporal) is equal over time, the bandwidth requirements of these video streams are different. Figure 4.17 shows that streams were allocated different shares in the same period in order to achieve equal quality.

Weighted Fairness

While defining both fairness policies, we assumed all streams are equally important. In this section, we evaluate how weighted fair sharing can be used to define importance across streams. For example, in a distance learning software, each client might be generating multiple streams (video, presentation slides, text messages, and advertisements). However, not all these streams are of equal importance. While the video of the speaker may have the core focus of attention, text messages or the slides have lower importance. One would expect decreases in the quality of

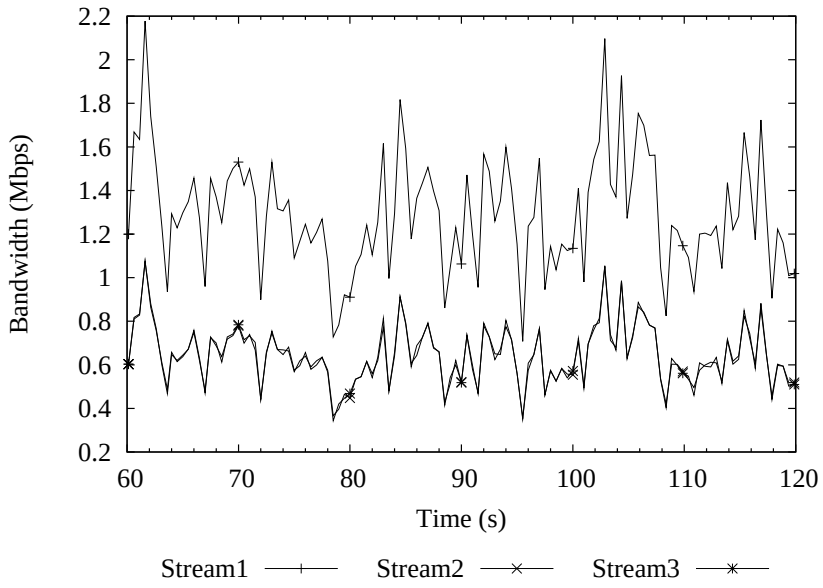


Figure 4.18: Weighted bandwidth fairness

other streams to be more acceptable and less noticeable than the focus video. Such relative importance can be expressed using stream weights.

We use video streams in a setup consisting of four client machines and two server machines. Each server streams three videos to each client. For each resource sharing policy, the clients request one of their videos to be assigned a weight twice the other two, representing the focus video. Figure 4.18 shows the bandwidth allocation to each video stream of a single client with resource fairness. Stream 1 has weight 2 while Stream 2 and Stream 3 have weight 1. Therefore, we see 2:1 relationship in Figure 4.18.

Figure 4.19 shows the temporal quality relationship between streams according to their weights. Stream 1 and Stream 3 were assigned a weight of 1, while Stream 2 was assigned a weight of 3. The sharing policies are work conserving, providing equal resources when available (around the 96th second in the figure), or according to service guarantees when facing resource limitations. We should point out that the utility function we use to map temporal quality (frames per second) to message importance is non-linear. The non-linearity is because we deem importance to vary

more rapidly at lower frame rates, hence a 3:1 ratio of stream weights translates to a less than 3:1 ratio of temporal quality.

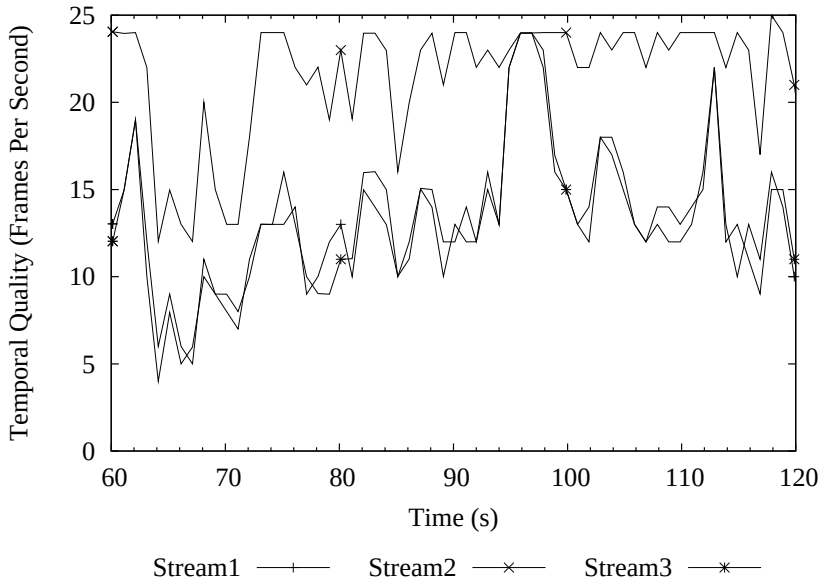


Figure 4.19: Weighted quality fairness

It is interesting to note that quality fairness is completely controlled by application quality metrics. We provide applications with the notion of importance to control adaptation within streams. Weights and marginal utility, on the other hand, specify importance across streams' boundaries.

