INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

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

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Composing and Coordinating Adaptations in Cholla

by

Patrick G. Bridges

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2002

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Patrick G. Bridges entitled Composing and Coordinating Adaptations in Cholla

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Richard D. Schlichting	12/16/02 Date
Matti Hiltunen	12/16/02 Date
John H. Hartman	12/16/02 Date
John Palmer	12/17/02 Date
Joseph Watkins	12/16/02 Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

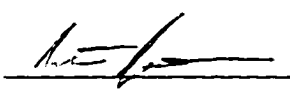
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Richard D. Schlichting	12/16/02 Date

STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

ACKNOWLEDGMENTS

After struggling through a Ph.D. program for many years and finally reaching the end of the journey, there are several people whom I want to thank for their tireless work on my behalf. Without their faith and support, none of this would have been possible. Foremost among these is, of course, my wife Terese. Her support has helped me weather difficulties and doubts that would have been much more formidable without her encouragement. Likewise, the support of my parents has been invaluable in this endeavor, as has that of my good friends Scott Watterson and Brady Montz.

Needless to say, a number of current and former members of the University of Arizona faculty were also instrumental in my completing this degree. My advisor, Rick Schlichting, has been an outstanding mentor in the process of learning to perform independent research. The value of his patient guidance has been immeasurable, as has that of three others: Larry Peterson, John Hartman, and Matti Hiltunen.

I'd also like to thank several outstanding staff members at the University of Arizona. Wendy Schwartz and Sonia Economou in particular provided invaluable help as graduate advisors in navigating the various Ph.D. program requirements. In addition, I've had the help of a truly outstanding computer support staff, in the persons of John Luiten, Phil Kaslo, John Cropper, Cliff Hathaway, Jim Davis, and Sandy Miller.

Finally, I would like to thank the National Science Foundation for their support of my research under grants ANI-9979438 and CDA-9500991 and the Defense Advanced Research Projects Agency for their support under grant N66001-97-C-8518.

TABLE OF CONTENTS

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

TABLE OF CONTENTS—*Continued*

LIST OF FIGURES

LIST OF FIGURES—*Continued*

LIST OF TABLES

ABSTRACT

Adaptation is an increasing important attribute for software that must operate well in changing environments, such as those encountered by mobile devices connected by wireless networks. However, adaptive software can be difficult to design, implement, and build, especially in systems with multiple adaptable components on multiple machines. A key challenge in such systems is coordinating adaptation across components, whether these components on located on the same or different machines. Without such coordination, for example, components may adapt in inconsistent or incompatible ways, leading to instability or poor performance. In addition, customizing adaptation policies to match the demands of the system and constructing testbeds are also difficult.

This dissertation describes Cholla, a framework for implementing adaptation in configurable networked software. Cholla addresses the challenges of inter-component coordination on a single machine and can be used along with existing techniques to implement coordination across machines. In addition, Cholla extends the benefits of configurable software to adaptation by allowing the policies that control adaptation to be constructed in a configurable manner. This allows the control logic to be analyzed, customized, and composed in ways that would be difficult at best in other systems. A prototype implementation of Cholla that uses Cactus, a system for building highly configurable network protocols and services, is also presented.

Two example applications are presented to demonstrate the effectiveness of Cholla: a multimedia transmission system and a configurable proxy for wireless networks. Both applications use CTP, a Cactus-based configurable transport protocol, and are structured in such a way that Cholla controls adaptive behavior in both CTP and between CTP and the application. Experimental results show that approach is the effective, especially in cases where adaptation mechanisms are limited and system behavior is very sensitive to adaptation choices.

Finally, this dissertation describes a WaveLAN emulator that allows the testing of adaptable software for wireless systems without constructing a complete hardware testbed. While the emulator can only provide accurate results under light network loads, it is nonetheless useful for emulating the dynamic nature of connectivity in low latency wireless networks.

CHAPTER 1
INTRODUCTION

Modern computer software, particularly software for networked computers, must increasingly deal with changing system conditions. For example, normal desktop machines must deal with variations in the quality of their network connection as the amount of congestion in the network changes. Variations in system conditions are even more severe in mobile and wireless devices because these devices may have to deal with issues such as radio interference, changes in location, and changes in hardware capabilities due to power conservation measures. This dissertation focuses on building frameworks and tools that allow software to deal with such dynamic environments.

Adaptation is a technique that allows computer software to deal with changing operating conditions. In adaptable software, a system changes dynamically in response to changes in the environment to optimize its behavior. Adaptation may be performed by modifying parameters that affect system behavior or by making larger-scale changes, such as completely changing the algorithm used to perform certain functions. Adaptation has a long history of use in networking and system software, including adaptive retransmission timeouts, adaptive CPU scheduling, and adaptive congestion control.

Implementing adaptive behavior can be difficult, especially in systems with multiple adaptable components distributed among multiple cooperating networked machines. An adaptable system may, for example, need to coordinate adaptation between multiple machines. Without such *inter-host coordination*, different hosts may adapt in an inconsistent manner, leading to incompatible software configurations on different machines, poor performance, over-adaptation, or instability. In addition, the policy that controls adaptation must be sensitive not only to changes in resource availability, but also to the demands of the application and the needs of the user. Other difficulties also exist on these systems,

such as how to construct appropriate adaptation policies and how to deal with inconsistent views of the current state of the system.

Similar issues are also present on a single machine, especially when using *highly configurable software*. Configurable software is constructed from sets of replaceable modules, each of which implements a portion of the behavior of the entire system. This ability to choose the appropriate functionality makes it easier to customize the functionality of the system for different situations. Such software has advantages when it comes to adaptation, such as making it easier to implement algorithmic adaptations by simply replacing one module with another that implements an alternative algorithm. However, there are also challenges to realizing adaptation in configurable software, such as how to structure adaptation policies in a configurable manner and how to perform *inter-component coordination*. Without inter-component coordination, for example, multiple adaptable components may adapt in inconsistent or incompatible ways, potentially hurting system performance.

This dissertation presents Cholla, a framework for implementing adaptation in configurable networked software. Cholla addresses the challenges of inter-component coordination on a single host and can also be used along with existing techniques to implement inter-host coordination. In addition, Cholla extends the benefits of configurable software to adaptation by allowing the policies that control adaptation to be constructed in a configurable manner. This dissertation also describes a highly configurable network protocol in which adaptation is useful, and an approach to wireless network emulation that can be used to study how adaptable software responds to wireless networking conditions.

1.1 Dynamic Environments

1.1.1 Overview

Until recently, computers and computer software have been designed for a single or handful of related tasks in a single environment that rarely changes. Instead of using one computer system in different ways in multiple locations, people have traditionally used different com-

puters and software for different tasks, even if the tasks are related. The capabilities, roles, and resources of a computer and its software did not change dramatically day-to-day or moment-to-moment.

This is no longer true—modern computers operate in dynamic conditions that can vary dramatically from moment to moment. The proliferation of network-based applications that have to share a changing network with computers all over the world is just one example of such changes. These changes are even more dramatic in mobile and wireless systems that can be physically moved while operating. This presents difficult challenges to software design. In this section, we elaborate on the type of changing operating conditions in which computers and computer software must operate. This includes changes both in the hardware resources on which the software runs, and in the demands of the applications and users.

1.1.2 A Canonical Example

A motivating example of a system that might have to deal with changing conditions is an application that sends video over a network. Figure 1.1 shows how such a system might be structured. In this system, a video application obtains video data from one of a variety of possible sources (e.g., a file or a camera), compresses the video for transmission, and then uses a network transport protocol to send video over a network to another machine. This machine receives the video, and then uncompresses it and sends it either to the screen for display or to a file for archival.

The software on each machine, as in most machines, consists of two separate parts: application software and system software. Application software is the software that most people are familiar with, and includes programs such as word processing programs, web browsers, and calendar programs. In this case, the video acquisition, encoding, decoding, and display software comprises the application software. System software, on the other hand, is the underlying support software that runs the computer and provides services to

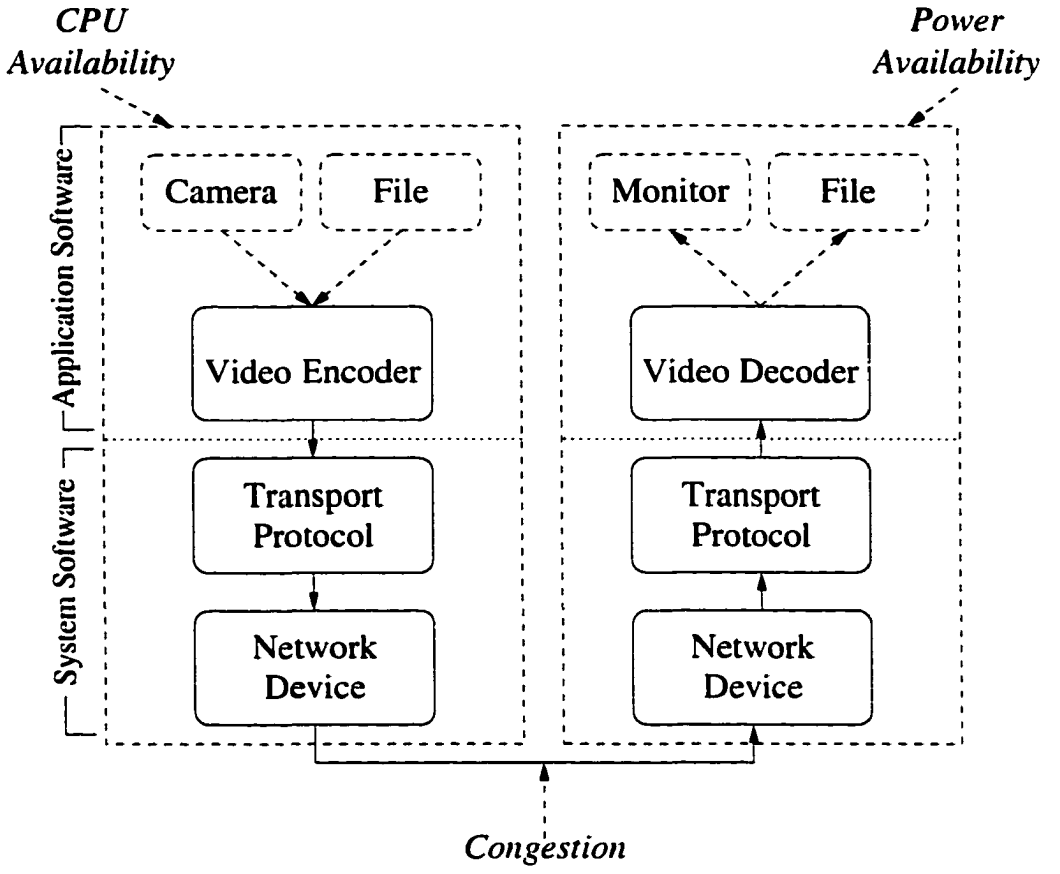


FIGURE 1.1. A Video Transmission System

applications that allow them to interact with the computer hardware and with other programs. System software includes both the operating system and *middleware*, software that sits between the operating system and applications. In figure 1.1, the transport protocol and the software that manages the network device are the primary examples of system software.

In this example, there are several sources of changing operating conditions, such as changes in available CPU cycles, available power, and network connection quality. When deciding how to deal with these changes, the system must also take into account the application's requirements and the user's preferences, and these factors could also change over the lifetime of the program. The following sections discuss each of these issues in turn.

1.1.3 Changing Hardware Resources

A number of different hardware resources can affect the performance of software. These include the number of available CPU cycles, the amount of free memory, how much power is left in the computer's battery, and the quality of the network connection. Each of these resources can change while a computer is running, particularly when they are being shared with other applications or computers.

CPU and Power Availability. Changes in the number of available CPU cycles can dramatically affect the behavior of an application. Such changes can happen for a variety of reasons, such as competition from other applications or increased demand for the CPU by the same application. For example, a system that could previously compress video in software when running the application shown in figure 1.1, may no longer be able to do so if fewer CPU cycles are available. The number of CPU cycles available for encoding video could change if, as another example, the application started transmitting audio as well as video, placing additional load on the system.

In mobile systems, measures designed to save battery power could also cause dramatic changes in hardware resource availability. It is not uncommon for laptops to reduce CPU speed by 3/4 when running on battery to conserve power, for example. In such cases, portions of memory could also be disabled or the computer screen dimmed. Applications may also directly change their behavior so that the system will have enough battery power to complete its tasks.

Network Quality. Another hardware resource that can change is network availability, where connection quality can vary depending on the type of connection and how many people are sharing the network, for example. Typically, connection quality is described in terms of bandwidth, latency, and reliability. Bandwidth, the rate at which data can be sent and received on the connection, can vary from below nine thousand bits per second (9 Kpbs) on a wireless telephone link, to upward of one hundred million bits per second (100

Mpbs) on high-quality wireline links. Similarly, latency, the amount of time it takes for a single byte to travel from the transmitter to the receiver, can vary from on the order of a few microseconds for fast local links to several seconds for international satellite connections. Finally, the reliability of a network connection describes how frequently the network drops or corrupts a piece of data sent over it.

In addition to the variations present in different types of networking hardware, the quality of an individual network link can also vary over time. One of most common sources of variation is *congestion*, which refers to the contention for network resources that occurs when multiple computers compete for a single shared network resource such as a router or backbone link. Congestion can also occur when a single computer attempts to use more network resources than are available. In the case of a video application like that shown in figure 1.1, congestion can limit the amount of bandwidth available in the network, preventing a high-quality video from being sent. Congestion can also cause latency to increase as packets are queued at routers, and reliability to decrease as these queues overflow and packets are dropped by the network. In wireless networks, radio interference from sources such as microwave ovens and wireless telephones can also cause changes in bandwidth and reliability.

Network quality can vary even more dramatically in mobile systems. In addition to the problems of network congestion and radio interference common in the wireless networks used by mobile devices, mobility can cause additional variations. As a mobile host moves, the strength of the radio signal it receives varies, which affects the quality of the connection. In addition, a mobile host may have to deal with hand-offs between multiple wireless networks when it leaves the area covered by one network and enters an area covered by a different network. Such hand-offs may change a mobile device's network connection from a high-quality office link to a medium-quality home link to a low-quality cellular telephone link. A laptop also may occasionally lose network access altogether and become disconnected.

In many cases, different types of network protocols are needed to deal with each of these situations. How these protocols should operate depends on the current bandwidth, latency, and reliability characteristics of the network being used. In systems that operate in dynamic environments, system software must be able to provide services to applications that are appropriate for the underlying hardware environment even as it changes.

1.1.4 Changing User Requirements

Other sources of changing operating conditions include the semantics of the application being run and the preferences of the user. While these constraints may not usually change as quickly as those in hardware environments, software must be prepared to deal with these sources of dynamic behavior.

Application Semantics. When computers run applications that are relatively similar and in an environment that is relatively static, system software only needs to supply a few types of services to applications. However, as operating conditions and application demands change, they place greater demands on system software for more diverse and flexible services. One of the most obvious examples of this is the different kinds of network protocols available on most operating systems.

There are many different types of semantics that can be provided by network protocols to applications, including completely reliable in-order data delivery [Pos81], unreliable real-time delivery for video applications [SCFJ96], and protocols that can provide multiple types of service [WHS01, DHLB98]. The type of service needed by a particular application depends on the type of data the application is transferring, and how that data will be used. These needs can change between invocations of the application, or while the application is running.

For example, if an application such as the one described in section 1.1.2 is transmitting video for real-time display on a screen, a lossy multimedia protocol might be appropriate. Such a protocol trades away complete reliability for the ability to deliver data in a timely

fashion. However, if the user decided to stop watching the show and archive the remaining video for later perusal, a different type of service may be more appropriate given the change in application semantics from real-time to non-real-time. By changing to a reliable networking service, the copy of the video archived for later display may be of much higher quality than if an unreliable protocol continued to be used. Even seemingly small variations in a single application such as this can place markedly different demands on the types of services that system software must provide.

User Preferences. Likewise, different users of a piece of software may have different preferences for how that software should behave. For example, when compressing video for transmission as described in section 1.1.2, there are several different ways to reduce the amount of space required. Video data consists of a sequence of pictures called *frames* that are played back quickly in order to reproduce the illusion of motion. Two different ways to reduce the amount of data are to either send fewer frames per second (reduce the *frame rate*), which may make the action appear choppy, or to send pictures that are of lower quality.

Depending on the type of video being watched, the user may prefer to use a different strategy to compress the data. If the user is watching a travelogue, for example, they may be willing to tolerate less smooth motion, as long as the pictures are very clear. On the other hand, if the video contains large amounts of important motion such as a sporting event, then the smoothness of the motion may be more important than the clarity of any individual picture. These preferences could also change if the type of video the user is watching changes. The net effect is that the appropriate method of reducing the size of the video feed depends both on the user and the specific situation.

as opposed to simply changing a parameter inside a particular algorithm. This might correspond to changing the algorithm used to compress video from a lossless algorithm to a lossy one.

Algorithmic adaptations are generally more drastic than value adaptations, with more potential to respond to large-scale changes in the system environment. However, these adaptations are also generally more difficult to implement and control than value adaptations. Implementing value adaptation requires simply changing a parameter that frequently has only localized effects. Changing algorithms, on the other hand, may require setup information to be exchanged with a remote computer, which can be time-consuming. It may also require structuring software in a way that makes it possible to dramatically change how a significant portion of a running program behaves.

Creating policies that control adaptive mechanisms can also be difficult, since they may need to take into account all of the various factors described in section 1.1. For example, the appropriate policy for detecting and adapting to network congestion may depend on whether the computer is on a wired or wireless network, while the appropriate policy for controlling video rate and quality may depend on both the application semantics and the preferences of the user. In addition, adaptation policies may also need to take into account other factors. For example, if the system spans several computers, the appropriate adaptation decision may be based not only on local information, but on the state of all computers in the system. Such policies may also need to take into account fairness to other adaptation policies on remote machines with which the local system competes. Constructing adaptation policies that take into account all of these factors and that are also appropriate and responsive to changes in the system environment is a significant challenge.

1.2.2 Adaptation in Configurable Software

Configurable software is software that is built to allow its behavior to be customized depending on the situation in which it is expected to operate. In configurable software sys-

tems, a program is built from *components*, each of which implements a portion of the functionality of the entire system. In some cases, multiple components may exist that are capable of performing the same task in different ways. This allows the system to be constructed by selecting those components most appropriate for the current situation. Configurable software techniques are commonly used when constructing complex software such as network protocol stacks like System V STREAMS [Rit84] and the *x*-kernel [HP91], and object-based middleware like CORBA [OMG98]. Configurability helps deal with the complex demands placed on such systems.