Summary of Stream Fairness Results

While individual streams can adapt to available resources, the stream layer provides two different notions of fairness across streams: quality and resource fairness. In small time scales, Paceline resource fairness can have tighter control over bandwidth sharing than TCP. Paceline also allows video streams to have equal frame rates and spatial quality using a novel representation of quality fairness.

4.5 Related Work

High bandwidth streams with low latency requirements are challenging to support in current transports. To enhance the quality of service, the best-effort model of the Internet was challenged with various service models, such as IntServ [12], DiffServ [11], and more recently Rate-Delay (RD) network services [55]. For example, RD proposes changing routers to implement a separate queue per traffic type so applications can choose either low delay or high bandwidth traffic, not both. Service models share our motivation of providing high bandwidth and low delay communication; however, the Internet remains a best-effort platform with TCP as the dominant transport (typically carrying more than 90% of the data in the Internet) [25]. Paceline reflects the general trend in multimedia and Web transports [71] toward user-level implementations that leverage TCP's strengths and mitigate its latency weaknesses.

Framing and multiplexing messages in Paceline is similar to RTMP's [78] small fragments that are interleaved and multiplexed over a single TCP connection. Failover, on the other hand, happens when TCP experiences back to back retransmissions and is analogous to the scenario where a user presses the stop/reload buttons in their Web browser upon encountering a slow response. Automated failover may sound quite radical, but it resembles removing exponential backoff from TCP which has been argued to be safe [50].

The main influences on our latency controller have been work on congestion control and alternative transport protocols. TCP Vegas was a seminal work in congestion control that proposed the use of delay measurements to proactively respond to congestion [13]; so Paceline's latency controller is part of the long line of work that has since employed similar techniques. Later work on slowly responsive TCP-friendly congestion control better suits the needs of multimedia applications [3]. A major inspiration for Paceline was work of Bhandarkar [8] which proposed to overcome obstacles facing active queue management (AQM), by emulating it at end hosts. Paceline is similar in that its latency controller can be viewed as a user-level emulation of TCP Vegas rate-based congestion control.

Aside from congestion control, alternative transport service models have also appeared, such as SCTP [56] and DCCP [36], and more recently Structured Streams

[24]. SST is state of the art TCP replacement supporting a light-weight stream abstraction (which acts like the message abstraction in Paceline), congestion-control and reliability. Streams are delivered fully independently with minimum head-of-line blocking. Similar to Paceline, SST supports priority and reset (cancellation) of streams so using one stream per multimedia frame, we were able to support our adaptation service model.

Like SCTP, Paceline identifies head-of-line blocking as a major issue at the transport level. Paceline's use of multiple transport connections has some similarity to SCTP's support of multi-homing, but SCTP's connections are negotiated at session startup and are used with redundant physical paths, while Paceline's failover is dynamic and employed for connections on the same path. Paceline shares the datagram (message) orientation of DCCP, and like DCCP, Paceline was designed with multimedia applications such as video streaming as the main target. However, Paceline works above TCP rather than providing a complete replacement.

Similar to Paceline, SST [24] supports the stream abstraction. Concurrent streams in SST use FIFO scheduling with minimum head-of-line blocking over UDP. On the other hand, Paceline streams share the same underlying TCP channel and use weighted fair sharing across concurrent streams. Paceline's support for fairness across multiple concurrent streams is influenced by CPU scheduling for multimedia applications [41]. Finally, application-level protocols, such as MUX [26] and BEEP [62] multiplex logical streams over one communication channel similar to Paceline; however, they do not deal with timeliness or fairness across streams.

A recent evaluation study for adaptation algorithms [42] has shown that priority-progress adaptation for streaming video is more stable than feedback-based adaptation algorithms in terms of packet delay and jitter. Other interactive multimedia applications can benefit from the adaptation mechanisms in Paceline. Fast-paced large scale games have high bandwidth requirements [10], so they do not adhere to the old wisdom of network games having thin communication streams [53]. DonneyBrook's [10] main contribution is defining interest sets to reduce the bandwidth requirements of games. Priorities can better capture the range of players' interests instead of using two discrete types of updates (important and less frequent). Moreover, cancellation of expired updates can enable rate adaptation based on the

network conditions without using a complex reservation scheme for important updates.

Paceline was used in a cloud-based game prototype to scale the communication in an Epic scale game scenario [69]. The authors used interest sets with two priority classes based on distance to scale communication. Even though the study was limited to a small prototype, adaptation in Paceline improved the performance of wide area networking (WAN).