For similar reasons, configurable software can be an ideal platform to use for implementing adaptation. Configurability makes it easier to implement algorithmic adaptations, especially if each module in the configurable system corresponds to a different algorithm. In this case, algorithmic adaptations can be realized by simply replacing one module in the running system with another. Configurability can also be a useful way to deal with the problem of constructing appropriate adaptation policies for different systems.

Unfortunately, implementing adaptation in a system with multiple adaptable components, as is often found in distributed and networked systems, is especially difficult. If the various adaptations can interact with each other, additional mechanisms may need to be added to ensure that the adaptations in various components are synchronized. In addition, the policies that govern how each component adapts may have to take into account the potential behavior of all other adaptable components in the system, some of which may not even be on the same machine.

Coordination of multiple adaptable components is a key challenge when building systems that combine adaptation and configurability. Without *inter-component coordination*, each component adapts separately to a changing situation and the system could overreact to changes or react in an inconsistent or potentially unstable manner. Implementing inter-component coordination requires both mechanisms for synchronizing adaptation, and policies that take into account the opportunities of coordinating multiple adaptive actions.

In distributed systems where adaptive components are spread across multiple machines, the problems are even more severe. Components in such systems do not necessarily have a consistent view of the state of the system, and decisions cannot necessarily be made in a centralized manner. In addition, adaptation at multiple points in the system may need to be synchronized to prevent incompatibilities between different portions of the system. Without general support for solving these problems of *inter-host coordination*, adaptation in distributed systems can be very difficult to implement.

Configurable software techniques can also be used to address some of the issues associated with constructing appropriate adaptation policies. These policies must take into account any necessary coordination action required in a system with multiple adaptable components. In addition, the appropriate way to adapt may depend on the hardware, application, and user. For example, if the amount of video compression used in the application in figure 1.1 is going to be changed, the way in which it is performed should take into account the current preferences of the user watching the video. Likewise, if an application is running on a wireless network, the appropriate policy for adapting to network congestion might be different than the policy used if the network were a relatively reliable wired network.

Given these contraints, constructing adaptation policies in a modular fashion would allow them to be customized to do necessary coordination and to take into account the demands of the hardware, application, and user. This could be done by augmenting or replacing policy components with ones more appropriate to the system operating environment. It is not necessarily clear, however, how to construct adaptation policies so that they can be composed as needed in this fashion.

1.2.3 Environments for Evaluation and Testing

Evaluating and testing adaptable systems, or indeed any system that is intended to operate in a dynamic environment, can be quite difficult. Without a means to evaluate software

in situations similar to where the software is expected to operate, it can be difficult or impossible to study how different configurations and adaptations might behave. Doing so requires being able to control and reproduce the dynamic conditions in which the system is expected to operate. Such operating environments can be large, unwieldy, or expensive to construct under controlled laboratory conditions, however.

One case where constructing testbeds for dynamic environments is especially difficult is for mobile systems with wireless network connections. The main source of difficulty is that the quality of a wireless network connection normally depends on the location of all of the machines that are communicating, and varies as these machines move. Deploying a large-scale hardware testbed for wireless networks would require hardware for controlling and changing the location of a large number of computers, and controlling for sources of outside interference such as microwave ovens. Such a testbed would be prohibitively expensive.

Network simulation is one commonly used alternative to conducting tests using actual hardware. In a simulation, a computer program such as the `ns` network simulator simulates the behavior of the network on which the software would normally run [NS]. Typically, simplified versions of the program that model the networking portion of program behavior are run on the simulator. For programs with significant non-networking components, however, this may not be a feasible approach, particularly if there are complex interactions between the networking components and other components that may not be trivial to simulate. Because of this, tools in the middle ground between simulation and complete hardware environments are needed to allow for evaluation of complex software on mobile wireless systems.

1.3 The Cholla Approach to Building Adaptable Software

This dissertation describes *Cholla*, a framework for constructing adaptable configurable software that can operate in diverse system environments. Cholla provides facilities for im-

- Design of a framework that supports coordinated adaptation and composable control policies across multiple software components.
- Implementation of a prototype of this framework that runs on Linux.
- Evaluation of this framework for implementing control and coordination in three different applications: a configurable transport protocol called CTP, a multimedia transmission application, and a proxy for wireless networks.
- Design, implementation, and experimental evaluation of a wireless network emulator.

When taken together, these contributions illustrate Cholla's usefulness as a framework for building adaptable composable software for dynamic environments.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 reviews other research that addresses the issues described above. Chapter 3 then describes the overall Cholla architecture for constructing adaptable composable systems, with particular emphasis on the techniques used in Cholla for composing and coordinating adaptation policies. A prototype implementation of this architecture is described in chapter 4. Chapter 5 describes the control of adaptation in a configurable transport protocol called CTP and in doing so also describes a number of enhancements to CTP. This is followed by chapter 6, which illustrates the use of Cholla for performing composition and coordination in two different applications, a multimedia transmission application and a wireless network proxy. Chapter 7 describes work on a network emulator that can be used to evaluate software for mobile wireless systems. Finally, chapter 8 summarizes the results in this dissertation and offers areas of future work.

CHAPTER 2
RELATED WORK

A great deal of research has been done on adaptable software, including a variety of systems that use adaptive mechanisms and policies, and a number of different frameworks that attempt to provide useful infrastructure for building adaptable software. Much of this work has dealt with adaptation in multimedia applications and system software, where configurable software techniques are commonly used; as a result, some of this work covers adaptation in configurable software. A few of these systems include limited support for coordination of multiple adaptable components.

This chapter describes work that pertains to adaptable software, adaptation in configurable software, and environments for testing adaptable software. Specifically, section 2.1 describes a number of adaptable systems, both recent and historical, including several frameworks that provide general features useful for constructing adaptable software. Because configurable network systems are an important and challenging environment in which to perform adaptation, section 2.2 describes systems for building highly configurable software. Section 2.3 then discusses a number of systems for performing adaptation in various types of configurable software, including several systems that include limited support for coordinating multiple adaptable components. Section 2.4 describes a variety of simulation and emulation environments that can be used to test adaptable software. Finally, section 2.5 summarizes the approaches others have taken in relation to the challenges described in chapter 1.

2.1 Adaptable Software

2.1.1 Overview

Adaptation is a very powerful technique for dealing with dynamic environments, such as those encountered in networked systems. One of the first places adaptation was used in computer networking was for handling retransmission timeouts after collisions in the media access protocol for the ALOHA radio network [KT75]. A similar adaptive retransmission strategy is used in TCP, the Transmission Control Protocol [Pos81]. TCP also includes techniques for adaptive flow control and congestion control [Jac88].

More recently, the proliferation of multimedia applications has spurred further research into adaptation in that environment. Multimedia applications can frequently transmit data at a range of different qualities, and so adaptation in multimedia applications focuses on controlling the *quality of service* (QoS) provided. Campbell *et. al* constructed one of the first systems to provide an architecture for controlling QoS [CCG+93], and a wide range of quality-of service adaptation frameworks have been created since then. The advent of mobile and wireless systems has added additional aspects to both general adaptation problems and adaptation for QoS [Kat94].

In this section, we describe a number of the more recent systems designed to support adaptation. This includes both adaptation in general, adaptation for QoS, and adaptation in mobile systems. We also describe a number of different network protocols that provide adaptive services. Research on adaptation in configurable systems is described separately in section 2.3.

2.1.2 Odyssey

Odyssey is a system designed to give applications on mobile hosts adaptive access to information based on the quality of that access [NSN+97]. Access to data is made through *wardens*, which encapsulate type-specific functionality such as how to access the data and

this approach, the target task is controlled using an *adaptation task* that received state information from an *observation task*. Adaptations can either be *qualitative adaptations*, which modify a single parameter, or *reconfigurations* of the task graph. In addition to deriving a framework for analytical PID control of quality-of-service systems, this research also notes that fuzzy control [Wan97] could be a useful means of controlling adaptable systems.

2.1.5 Fugue

Fugue is a multimedia transmission system that features a 3-controller approach to controlling video encoding and transmission parameters in a wireless environment [CNW01]. In Fugue, two different components are controlled: a video encoder and a wireless transmitter that can send at configurable data rates and transmission powers. Each controller controls adaptation in a different part of the system and works at a different timescale, with the controllers at coarser timescales making decisions that are used by the controllers at finer timescales. By breaking the control decisions into relatively independent pieces that operate at different timescales, Fugue obviates the need for more direct coordination of the various controllers.

2.1.6 Adaptation of Tunable Applications

Several frameworks have been built to support *tunable distributed applications*. These applications export a variety of potential configurations to the framework, which then configures the application based on the system environment. Brandt *et al.* developed a dynamic QoS manager (DQM) in which applications describe *execution levels* to the resource manager [BNBH98a, BNBH98b]. These execution levels specify a relation between the amount of CPU given to the application and the benefit the application receives. The QoS manager then sets the execution level for applications it manages in an effort to maximize the total benefit to the system. Similarly, applications in the Active Harmony framework provide *utilization profiles* to the framework [KHP99]. These profiles describe how the application

behaves under different resource constraints in different configurations. As resource availability changes, Active Harmony exploits these profiles to determine in which configuration to run the application. In contrast, Chang and Karamchetti have designed a framework which does not require applications to present utilization profiles [CK01]. Their framework uses simulated offline execution in a *virtual execution environment* to derive information about different application configurations.