4.6 Conclusions

Paceline is a transport service supporting interactive high bandwidth applications, such as HD video conferencing, online large scale multi-player games, and virtual reality. Such applications require low latency, but due to their high bandwidth requirements they also require effective congestion control. Paceline leverages the strengths of current TCP implementations, which include their robust and proven congestion control, while mitigating TCP's weaknesses in latency. Paceline employs several techniques to improve latency: message framing and multiplexing to limit transmission delays, a latency-controller to manage client-side queuing delays, and a failover mechanism to handle transient TCP stalls. Paceline reduces TCP's latency profile, enabling an increase in the traffic volume by a factor of 8 with the same latency profile as plain TCP. Using a video conferencing application as an example, Paceline brings dramatic improvements over TCP in terms of video quality metrics and is competitive with the Structured Stream Transport (SST), which is representative of clean-slate replacements for TCP.

Paceline introduces adaptation mechanisms as essential transport primitives to resolve the conflict between timeliness and best-effort transports for high bandwidth multimedia streams. Paceline also enables Priority-Progress adaptation across concurrent video streams. Streams get timely message delivery and equal quality (e.g., frame rates and spatial quality) using a generic representation of quality fairness. Priority-Progress adaptation mechanisms, Paceline enables applications to scale quality with available resources and to use the limited available bandwidth in transferring data with more influence over quality. Paceline's data service model provides interactive applications with the necessary mechanisms: *priority* to pro-

vide timely delivery of important data and *cancellation* to perform informed dropping to match the application data rate with available network bandwidth. These mechanisms can be utilized by domain-specific application-level adaptation policies to provide timely data delivery over best effort transports, mainly TCP and in general other congestion-controlled transports (e.g., SST).

Chapter 5

Conclusions and Future Work

Interactive multimedia applications have low latency interactions and high resource (network bandwidth, CPU, and storage) demands. The demands of multimedia can exceed available resources due to the dynamic fluctuations in application demands or in available resources while using best-effort platforms with no guarantees. The key insight is that it is impossible to process all computations and data in a timely fashion when demands exceed available resources. Our approach, based on the Priority-Progress quality adaptation model, addresses resource volatility by scaling demands (up and down) with the available resources and utilizes scarce resources by giving precedence to computations and data that have more influence over perceived quality.

The mismatch between application demands and available resources is observed to varying degrees in all resources. To reduce the end-to-end delay and improve the overall perceived quality, our research addresses the performance limitations in multiple resources. This thesis addresses the conflict between interactivity and the best effort nature of current transports and execution platforms. We have built enhancement layers to maintain the strengths of best-effort platforms and mitigate their weaknesses through Priority-Progress adaptation. The execution layer, DOHA [20, 21], extends the Priority-Progress CPU adaptation to work in games and across multiple execution threads with no shared memory. Similarly, the transport layer, Paceline [19], introduces low latency techniques over TCP and exposes Priority-Progress adaptation mechanisms as essential transport features.

In the rest of this chapter, we summarize the research contributions of this dissertation. Then, we reflect on the research approach; and finally we suggest venues for future research.

5.1 Primary Research Contributions

The following three primary contributions arose from our research work.

5.1.1 Adaptation as an Essential Infrastructure Feature

This thesis produced a general purpose transport and execution infrastructures that expose Priority-Progress adaptation primitives to address the limitations of best effort Web platforms (i.e., TCP and JavaScript engines) in supporting real-time games and video conferencing. The adaptation primitives were introduced at the appropriate level: the transport stream level in Paceline and at the event-loop level in DOHA without changing the best effort nature of the underlying platforms. Both DOHA and Paceline enable application adaptation through prioritization to provide timely processing of important data with more influence over quality and cancellation to adapt the application rate to match available resources. Important high priority data and computations get better application quality, measured in frames per second and jitter profile, in both DOHA (Section 3.2.1) and Paceline (Section 4.4.2) yielding better overall perceived quality.

In order to provide consistent quality in best effort platforms, the infrastructure and the application need to be agile in responding quickly to the adaptation policy hints. For execution agility, the prevalent monolithic game loop architecture was broken up to only issue one explicit execution event for each game entity allowing adaptation at the fine-granularity of a single event instead of the coarse-granularity of the game frame with all the updates. Secondly, DOHA gives precedence to timer events and respects the priority of best-effort events maximizing quality within the timing limits. To improve transport agility in Paceline, we developed several mechanisms (Section 4.4.1): a rate controller to reduce queuing delay due to excessive socket buffering; a failover mechanism among TCP connections to handle extreme cases of congestion; and a message fragmentation technique to reduce the granularity of preemption. The low-latency techniques in DOHA and Paceline are general

and can allow adaptation policies to respond quickly to fluctuations in resource availability.