2.1.7 Synthesis

Synthesis was one of the first operating systems to make extensive use of adaptation [PMI88, Mas92]. Synthesis includes a number of different adaptive techniques, including feedback paths for adaptive fine-grained scheduling, and dynamic code generation. For example, the scheduler in Synthesis dynamically adapts the amount of CPU time given to processes based upon the amount of work remaining in the process's job queue. This allows a pipeline of processes cooperating to accomplish one task to operate efficiently.

Synthesis also dynamically generates code to speed up commonly used system functions. In most operating systems, functions in the trusted operating system kernel include a large amount of argument checking, since many of these functions can be called from both untrusted and trusted locations. However, when a trusted caller invokes such a function, argument checking may be redundant with the checks already performed by the trusted caller. Synthesis avoids this problem by adaptively specializing general-purpose functions not to include unnecessary argument checks. This avoids system overhead and improves the performance of the entire system, since only commonly called code paths are specialized in this manner.

2.1.8 Adaptable Network Protocols

A large number of different network protocols have been written that can adapt to changing conditions in computer networks. These protocols each aim to provide a particular kind

of service to applications, while adapting to changes in network conditions. The most common network conditions requiring adaptation are changes in the amount of congestion, end-to-end latency, or lossiness.

TCP congestion control is the best-known algorithm for adapting to changes in network congestion [Jac88]. A number of other congestion control techniques have been explored, including a technique using packet-pairs to detect congestion [Kes91], techniques useful for multimedia applications [SCFJ96, MJV96, CPW98], and techniques for congestion control when using rate-based protocols [RHE99, FHPW00, YL00]. Other examples of adaptive network protocol techniques abound, including techniques for adaptive retransmission in the ALOHA network [MST80], for media access in CSMA/CD media access protocols [Dim87], for supporting multiple reliability semantics in one protocol session [DHLB98], and for adaptive RPC in mobile systems [DFBC96].

2.1.9 Adaptation in Other Systems

Adaptation has also been used in other types of system software. For example, the Pablo performance analysis environment advocates using adaptation to steer scientific applications [REM+96]. Pablo uses fuzzy control rules to decide how and when to prefetch file system data for grid-based applications. The ISTORE system, a back-end storage architecture, uses a self-monitoring approach where the system collects statistics about its current behavior to facilitate adaptation [BOK+99]. Adaptations in ISTORE are controlled through triggers that fire under specified conditions and rules that control how to adapt when different triggers fire.

Adaptation has also been advocated for addressing problems stemming from computer mobility. Puppeteer, for example, provides mobile-specific adaptation in component-based systems built using the Microsoft Component Object Model, such as Microsoft PowerPoint and Internet Explorer [dLWZ01]. Other groups have explored using proxies in wireless

networks to adapt media quality in real-time streams and HTTP requests [FGCB98, MP99, SP02].

2.2 Highly Configurable Software

2.2.1 Overview

Many of the frameworks for building configurable software are designed for constructing network protocol software. Network protocols are conventions for how information is exchanged over a computer network to carry out a particular task. For example, the Ethernet protocol describes what information will be contained in a *packet* of Ethernet information. Likewise, the Internet Protocol (IP) describes how data will be addressed and routed through the Internet, while the Transmission Control Protocol (TCP) describes a standard for exchanging data reliably between computers.

One protocol may also depend on other protocols. For example, HTTP, the protocol for transporting web pages in the Internet, uses TCP for reliable data transmission. TCP likewise uses IP for addressing, locating, and routing its packets through the Internet, while IP may use one of a variety of low-level protocols such as the Ethernet protocol for transmitting data over the wire. In a system such as this, protocols are *layered* on top of each other, as shown in figure 2.1. An upper layer uses the services provided by the next lowest layers, which in turn uses the services of the protocols below it. If different functionality is needed, then one of these layers could be replaced by a protocol that provided the appropriate functionality.

Because of the layered nature in which network protocols are composed, the network subsystems that implement these protocols have traditionally been built in a hierarchical fashion. Several frameworks described below implement just such layered protocol processing. This section also decribes other frameworks that allow more flexible means of composing a networked or distributed system out of composable software modules.

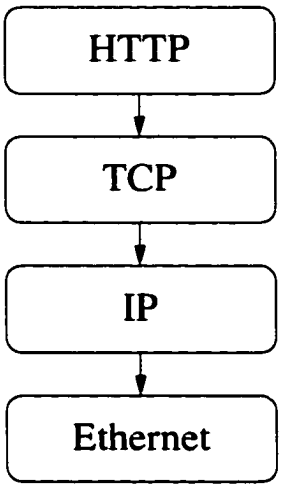


FIGURE 2.1. Layered Network Protocols

2.2.2 STREAMS

System V STREAMS is a hierarchical modular protocol architecture designed and built for the UNIX System V operating system and its derivatives [Rit84]. Originally designed for terminal I/O applications, an *I/O stream* is a sequence of modules that each support a UNIX-like read/write interface. This uniform read/write interface allows modules to be exchanged and used interchangeably. Each module also has input and output queues for messages currently in that module. The input queue contains messages that the module is about to process, while the output queue contains messages that the module has finished processing and that need to be sent to the next module. In network protocol applications, each STREAMS module corresponds to a protocol implementation in a protocol stack, such as the HTTP, TCP, or IP protocols shown in figure 2.1.

2.2.3 *x*-kernel

The *x*-kernel is a framework for hierarchically composing protocol modules [HMPT89, HPOA89, PHOR90, HP91]. In the *x*-kernel, protocol modules are layered on top of each other, with protocols higher in the hierarchy using services provided by protocols lower

in the hierarchy. These protocols interact with each other through a *Uniform Protocol Interface* (UPI).

The UPI encapsulates the common operations used by layered network protocols, as well as the data structures (such as *x*-kernel messages [Mos97]) that these operations should use. For example, when a lower-level protocol has received a message from the network or a protocol below it and wishes to deliver that message to a higher-level protocol, it calls the higher level protocol's `pop()` operation, passing the message to be delivered as an argument. Similarly, a higher-level protocol calls a lower-level protocol's `push()` function to send a message using that protocol. The UPI also includes operations for implementing remote procedure call semantics and a general control operation for dealing with other protocol-specific operations. This standard interface allows protocols to be added, removed, and replaced easily because the interactions between each protocol module are always those defined by the UPI. The techniques used in the *x*-kernel have also been used for structuring configurable operating systems such as Scout [MMO+94, MP96].

2.2.4 Horus

Like the *x*-kernel, the Horus framework is built around hierarchical composition of network protocols using a well-defined interface, the Horus Common Protocol Interface (HCPI) [RBF+95]. Unlike the *x*-kernel, Horus was designed for the implementation of group communication protocols, which frequently have more complex semantics than other protocols. Because of this, the HCPI includes not only operations similar to those in the *x*-kernel UPI, but also operations for manipulating group membership.

2.2.5 Coyote and Cactus

Coyote [BHSC98] and Cactus [HS00] are also frameworks for implementing complex network protocols. Both systems use an *x*-kernel-like interface between protocols, and in fact some implementations of Coyote and Cactus use portions of the *x*-kernel framework

for hierarchical composition. Unlike the *x*-kernel and Horus, however, Coyote and Cactus allow for more complex and flexible forms of composition centered around the construction of *composite protocols*. These protocols can then be layered together using standard techniques like those in the *x*-kernel.

In Coyote and Cactus, composite protocols are constructed by combining smaller components called *microprotocols*. Each microprotocol implements only a small portion of the functionality of a full protocol, but when combined together with other microprotocols, the resulting composite protocol implements a fully-featured protocol layer. The decomposition of composite protocols into microprotocols allows composite protocols to be customized to a much finer granularity than is possible with standard layered frameworks.

Microprotocols interact with each other through shared data structures, a dynamic message abstraction [HWS01], and an event mechanism. Messages are processed by the microprotocols in a composite protocol through invocation of *event handlers*. Microprotocols bind event handlers to well-known events such as `MessageFromNet`, and these handlers are invoked when, for example, a new message for the composite protocol arrives from the network. Unlike strictly hierarchical message handling in systems like Horus and the *x*-kernel, Cactus's use of event handlers for message delivery allows messages to be processed inside a composite protocol in a more flexible manner when necessary.

CTP is an example of a composite protocol built using Cactus [WHS01]. CTP is a configurable and extensible Cactus transport protocol that can be customized based on the needs of the application using the protocol. It includes microprotocols for implementing many standard transport protocol features, including positive acknowledgments, forward error correction, segmentation and reassembly, flow control, congestion control, and a variety of other features.

2.3.1 ARC

ARC implements control of an adaptive video source and transmission parameters on a wireless network [vDLS00]. ARC makes QoS decisions using a "bottom-up" negotiation between layers in a multilayer system like those commonly used for implementing networking software. In ARC, upper-layer protocols state their *requirements* to lower-layer protocols. Lower layer protocols then return multiple *adaptation choices* to the upper-layer protocol, which makes the final adaptation decision. These returned choices may themselves be the result of negotiation between the lower-layer protocol and other protocols even lower in the protocol stack. If the available choices are inadequate, the upper layer can change its requirements and repeat the entire negotiation process. This iterative negotiation strategy provides a level of composability and coordination between adaptive layers, although it requires upper levels to be strictly dependent on the services provided by the lower layers. One potential limitation is that this iterative negotiation process must be performed every time the state of the system changes, and could potentially be computationally expensive.

2.3.2 SEDA

SEDA (Staged, Event-Driven Architecture) is a framework for building applications that degrade gracefully under heavy load by adapting the behavior of various phases of the system [WCB01]. Applications built using SEDA operate on data in a series of *stages*, each of which performs part of the task of processing a piece of data. This decomposition is similar in many ways to how messages are handled in stack-based protocol systems such as Scout or STREAMS. By monitoring how much data is waiting to be handled by any particular stage, the SEDA architecture can identify bottleneck processing points and shed load by adapting the behavior of the bottleneck stages.

code for detecting more complicated fault conditions, however, so that the system can adapt its behavior to use an algorithm that can tolerate such faults if necessary. This approach avoids much of the overhead of associated with high degrees of fault tolerance when such overhead is unnecessary, but still allows the system to continue functioning when such fault tolerance is necessary.

Such adaptation still requires system and protocol support for adapting the distributed system and for coordinating adaptation between all of the machines in the system. If this requires stopping the running system while the change happens, the performance penalty can be so large that it might not have been worth, for example, running without the fault-tolerance protocols to begin with. To address this problem, Chen has extended Cactus with a mechanism to allow the microprotocols being used in a running system to be changed without stopping the flow of packets [CHS01]. This process requires that every machine agree on the state of the system, run a deterministic function to decide what adaptation will be performed, and then execute several synchronization steps that effect the change. All of these steps are performed while packets continue to flow through the original protocol configuration. Thus, adaptation is coordinated between all of the machines in the system without disrupting the operation of the system.

2.3.5 QuO

The Quality Objects (QuO) framework [ZBS96] is a distributed object framework designed to allow adaptation in distributed object systems. QuO is centered around the principle of *separation of concerns*, where adaptive behavior is specified seperately from the rest of the system. QuO implements adaptive behavior primarily through *contracts*, which describe the QoS service required by a client, the different QoS levels available, and what actions to take when the QoS level changes. These contracts are specified using a Quality Definition Language (QDL). QuO has been used to implement adaptation in distribution of video from an unmanned aerial vehicle [KRL+01] and in fault-tolerant computing systems built using

Ensemble, under the name AQuA [CRS+98]. In addition, work is in progress on making adaptation behaviors and policies reusable by packaging them into *qoskets* [SLMP02]. Qoskets encapsulate the QuO objects associated with the adaptation, such as contracts, initialization code for these objects, and the interfaces that a qosket exposes.

2.3.6 The VINO Operating System

The VINO operating system is an extensible adaptable operating system in which applications can extend the kernel with application-specific code [SESS96, SS97]. To install an extension, an application downloads binary code into the kernel. The kernel protects itself from misbehaving extensions using software fault isolation and by running extensions as transactions. Software fault isolation [WLAG93] prevents extensions from modifying memory to which they should not have access. Running each invocation of an extension as a transaction prevents misbehaving extensions from leaving kernel data structures in inconsistent states.

VINO supports adaptation through *self-monitoring* and *in situ simulation*. Self-monitoring allows the system to monitor and analyze the current behavior of the running system, and is accomplished by collecting a database of statistics on the behavior of the system. *In situ* simulation allows the system to simulate how the different adaptations would affect the behavior of the system. The results of these simulations can then be used by the system to adapt its behavior for the current environment.

2.4 Network Simulation and Emulation

2.4.1 Overview

Because of the difficulties in constructing testbeds for networked systems, a number of different simulation and emulation environments have been developed. In a simulator, the entire system being tested exists only within the computer running the simulation. The

actions of every element of the system relevant to what is being studied are simulated, including the processor, operating system, and application. Because simulation of a complete system is normally computationally infeasible, simplified versions of systems are typically simulated instead of complete systems. For example, a simulation might use a simplified model of an application instead of the application itself. Of course, this may make the results of the simulation less accurate than desired, and it may also make it difficult to study interactions between different components in the system. Because of this, emulation is sometimes used instead of simulation.

In an network emulation environment, standard computers run complete operating systems and applications. Instead of communication over normal network devices, however, the computers instead communicate using an *emulator* that behaves like a different network than the one actually connecting the computers. In mobile and wireless networks where networking hardware can be expensive and controlling the motions of every machine in the system can be difficult, this can be particularly useful. Emulation allows the behavior of complex systems to be studied without the simplifying assumptions that must be made in simulation or the burdensome expense and management of full hardware testbeds.

2.4.2 NS

The `ns` network simulator is the defacto standard network simulator for studying most network protocols [NS]. `ns` is written in a combination of C++, which is used for low-level compute-intensive simulation tasks, and the OTcl scripting language, which is used for high-level configuration tasks. `ns` has also been extended by the CMU Monarch project to support a variety of wireless devices and protocols [CMU98]. As a simulator, `ns` includes support for a simplified subset of the Transmission Control Protocol (TCP) and several of its variants, along with a variety of other protocols. The applications that NS includes are relatively simple, including a simulated application that sends at a constant bit rate

and a simulated application that sends data as quickly as possible, similar to a file-transfer application.

NS also includes basic support for centralized network emulation. In this configuration, applications running on separate computers send their network traffic to a centralized NS simulator, which simulates the network effects on the traffic it receives and then forwards the data to the computer that should receive it. This configuration introduces a single bottleneck into the system, and no emulation is possible if the simulator cannot keep up with the load imposed by the multiple applications in the emulation. It also introduces additional latency into each connection, specifically, the time it takes the data to get from the sending machine to the emulation process and the time it takes to simulate the behavior of that packet. This behavior makes it inappropriate for emulating low-latency networks.

2.4.3 Maise and GloMoSim

Maise is a specialized modeling language for wireless network simulators developed as part of the Wireless Adaptive Mobile Information System (WAMIS) project at UCLA [SBK95]. Programs written in the Maise language specify the behavior of the system. This specification is then compiled into a parallel simulation that can run on a large number of machines. This allows for large simulations to be run quickly by using large parallel computers. GloMoSim is a component-based modeling library derived from Maise that allows wireless simulations to be setup quickly and easily by composing components in the GloMoSim library [ZRM98].

2.4.4 Kernel-based Network Emulation

Several systems have been built for performing emulation of various network characteristics by placing specialized code in the packet processing path in the network kernel. NISTnet [NIS], the Ohio Network Emulator (ONE) [ACO97], and Dummynet [Riz97a] all use kernel extensions to modify the behavior of received packets. These systems allow pack-

ets to be dropped at specified rates, additional latency to be added to connections, and for bandwidth and queuing restrictions to be placed on different links. These systems do not attempt to accurately replicate the behavior of any particular network interface or device. They do, however, allow the behavior of networked systems to be studied under different network congestion and latency conditions.

2.4.5 Wide-area Wireless Network Emulation

Some emulators, such as Seawind [KGM+01] and the network emulator portion of the ANSAware platform [DBCF95], attempt to replicate the behavior of specific network devices. The ANSAware emulator emulates three different wide-area wireless networks: the GSM cellular telephone system, a U.K. analog cellular service, and an analog private mobile radio system. In this emulator, UDP packets are intercepted and routed to a single centralized process that introduces delay and errors appropriate for the conditions of the wireless network. Similarly, Seawind normally uses a dedicated central machine running a *simulation process* to simulate the behavior of a wireless network. Seawind can also be configured to use multiple pipelined machines to simulate the interconnection of multiple wireless networks. Both emulation systems operate at the user level, are easily ported to different environments, and are targeted at emulating wireless networks with relatively low bandwidth and high latency characteristics.

2.4.6 Trace-based Emulation

An alternate approach to a centralized emulation process is the trace-based approach taken by Noble, *et al.* [NSNK97]. In centralized emulation systems such as those described in sections 2.4.2 and 2.4.5, the processing power of the central simulation process can be a bottleneck that prevents emulation of large-scale or high-speed networks. For example, the ANSAware emulator described in section 2.4.5 can only emulate networks that run

at approximately 9600 bits per second and can only support approximately 16 hosts, as opposed to the megabit speed and large-scale networks common today.

Trace-based emulation avoids these problems by guiding the behavior of the emulated network based upon traces of the behavior of a network that have been simulated using a simulator such as `ns` or GloMoSim. In a trace-based system, a simulation is run that models the behavior of the network using simulated applications and operating systems. A record of the results of this simulation is saved, and each computer participating in the emulation then replays the portion of the trace relevant to its network interface, modulating the behavior of packets that traverse that interface based on the results of the emulation. While this requires simulating the behavior of the system to be emulated, it does allow larger and faster systems to be emulated compared with other approaches.

2.5 Summary

The systems described in this chapter illustrate the many different approaches that have been taken for building and testing adaptive behavior, including adaptive behavior in configurable systems. Most of the approaches are either specific to a particular application domain or hierarchical in nature. Some systems also provide mechanisms for synchronizing adaptation on multiple machines in a distributed system. None of these approaches, however, addresses how to coordinate adaptation among multiple components in highly-configurable non-hierarchical systems such as Cactus. Only one system, the SWiFT toolkit, seeks to provide general support for constructing adaptation policies, but even it does not address coordination or decomposition of existing complex control policies.