5.1.2 Support for Concurrent Software

The Priority-Progress model was extended in this thesis with policies and enhancement layers that support concurrent software. DOHA supported timely execution using multiple concurrent threads with no shared memory and Paceline supported timely message delivery across multiple concurrent streams over a shared TCP channel. For timely execution using HTML5 worker threads, DOHA provided an event loop per worker to allow adaptation in all threads and a communication layer to support state management and load balancing across workers. As a result, quality scaled linearly with a small number of cores in the parallel version of the game (using DOHA) as shown in Section 3.2.2. For timely and fair communication using multiple concurrent message streams, Paceline supports two notions of fairness: resource and quality fairness. Resource fairness in Paceline guarantees fair bandwidth allocation among streams at a finer granularity than TCP. More importantly, Paceline supports quality fairness to ensure fair application-level quality, in terms of frames per second for example, across streams as shown in Section 4.4.2.

5.1.3 Priority-Progress in Games

This thesis developed CPU quality-adaptation policies inspired by Priority-Progress adaptation in a new application domain, HTML5 games. Priority-Progress was developed in video streaming so the policies assumed scalable video coding and the quality dimensions were well-studied in the multimedia literature. Our work in DOHA explored the use of Priority-Progress adaptation in game loops. Our test policy used distance from the player as the criteria to decide what entities have more influence over perceived quality. We also developed a second policy to minimize starvation based on a minimum frequency update per game entity. These are the basic policies for game loop adaptation so they can be extended or overridden in different entities. For example, a bullet entity can add a policy to assign higher priority based on the current speed. Finally, Web-based games have other places where scalability can help trade accuracy for performance, such as the particle en-

gine (visual effects accuracy) and AI logic (algorithm accuracy).

Paceline improved the performance of wide area networking (WAN) in a cloud-based game prototype running an epic scale game scenario [69]. The authors used Paceline with a limited adaptation scheme that has two priority classes based on distance to scale communication. Further research is needed to validate the results of this small study but as the bandwidth demands for games increase Paceline's adaptation mechanisms would be more applicable.

5.2 Secondary Contributions

The following two secondary contributions arose from our work.

1. While re-structuring the simulation engine of an award-winning Web-based game (RAPT [76]), we examined the challenges and opportunities of using HTML5 Web workers and share our qualitative and quantitative observations.

2. Paceline and DOHA with their respective modified applications were contributed to the QStream open-source repository at http://qstream.org to facilitate further research.

5.3 Reflections on the Research Approach

In this thesis, we spanned multiple resources instead of choosing one resource and conducting more studies on adaptation in that resource. We chose two different resources that appeared to define the end-to-end performance of interactive multimedia applications. Our main motivation is that the mismatch between demands and available resources is observed to varying degrees in all resources so any resource can be the performance bottleneck based on the environment conditions and application loads. In addition, fixing a performance concern in one area can cause a new one to arise somewhere else. We reflect on the advantages and disadvantages of this research approach.

Our approach allowed us to get a deeper understanding of adaptation and the overall real-time performance in different interactive multimedia scenarios and

across multiple resources. We were also able to focus on adaptation policies and mechanisms while developing an appreciation of the details special to each resource. On the other hand, we spent more time than anticipated to review related work and gain the required expertise to test our ideas in real application scenarios.

5.4 Future Work

In this section, we present future venues for research related to this thesis.

- The natural extension for DOHA is to build the load-balancing policies that can distribute work evenly across workers. Currently, our adaptation policies and the location-based partitioning algorithms are developed separately. Developing a load-balancing algorithm that is quality-aware can improve the game quality significantly. For example, the load-balancing policy can distribute high priority entities evenly across cores to maximize their chance of getting updated. Since communication is the major performance limitation, this direction of research requires an in-depth study of the communication requirements of entities in different games.

- Concurrency is generally hard and it is a major source of systems errors [63]. More tools to support understanding concurrent programs would be very useful especially after introducing real concurrency in parallel Web workers. In Dingo [63], a state-machine based formal language was used to describe the protocol between device drivers and the operating system. Similarly, we need to capture communication protocols between application components running in different Web workers.

- Performance of multimedia in browsers is not well understood because of the lack of performance monitoring tools and benchmarks to quantify and compare perceived multimedia quality. Browsers have primitive debugging and performance monitoring tools and Web workers have even less support. To conduct a rigorous experimental study and quantify performance, we built a few performance analysis tools. We built a parallel performance monitor to capture performance data from workers, a server to receive and persist data, and a visualization tool to display performance signals in real-time for

interactive performance debugging. We plan to build a tool to record and replay the performance data and the JavaScript execution sessions. We can enable querying over the data for interactive performance analysis to answer questions, such as what are the ten functions with the worst duration? Which functions took more than 50ms to execute? What are the functions leading to a specific performance anomaly? In addition, we can automatically generate representative multimedia Web benchmarks using a similar technique to [61]. The benchmark can use the application-level quality metrics and our visualization tool to compare the browser support for interactive multimedia.