Of the testing environments, the majority are centralized solutions that can only emulate restricted types of networks. The sole exception, trace-based solution, can only emulate networks that have previously been simulated. Moreover, because simulator-based solutions such as *ns* and GloMoSim use domain-specific languages, libraries, and infras-

tructure, the amount of implementation required to emulate a complex system can be significant.

In contrast, this dissertation describes an approach to coordination and composition of adaptation in both hierarchical and non-hierarchical systems. This approach allows coordination between multiple components in a highly configurable system and for decomposition and customization of adaptation policies. It also aims at allowing control decisions to be made and coordinated in distributed systems, where an iterative negotiation approach like that used in ARC would be infeasible. In addition, this dissertation describes an architecture for emulation that avoids some of the limitations of centralized and trace-based approaches. In the following chapters, we describe these contributions in detail, apply them to different example applications, and contrast our approach with those of closely related work.

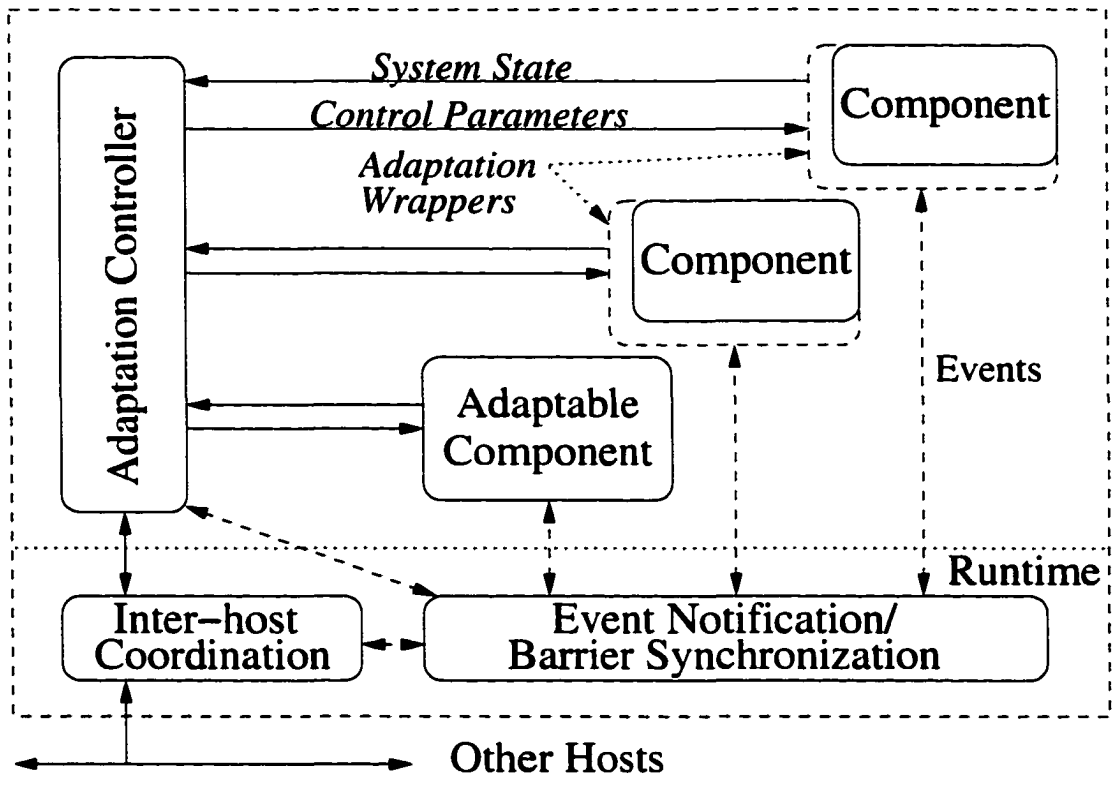


FIGURE 3.1. The Cholla Adaptation Architecture

ponent with the Cholla framework. Adaptable components in Cholla export *system state* and *control parameters* to adaptation controllers, which are responsible for making *adaptation policy* decisions. The system state exported by a component consists of measurements of the environment in which the system is operating that are relevant to that component. This state is used by controllers to determine how to adapt to the current environment. Control parameters, on the other hand, are exported by components and set by controllers to modify the behavior of adaptable components. These parameters may affect actions ranging from subtly altering the behavior of the component to enabling or disabling it completely. Almost any component containing an adaptable mechanism can be controlled by the Cholla framework by encapsulating it in an adaptation wrapper that attaches it to the Cholla controller and runtime system.

Adaptation controllers in Cholla are responsible for encapsulating the adaptation policies that control and coordinate adaptable components in Cholla. These controllers are designed to facilitate the composition of fine-grained adaptable components, to allow separation of adaptation policy from the mechanism that implements adaptation, and to support policies that require coordination of multiple adaptable components. Controllers interact with adaptable components in two ways: they monitor the state and set parameters exported from components, and they raise and handle events from the runtime's event notification mechanism. This is in contrast with most existing frameworks, where adaptive components are typically coarse grain, hard-coded policies are interwoven with adaptation mechanisms, and facilities for coordination are minimal.

The Cholla runtime system provides additional services necessary to support adaptable components and adaptation controllers, along with the elements necessary to tie these elements together into a coherent adaptation coordination architecture. These services, which are based largely upon existing techniques, include an event notification mechanism, a barrier synchronization primitive, and inter-host coordination protocols. Event-based notification is used for coordinating control flow in Cholla. This includes issues such as triggering the controller at the request of components and notifying components after the controller has made policy decisions. Barrier synchronization, which is integrated with the event notification service, is used to synchronize interacting adaptations so that coordinated policy decisions made in the controller are carried out simultaneously by the various adaptable components in the system. Finally, inter-host coordination protocols such as those used in Ensemble [RBH+98] and Cactus [CHS01] are used to collect remote state for making policy decisions and to coordinate algorithmic adaptations in distributed systems.

Figure 3.2 shows how the Cholla architecture can be used to control the video example presented in section 1.1.2. In this example, Cholla is responsible for controlling adaptation in a video encoder and a transport protocol. The components and the inter-host protocols gather information on the state of the system and network, and invoke the adaptation controller whenever an adaptation policy decision is needed, e.g., the receipt of a packet

and the amount of error correction used, potentially after synchronizing their actions using the runtime's barrier synchronization mechanism.

The Cholla architecture is targeted toward general adaptable systems. This dissertation, however, focuses on the issues of adaptation on a single machine, particularly the issues related to composing and coordinating adaptations in the context of frameworks for building highly configurable networking software. The rest of this chapter describes adaptation controllers and the Cholla runtime, both in general and in the context of the example given above, and how the controller and runtime interact to coordinate adaptation in adaptable components.

3.2 Adaptation Controllers

Adaptation controllers in Cholla contain the adaptation and coordination logic for a collection of adaptable components and are responsible for making the policy decisions that control how a collection of components adapts to changes in the environment. Controllers determine what adaptation is required by monitoring *input variables* that measure the current system state exported by adaptable components and initiate adaptation by modifying *output variables* that change control parameters exported by adaptable components. For value adaptations, the controller typically directly sets the parameters that control adaptation. For algorithmic adaptation, output variables could be a toggle indicating whether a particular component should be functioning, or a state variable in a component that in turn controls which other components are included in the running system. In figure 3.2, for example, adaptation wrappers added to the video encoder and transport protocol components export the current video data rate and information on network conditions to the adaptation controller. The controller monitors this state and changes the rate at which data is generated and sent according to the adaptation policy implemented by the controller.

Cholla controllers are constructed using a rule-based approach. In particular, controllers are created by selecting and composing together available *rule sets*, each of which imple-

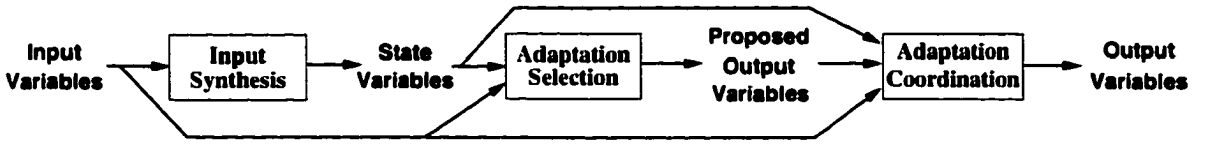


FIGURE 3.3. Controller Execution Phases.

ments a portion of a particular adaptation policy. The union of these rule sets determines the adaptation policy implemented by the controller. This allows adaptation policies to be customized and adapted to the different conditions described in chapter 1, including the needs of the hardware, application, and user, as well as the need to deal with any necessary policy coordination between adaptable components. Each rule set contains control logic in the form of rules that dictate how the values of the input variables map into values of the output variables. Section 3.3 describes the format of controller rules and how multiple rule sets are combined into a single control relation.

To simplify rule construction and composition, execution of adaptation controllers is divided into a sequence of several phases, shown in figure 3.3. Each phase is governed by its own rules and is composed from separate rule sets. The *input synthesis* phase assigns values to aspects of the system state that are difficult or impossible to measure directly, such as the degree of congestion in the network. It does this by combining inputs that provide a partial indication of the state, such as whether or not there has been a recent timeout. The *adaptation selection* phase uses these synthesized inputs along with standard system inputs to propose values for output variables. Finally, the *adaptation coordination* phase coordinates adaptations between components that were not necessarily designed to work together. This requires the addition of extra rules to the controller that implement the coordination policy. Section 3.4 gives more details on the roles of the various phases of the controller.