- As explained in Paceline, the main contributing factor to the end-to-end latency is *queuing* delays in nodes (hosts and network routers). Paceline addresses end-hosts queuing delays. To reduce queuing delay in network routers, we can use active queue management (AQM) techniques, such as Explicit Congestion Notification (ECN) [57]. However, these techniques are hard to configure and errors in configuration can reduce bandwidth utilization. Recent work [8] designed an end-host technique for active queue management by modifying TCP. Our future objective would be to emulate AQM at the application-level and eliminate the end-to-end queuing delay without modifying routers or TCP kernel implementations. The main requirement for such a technique is to automatically configure its parameters without manual intervention.

- The storage and memory [59] resources are becoming more important for Web applications. Multimedia storage is server-side challenge especially since browsers limit client-side storage to 5 megabytes (10 MB in Internet Explorer) [79]. The work by Krasic and Légaré [38] proposes the use of Priority-Progress adaptation in the server to enhance the interactivity while accessing stored video. Storage adaptation can be extended to work for real-time persistence and retrieval of interactive content (i.e., video and game sessions) for reply purposes. To match the high demands for interactive content, storage adaptation ideas for one server [38] can be extended to work in a distributed storage model using distributed data structures [29]. On the other hand, memory is not a schedulable resource that we can adapt. Appli-

cations can either have enough memory and function correctly or run out of memory and fail.

5.5 Concluding Remarks

When demands exceed available resources, scaling quality based on available resources using Priority-Progress adaptation improves timeliness and ensures consistent quality in interactive multimedia. Adaptation is an essential infrastructure feature enabling the exploration of ambitious more challenging scenarios without the fear of brittle real-time performance and inconsistent quality. It is imperative to continue investigating the tools, techniques, and infrastructure features needed to support the growing number of interactive multimedia applications.

Bibliography

[1] Ambiera. Copperlicht - fast WebGL javascript 3D engine. http://www.ambiera.com/copperlicht/. [accessed 20-May-2012]. → pages 15, 20

[2] R. Antonello, S. Fernandes, J. Moreira, P. Cunha, C. Kamienski, and D. Sadok. Traffic analysis and synthetic models of second life. *Multimedia Systems*, 2008. doi:10.1007/s00530-008-0125-1. URL http://dx.doi.org/10.1007/s00530-008-0125-1. → pages 45

[3] D. Bansal, H. Balakrishnan, S. Floyd, and S. Shenker. Dynamic behavior of slowly-responsive congestion control algorithms. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 263–274, New York, NY, USA, 2001. ACM. ISBN 1-58113-411-8. doi:10.1145/383059.383080. URL http://doi.acm.org/10.1145/383059.383080. → pages 92

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi:http://doi.acm.org/10.1145/1629575.1629579. URL http://doi.acm.org/10.1145/1629575.1629579. → pages 43

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2): 87–152, Nov. 1992. ISSN 0167-6423. doi:10.1016/0167-6423(92)90005-V. URL http://dx.doi.org/10.1016/0167-6423(92)90005-V. → pages 22, 42

[6] M. J. Best, A. Fedorova, R. Dickie, A. Tagliasacchi, A. Couture-Beil, C. Mustard, S. Mottishaw, A. Brown, Z. F. Huang, X. Xu, N. Ghazali, and A. Brownsword. Searching for concurrent design patterns in video games. Euro-Par '09, pages 912–923, Berlin, Heidelberg, 2009. Springer-Verlag.

ISBN 978-3-642-03868-6.
doi:http://dx.doi.org/10.1007/978-3-642-03869-3_84. URL
http://dx.doi.org/10.1007/978-3-642-03869-3_84. → pages 3, 14, 18, 24

[7] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via scheduling: techniques for efficiently managing shared state. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 640–652, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi:10.1145/1993498.1993573. URL http://doi.acm.org/10.1145/1993498.1993573. → pages 43

[8] S. Bhandarkar, A. L. N. Reddy, Y. Zhang, and D. Loguinov. Emulating AQM from end hosts. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 349–360, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-713-1. doi:10.1145/1282380.1282420. URL http://doi.acm.org/10.1145/1282380.1282420. → pages 92, 101

[9] A. Bharambe, J. Pang, and S. Seshan. Colyseus: a distributed architecture for online multiplayer games. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267680.1267692. → pages 14, 43

[10] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: enabling large-scale, high-speed, peer-to-peer games. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 389–400, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-175-0. doi:10.1145/1402958.1403002. URL http://doi.acm.org/10.1145/1402958.1403002. → pages 3, 14, 42, 45, 53, 93

[11] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475, An architecture for differentiated services. http://www.ietf.org/rfc/rfc2475.txt, 1998. → pages 92

[12] R. Braden, D. Clark, and S. Shenker. RFC 1633, Integrated services in the Internet architecture: an overview. http://www.ietf.org/rfc/rfc1633.txt, 1994. → pages 92