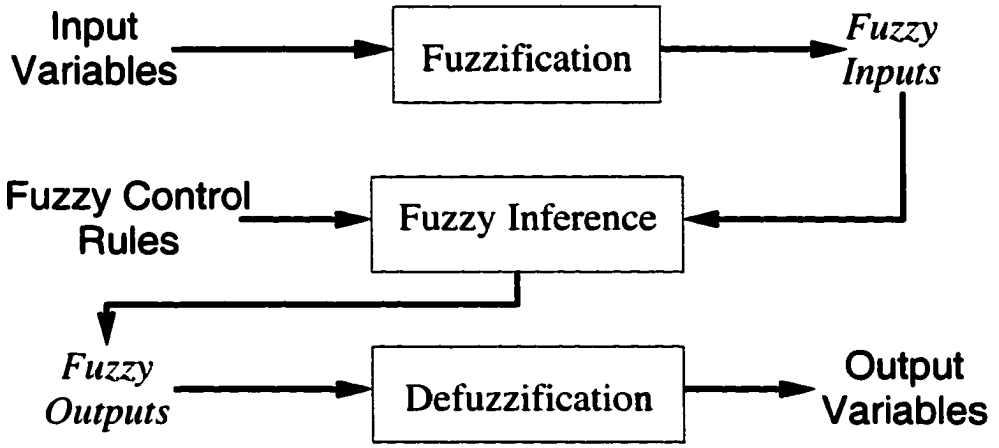


FIGURE 3.4. Phases of Fuzzy Control

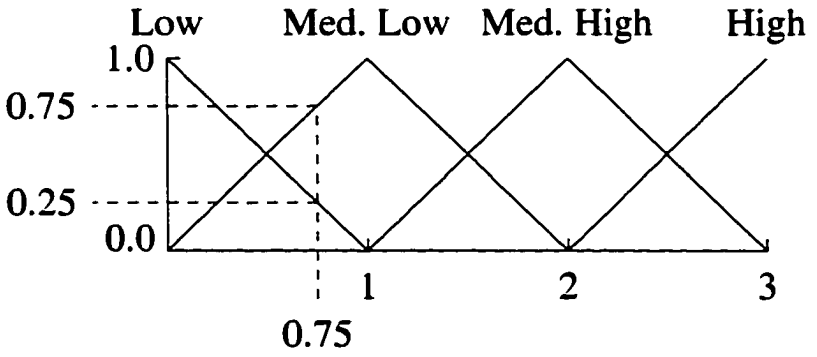


FIGURE 3.5. Fuzzification of 0.75 into (0.25 LOW, 0.75 MED. LOW)

used to perform this mapping, such as picking the point with the greatest membership value or some form of weighted average, such as the "center-of-gravity" of the fuzzy set.

Fuzzy rules are normally if-then inference statements that set the membership values of output fuzzy sets based upon the membership values of input fuzzy sets. The fuzzy set described above could be set by rules that relate system events to the level of congestion in the system. A simplified example of the fuzzy rules that might be used by a transport protocol for determining the amount of congestion in the system is shown in figure 3.6. The first rule in figure 3.6, for example, modifies the membership value of the "high" element of the *congestion* set to be the fuzzy union of the previous value of that element and the

if **RecentTimeout** is *true* then **Congestion** is *high*;
if **RecentDrop** is *true* then **Congestion** is *medium*;
if **RecentAck** is *true* then **Congestion** is *low*;

if **RoundTripTimeIncrease** is *high* then **Congestion** is *high*;
if **RoundTripTimeIncrease** is *medium* then **Congestion** is *medium*;
if **RoundTripTimeIncrease** is *low* then **Congestion** is *low*;

FIGURE 3.6. Example Fuzzy Control Rules

membership value of the "true" element of the *RecentTimeout* set. The union of two fuzzy set elements can be implemented in a variety of ways; two of the most common methods are using the maximum or the bounded sum of the two membership values. Note that inputs and rules can be added modularly; if a new measurement for determining system state is desired, it can be added as a new input variable, along with rules that describe the relationship between the input and the system state being measured.

In addition to if-then inference rules, it is also possible to define rules that perform standard mathematical operations on numerical values represented using fuzzy sets. The most general approach to implementing mathematical operations in fuzzy control rules defines versions of the standard mathematical operations that operate on fuzzy sets. Such an approach propagates the information in multi-valued inputs through the computation, allowing this information to be used by all steps in the decision-making process. Another more ad-hoc approach works by converting fuzzy sets to and from crisp numeric values as necessary to perform mathematical operations. In this case, a rule such as "if **Congestion** is *High* then **Window** ← **WindowIn** / 2" would be executed as follows:

1. Convert the **WindowIn** fuzzy set into a crisp value using defuzzification and perform the division by 2 on this crisp value.

2. Convert this crisp value into a result fuzzy set using fuzzification.

3. Scale the set membership values of every element of this result set by the membership value of the *High* element of the **Congestion** set.

4. Union every element of the result set into the **Window** fuzzy set.

While such a process is somewhat ad-hoc and can lose set membership information through the intermediate fuzzification and defuzzification steps, it is relatively easy to implement. Such a process is used by the prototype described in chapter 4.

The use of fuzzy sets to represent input and output values, and fuzzy rules to implement composable versions of existing control heuristics has several benefits in the context of adaptation control:

- The fuzzy set representation and fuzzy inference provide a way to represent and make decisions using qualitative data.

- Fuzzy composition provides a structured way to compose multiple rules into a single control relation.

- Defuzzification provides a first step toward reconciling and coordinating conflicting control policies.

Other researchers have also noted advantages to using fuzzy control for software systems [LN99, REM+96]. There are also a number of excellent books on fuzzy control concepts [Wan97].

Fuzzy control is no panacea, however, and each control strategy must be carefully designed and experimentally evaluated. Furthermore, the interactions between different rule sets must be closely examined and tuned. This tuning, however, is only required for those rule sets that set the same output variables in the same phase of the controller. While coordination of multiple rule sets is currently done manually in Cholla, more general automated coordination methods are also being explored.

3.4 Rule Sets

The rules that are associated with the three execution phases govern how output variables are set to control component adaptations. These rules are divided into composable rule sets, each of which consists of a small number of rules that assign the value of a single variable or a closely related set of variables. For example, a rule set in the input synthesis phase dealing with network congestion might set one state variable corresponding to the congestion level and another corresponding to the change in the congestion since the last time the controller was executed.

Rule sets are the "replaceable modules" in the controller, encapsulating different policies for the conditions needed to trigger adaptive actions, what action each component takes, and the degree to which those actions are coordinated. A collection of alternative rule sets is typically associated with a given component. This allows the controller to be constructed in a modular fashion by selecting the appropriate policies for the given scenario based on, for example, the application domain or network architecture. Other rule sets may need to be developed based on the specific configuration and its need for coordination. Rule sets can also be replaced dynamically during program execution to alter policies implemented by the controller. The role of a given rule set depends on the controller execution phase to which it belongs, as described below.

3.4.1 Input Synthesis

Rule sets used during the first phase synthesize a variety of input variables that characterize system state qualitatively or provide only partial indications of the state of the system into fuzzy sets that more fully characterize the state of the system. There are, for example, a wide variety of ways to measure network congestion, such as using timeouts and duplicate acknowledgments [Pos81], using inter-acknowledgment spacing [SVSB99], or using changes in segment round-trip times [BOP94]. Each of these measures is useful in some scenarios, but not others. However, none of these methods are a *quantitative* measure of

congestion—they provide only a very coarse *qualitative* measure of the state of congestion in the network, such as whether congestion is *low*, *medium*, or *high*.

By synthesizing multiple measurements into a single fuzzy set representation, Cholla can achieve a more accurate view of the actual level of congestion in the system. The specific rule sets chosen determine the actual input synthesis policy used, and these rule sets can be customized based on the characteristics of the specific network being used. Figure 3.6 shows two different rule sets for detecting congestion in a transport protocol. The first set of rules uses TCP-like semantics for detecting congestion based upon information on acknowledgments, drops, and timeouts. The second rule set, similar to existing heuristics based upon changes in round trip times, might be more appropriate in other situations such as in wireless networks, or when transmitting video. Chapter 5 provides a complete example of input synthesis in a network transport protocol.

3.4.2 Adaptation Selection

Rule sets used during the second phase establish tentative values for controller output variables based on input variables and the state variables determined in the previous phase. Each such rule set is associated with a single component and implements a possible adaptation policy for the variable or variables in that component. For example, a rule set associated with a transport protocol may change the window size based on the level of congestion as captured by the input and state variables. Note that all the variables used at this stage are fuzzy sets.