[13] L. Brakmo and L. Peterson. TCP Vegas: end to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13

(8):1465 –1480, oct 1995. ISSN 0733-8716. doi:10.1109/49.464716. → pages 58, 92

[14] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A distributed real-time MPEG video audio player. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '95, pages 142–153, London, UK, 1995. Springer-Verlag. ISBN 3-540-60647-5. URL http://dl.acm.org/citation.cfm?id=646982.712173. → pages 6

[15] J. Chen, B. Wu, M. Delap, B. Knutsson, H. Lu, and C. Amza. Locality aware dynamic load management for massively multiplayer games. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 289–300, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi:10.1145/1065944.1065982. URL http://doi.acm.org/10.1145/1065944.1065982. → pages 14

[16] M. Claypool and K. Claypool. Latency and player actions in online games. *Commun. ACM*, 49(11):40–45, 2006. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/1167838.1167860. → pages 3

[17] M. Cook. Pistol slut. http://pistolslut.com, 2011. [accessed 3-May-2011]. → pages 3, 13, 14, 15, 18, 20

[18] B. Entertainment. World war craft pvp battlegrounds. http://www.worldofwarcraft.com. URL http://www.worldofwarcraft.com/index.xml. [accessed 9-Oct-2009]. → pages 14

[19] A. Erbad, M. Tayarani Najaran, and C. Krasic. Paceline: latency management through adaptive output. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, pages 181–192, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-914-5. doi:http://doi.acm.org/10.1145/1730836.1730858. URL http://doi.acm.org/10.1145/1730836.1730858. → pages iii, 8, 9, 16, 96

[20] A. Erbad, N. C. Hutchinson, and C. Krasic. Scalable quality for web-based games. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, PLASTIC '11, pages 57–60, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1171-7. doi:10.1145/2093328.2093338. URL http://doi.acm.org/10.1145/2093328.2093338. → pages iii, 7, 15, 96

[21] A. Erbad, N. C. Hutchinson, and C. Krasic. DOHA: scalable real-time web applications through adaptive concurrent execution. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 161–170, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1229-5. doi:10.1145/2187836.2187859. URL http://doi.acm.org/10.1145/2187836.2187859. → pages iii, 7, 8, 15, 96

[22] N. Feamster, D. Bansal, and H. Balakrishnan. On the interactions between layered quality adaptation and congestion control for streaming video. In *Proc. of the 11th International Packet Video Workshop*, 2001. → pages 6

[23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616, Hypertext Transfer Protocol (HTTP)/1.1. http://tools.ietf.org/html/rfc2616, 1999. → pages 5

[24] B. Ford. Structured streams: a new transport abstraction. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 361–372, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-713-1. doi:10.1145/1282380.1282421. URL http://doi.acm.org/10.1145/1282380.1282421. → pages 47, 68, 93

[25] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17:6–16, 2003. → pages 5, 16, 47, 92

[26] J. Gettys. Simple MUX protocol specification. http://www.w3.org/Protocols/MUX/WD-mux-970825.html, 1996. → pages 93

[27] A. Goel, J. Walpole, and M. Shor. Real-rate scheduling. In *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pages 434 – 441, may 2004. doi:10.1109/RTTAS.2004.1317290. → pages 6

[28] A. Goel, C. Krasic, and J. Walpole. Low-latency adaptive streaming over TCP. *ACM Trans. Multimedia Comput. Commun. Appl.*, 4(3):1–20, 2008. ISSN 1551-6857. doi:http://doi.acm.org/10.1145/1386109.1386113. → pages 5, 49, 56

[29] S. Gramsci. A scalable video streaming approach using distributed b-tree. http://hdl.handle.net/2429/33848, 2011. → pages 101

[30] I. Hickson. Web workers. http://dev.w3.org/html5/workers/, 2009. → pages 4, 19, 25

[31] I. Hickson. HTML5: A vocabulary and associated APIs for HTML and XHTML. http://dev.w3.org/html5/spec/Overview.html, 2011. → pages 12

[32] I. Hickson. The WebSocket API. http://dev.w3.org/html5/websockets/, 2011. → pages 13

[33] International Telecommunication Union (ITU). Transmission systems and media, general recommendation on the transmission quality for an entire international telephone connection; one way transmission time (recommendation g.114). Technical report, Telecommunication Standardization Sector of ITU, 1993. → pages 3

[34] R. Jain, D. Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report TR-301, DEC Research, September 1984. URL http://www.cse.wustl.edu/~jain/papers/fairness.htm. → pages 73, 86

[35] Joyent Inc. Node.js: Evented i/o for v8 javascript. http://nodejs.org/. [accessed 2-Nov-2011]. → pages 42

[36] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: congestion control without reliability. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 27–38, New York, NY, USA, 2006. ACM. ISBN 1-59593-308-5. doi:10.1145/1159913.1159918. URL http://doi.acm.org/10.1145/1159913.1159918. → pages 47, 92

[37] C. Krasic. *A framework for quality-adaptive media streaming: encode once - stream anywhere*. PhD thesis, 2004. AAI3119036. → pages 1, 2, 6, 7, 8, 16, 19, 42, 45

[38] C. Krasic and J.-S. Légaré. Interactivity and scalability enhancements for quality-adaptive streaming. In *Proceedings of the 16th ACM international conference on Multimedia*, MM '08, pages 753–756, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-303-7. doi:10.1145/1459359.1459478. URL http://doi.acm.org/10.1145/1459359.1459478. → pages 2, 101

[39] C. Krasic, J. Walpole, and W. Feng. Quality-adaptive media streaming by priority drop. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pages 112–121, June 2003. → pages 6, 7, 16, 22, 42, 53, 68, 69

[40] C. Krasic, A. Sinha, and L. Kirsh. Priority-progress CPU adaptation for elastic real-time applications. In *Proc. of the Multimedia Computing and Networking Conference (MMCN)*, 2007. → pages 1, 2, 7

[41] C. Krasic, M. Saubhasik, A. Sinha, and A. Goel. Fair and timely scheduling via cooperative polling. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 103–116, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi:10.1145/1519065.1519077. URL http://doi.acm.org/10.1145/1519065.1519077. → pages 22, 42, 93

[42] R. Kuschnig, I. Kofler, and H. Hellwagner. An evaluation of TCP-based rate-control algorithms for adaptive internet streaming of H.264/SVC. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, MMSys '10, pages 157–168, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-914-5. doi:http://doi.acm.org/10.1145/1730836.1730856. URL http://doi.acm.org/10.1145/1730836.1730856. → pages 2, 16, 46, 93

[43] S. Ladd. Box2d and web workers for javascript developers. http://blog.sethladd.com/2011/09/box2d-and-web-workers-for-javascript.html, 2011. → pages 43

[44] B. Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9):1632 –1650, sep 1999. ISSN 0733-8716. doi:10.1109/49.790486. → pages 6

[45] S. Lohr. For impatient Web users, an eye blink is just too long to wait. http://www.nytimes.com/2012/03/01/technology, 2012. [accessed 2-Mar-2012]. → pages 1

[46] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 41–54, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-577-2. doi:http://doi.acm.org/10.1145/1755913.1755919. URL http://doi.acm.org/10.1145/1755913.1755919. → pages 43

[47] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 711–720, New York, NY, USA, 2010. ACM. ISBN

978-1-60558-799-8. doi:10.1145/1772690.1772763. URL http://doi.acm.org/10.1145/1772690.1772763. → pages 43

[48] J. Mickens and M. Dhawan. Atlantis: robust, extensible execution environments for web applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 217–231, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi:http://doi.acm.org/10.1145/2043556.2043577. URL http://doi.acm.org/10.1145/2043556.2043577. → pages 44

[49] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of AFIPS Fall Joint Computer Conference*, volume 33, pages 267–277, 1968. → pages 3

[50] A. Mondal and A. Kuzmanovic. Removing exponential backoff from TCP. *SIGCOMM Comput. Commun. Rev.*, 38(5):17–28, 2008. ISSN 0146-4833. doi:http://doi.acm.org/10.1145/1452335.1452338. → pages 92

[51] Mozilla Developer Center. Gecko plugin API reference. http://developer.mozilla.org/en/docs/Gecko_Plugin_API_Reference. [accessed 9-Oct-2009]. → pages 10

[52] Mozilla Labs. Rainbow. https://mozillalabs.com/rainbow/, 2011. → pages 13

[53] A. Petlund, K. Evensen, P. Halvorsen, and C. Griwodz. Improving application layer latency for reliable thin-stream game traffic. In *NetGames '08: Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 91–96, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-132-3. doi:http://doi.acm.org/10.1145/1517494.1517513. → pages 93

[54] D. Pittman and C. GauthierDickey. A measurement study of virtual populations in massively multiplayer online games. In *NetGames '07: Proceedings of the 6th ACM SIGCOMM workshop on Network and system support for games*, pages 25–30, New York, NY, USA, 2007. ACM. ISBN 978-0-9804460-0-5. doi:http://doi.acm.org/10.1145/1326257.1326262. → pages 14

[55] M. Podlesny and S. Gorinsky. Rd network services: differentiation through performance incentives. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 255–266, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-175-0.

doi:10.1145/1402958.1402988. URL http://doi.acm.org/10.1145/1402958.1402988. → pages 92

[56] E. R. Stewart. RFC 4960, Stream control transmission protocol (SCTP). http://tools.ietf.org/html/rfc4960, 2007. → pages 47, 92

[57] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168 (Proposed Standard), September 2001. URL http://www.ietf.org/rfc/rfc3168.txt. → pages 101

[58] A. Ranganathan and J. Sicking. File API. http://www.w3.org/TR/FileAPI/. URL http://www.w3.org/TR/FileAPI/. [accessed 12-Dec-2011]. → pages 13

[59] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1863166.1863169. → pages 22, 42, 101

[60] R. Rejaie, M. Handley, and D. Estrin. Quality adaptation for congestion controlled video playback over the internet. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '99, pages 189–200, New York, NY, USA, 1999. ACM. ISBN 1-58113-135-6. doi:10.1145/316188.316222. URL http://doi.acm.org/10.1145/316188.316222. → pages 6

[61] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi:10.1145/2048066.2048119. URL http://doi.acm.org/10.1145/2048066.2048119. → pages 101

[62] M. Rose. RFC 3080, The blocks extensible exchange protocol core (BEEP). http://tools.ietf.org/html/rfc3080, 2001. → pages 93

[63] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 275–288, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-482-9. doi:10.1145/1519065.1519095. URL http://doi.acm.org/10.1145/1519065.1519095. → pages 44, 100

[64] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(9):1103 –1120, sept. 2007. ISSN 1051-8215. doi:10.1109/TCSVT.2007.905532. → pages 7

[65] D. Sisalem and F. Emanuel. QoS control using adaptive layered data transmission. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, ICMCS '98, pages 4–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8557-3. doi:10.1109/MMCS.1998.693620. URL http://dx.doi.org/10.1109/MMCS.1998.693620. → pages 6

[66] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, RTSS '01, pages 59–, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1420-0. URL http://dl.acm.org/citation.cfm?id=882482.883826. → pages 6

[67] D. Szablewski. The awesomest way to create even more awesome HTML5 games! http://impactjs.com/. [accessed 28-Feb-2012]. → pages 4, 18

[68] D. Szablewski. Biolab disaster. http://playbiolab.com, 2011. URL http://playbiolab.com. [accessed 3-May-2011]. → pages 3, 14, 15, 18, 20

[69] M. Tayarani Najaran and C. Krasic. Scaling online games with adaptive interest management in the cloud. In *Proceedings of the 9th Annual Workshop on Network and Systems Support for Games*, NetGames '10, pages 9:1–9:6, Piscataway, NJ, USA, 2010. IEEE Press. ISBN 978-1-4244-8355-6. URL http://dl.acm.org/citation.cfm?id=1944796.1944805. → pages 16, 94, 99

[70] Tcpdump/Libpcap. Tcpdump and libcap. http://www.tcpdump.org/, 2010. [accessed 6-Dec-2011]. → pages 61

[71] The Chromium Projects. SPDY: An experimental protocol for a faster web. http://www.chromium.org/spdy/spdy-whitepaper, 2011. → pages 5, 92

[72] The Mozilla Foundation. High scores. https://gaming.mozillalabs.com/games/winners, 2011. [accessed 3-May-2011]. → pages 23

[73] The Wireshark Foundation. Wireshark: The world's foremost network protocol analyzer. http://www.wireshark.org/. [accessed 6-Dec-2011]. → pages 61

[74] D. D. Tran, I. Oksanen, and I. Kliche. The media capture API. http://www.w3.org/TR/media-capture-api/. [accessed 12-Dec-2011]. → pages 13

[75] B. Vandalore, W. chi Feng, R. Jain, and S. Fahmy. A survey of application layer techniques for adaptive streaming of multimedia. *Real-Time Imaging*, 7(3):221 – 235, 2001. ISSN 1077-2014. doi:DOI:10.1006/rtim.2001.0224. → pages 16

[76] E. Wallace, J. Ardini, and K. Gishen. Robots are people too. http://raptjs.com, 2011. [accessed 3-May-2011]. → pages 3, 4, 8, 13, 14, 15, 18, 20, 23, 99

[77] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via TCP: An analytic performance study. *ACM Trans. Multimedia Comput. Commun. Appl.*, 4(2):1–22, 2008. ISSN 1551-6857. doi:http://doi.acm.org/10.1145/1352012.1352020. → pages 5, 13

[78] Wekipedia. Real time messaging protocol (RTMP). http://en.wikipedia.org/wiki/Real_Time_Messaging_Protocol, . [accessed 9-Oct-2009]. → pages 92

[79] Wekipedia. Web storage. http://en.wikipedia.org/wiki/Web_storage, . [accessed 24-Feb-2012]. → pages 101

[80] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 255–270, New York, NY, USA, 2002. ACM. ISBN 978-1-4503-0111-4. doi:10.1145/1060289.1060313. URL http://doi.acm.org/10.1145/1060289.1060313. → pages 68

[81] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3633-0. doi:10.1109/SP.2009.25. URL http://dl.acm.org/citation.cfm?id=1607723.1608126. → pages 43

[82] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazires, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX 2003 Annual Technical Conference*, pages 239–252. USENIX, 2003. → pages 28, 43