Figure 3.7 shows simplified versions of the kinds of rules that might be used to control a flow control throttle, an error correction mechanism, and a video encoder. In this simplified version, a flow control throttle is controlled based upon the amount of congestion detected in the input synthesis stage. In contrast, the error correction is set using the average amount of loss found in the network, something that is measured directly by the system. At this stage, the video rate is controlled using information on the amount of data currently queued

```
if Congestion is high then FlowControlThrottle is HalfDecrease;
if Congestion is medium then FlowControlThrottle is Unchanged;
if Congestion is low then FlowControlThrottle is AdditiveIncrease;

ErrorCorrectAmount ← NetworkLossiness;

if TransmitQueue is high then VideoRate ← 0.8× VideoRateIn;
if TransmitQueue is medium then VideoRate ← VideoRateIn;
if TransmitQueue is low then VideoRate ← 1.2× VideoRateIn;

VideoRate ← VideoRateIn + PacketSize × DeltaAckRate
```

FIGURE 3.7. Example Adaptation Selection Rules

to be transmitted in the transport protocol and the change in rate at which the transport protocol is receiving acknowledgments. Chapter 5 provides a more complete example of rules for controlling congestion in a transport protocol, while chapter 6 provides details on controlling two application, a multimedia transmission application and a proxy for wireless networks.

Some coordination between components can be implemented in this phase by having a rule set use input or state variables associated with another component to determine a tentative output value. This structure exposes the relevant portions of the state of one component to another component via the shared controller. This might be used, for example, when an application uses variables related to network congestion to alter its behavior. This approach can also be used when components are designed to work together *a priori*, such as when they form constituent parts of a larger integrated service built using a system such as Cactus or Ensemble. In this case, the adaptation policy implemented by the rule set is, in essence, a policy of the entire service rather than of an individual component, with the variables viewed as shared state. The last rule in figure 3.7, which sets the video rate based upon state exported from the transport protocol, is an example of this. We term such coordination *implicit coordination*, as opposed to the explicit coordination done in the following phase.

```
if FlowControlThrottle is HalfDecrease
        then VideoRate ← 0.75× VideoRate;
if (ErrorCorrectAmount > OldErrorCorrectAmount)
        then VideoRate ← VideoRate ×
                (1 - (OldErrorCorrectAmount - ErrorCorrectAmount));
```

FIGURE 3.8. Example Adaptation Coordination Rules

3.4.3 Adaptation Coordination

Rule sets used during the third phase mediate among the tentative outputs to coordinate adaptation among multiple components. Two types of rule sets are common. The first takes tentative values for two different output variables that would produce undesirable effects when used together, and assigns new values to those variables. For example, a rule set combining frame rate for a video application and the congestion window size for a transport protocol may adjust those values to avoid over-adaptation in response to network congestion. The second type takes two different tentative values for the same output variable and generates a single new value. While needed for generality, rules of this type are often unnecessary since the defuzzification process naturally generates a single crisp value given multiple values represented as fuzzy sets.

Figure 3.8 shows a simple set of rules that could be used to coordinate video sending rate with error correction and flow control in a transport protocol. Both the flow control throttle and error correction mechanisms in the transport protocol, whose parameters may have been changed by the previous phase, can serve as bottlenecks to video transmission. Rules in the adaptation coordination phase can be used to take into account these potential bottlenecks. The rules in figure 3.8, for example, take decreases in the flow control throttle and increases in the amount of error correction into account for setting the rate at which video is generated.

The rule sets used in this phase are, as can be seen, dependent on the specific configuration. However, the number and size of these rule sets tends to be small, which simplifies construction. The ability to group these sets into a single phase also serves to isolate the configuration-dependent parts of the controller, which makes them easy to identify and change relative to ad-hoc solutions. Later chapters provide more complete examples of this structuring in the context of coordinating adaptations in video encoders and network protocols.

3.5 The Cholla Runtime

Supporting and interfacing adaptation controllers and adaptable components is the role of the Cholla runtime system. This runtime system, which uses techniques based largely upon well-known mechanisms, includes an event-based notification service, barrier synchronization primitives, and inter-host coordination protocols. These elements interact to allow coordinated policy decisions to be made and carried out effectively. In this section, we describe the various elements of the Cholla runtime system and how they interact with each other and the remainder of the Cholla architecture to realize a solution to the challenge of adaptation control and coordination.

3.5.1 Elements of the Runtime System

Cholla's event notification service is similar to those in many configurable networking frameworks and is used to manage control flow in the Cholla framework. Elements of the Cholla architecture, including controllers, adaptable components, and other portions of the runtime system, use the event notification service by *binding* handlers to well-known events, and *raising* notification of event occurrence. When an event is raised in the Cholla runtime, each handler bound to that event is invoked in turn by the Cholla runtime and given arguments that describe which event was raised. These arguments allow the element that bound the handler to deal with the event in an appropriate manner. Events are used

by adaptable components in Cholla to cause controllers to run and by controllers to signal components both before and after controller execution. Section 3.5.2 provides more details on how the Cholla event mechanism interacts with the remainder of the system.

Cholla's barrier synchronization can be used by adaptable components to synchronize the execution of various elements in the system. Barrier synchronization works by having *members* of the barrier *enter* the barrier when they wish to synchronize their actions with other members. In the case of Cholla, barrier members are typically adaptable components or different portions of adaptable components, such as the flow control and error correction portions of the transport protocol shown in figure 3.2. Once every member of a barrier has entered the barrier, each member is signaled using the Cholla event notifier. At this point, the barrier resets, allowing members to reenter and resynchronize if necessary.

Finally, inter-host coordination protocols such as those that have been described in the literature are included in the Cholla runtime system [RBH+98, CHS01]. These protocols gather information from remote machines to allow controllers to make decisions in distributed environments. In particular, the inter-host coordination protocols export *remote system state* to adaptation controllers, which they use along with the state exported by local adaptable components to make adaptation decisions. In addition, inter-host coordination protocols provide support for synchronizing adaptation among distributed components in a manner similar in spirit to how the barrier synchronization mechanism described above works.

While outside the scope of this dissertation, these coordination protocols do have some influence on controller design. In particular, such algorithms frequently require that control decisions be deterministic—if each machine can agree on the logic and inputs used to make adaptation decisions, then all machines will reach the same decision. The rule-based controllers around which the Cholla architecture centers have this property, allowing them to be used to implement algorithmic adaptation when combined with appropriate protocol support.

3.5.2 Interactions with Controllers and Components

The elements of the runtime described above are responsible for combining the various elements of the Cholla architecture into a functioning whole. As the runtime's event mechanism is the primary means of control flow in Cholla, it provides much of the glue tying together the various elements of Cholla. Several predefined events exist in Cholla that are used by components, controllers, and the runtime. These include the *RunController*, *ControllerFiring*, and *ControllerFinished* events associated with every controller, as well as a *SynchronizationCompleted* event associated with every adaptation barrier.

When an adaptable component or the inter-host coordination protocol receives an update that requires a new policy decision to be made, that component raises the *RunController* event. In the video transmission example in figure 3.2, for example, this could be because a new acknowledgment has been received by the transport protocol. After receiving the *RunController* event, the controller raises a *ControllerFiring* event. This event allows adaptable components to update the system state that they export to the controller prior to controller execution. In the video transmission example, the transport protocol would update its exported measurement of the lossiness of the network, while the video encoder might update its measurement of the average rate at which it is generating data.

After these events fire, the controller reads the state exported by the various components and executes the rule sets that define controller policy. The controller then updates the control parameters exported to it and fires the *ControllerFinished* event. Adaptable components managed by the controller receive this event as notification that their control parameters have been updated. In the case of the video example, these parameters are the new video rate, transport protocol transmission rate, and error correction level. If no other synchronization or coordination is necessary, the components can implement the adaptation policies mandated by the controller immediately.

Because controllers can be triggered by multiple components, it is frequently not possible to predict the rate at which they will run, although a component can mandate a mini-

mum rate. This can lead to problems if the control actions of running the controller twice in succession are cumulative—control actions could compound at an unpredictable rate. In general, controllers and their associated components need to be designed using the basic principle that they should perform no *worse* if the controller runs more often than expected, subject of course to the constraint that there are enough CPU cycles available to execute the controller.

This goal can be achieved in two ways: controller logic can be designed to be *idempotent* or components can mask the effects of faster-than-normal controller execution themselves. Idempotent control logic means that the controller sets outputs in a way that does not directly effect the inputs the next time the controller runs—the effects of the controller logic are not cumulative. This is the preferred way to deal with the issue of unknown controller execution rate, but is not always feasible. In cases where this is not possible, a component can mask the effects of unknown controller execution rate by keeping private copies of the parameters controlling that component. The component can then update its internal copies of the control parameters at a known rate, avoiding the problems that occur when the controller runs more often than anticipated.

There are also cases where additional coordination and synchronization is needed prior to carrying out the adaptations mandated by the controller. In these cases, adaptable components use the runtime's barrier synchronization primitive to assure that coordinated adaptations occur simultaneously. In the video transmission example, the video encoder, flow control throttle, and error correction mechanisms are all members of a shared *adaptation barrier*. Those components that are able to adapt their behavior immediately after the controller runs, such as the video encoder and the flow control portion of the transport protocol, enter the barrier on receiving the *ControllerFinished* event. However, the transport protocol's error correction mechanism may not be able to adapt immediately. A block error correction mechanism, such as one that might be used by the transport protocol in figure 3.2, can only adapt after a block of N packets has been completely sent, where N is typically between 5 and 25 packets. After the last packet in a block has been sent, the

error correction mechanism joins the adaptation barrier and every member of the barrier is notified using the *SynchronizationCompleted* event that they may proceed through the barrier.

Without such barrier synchronization, the coordinated policy decisions made by the adaptation controller may go to waste. For example, the controller may instruct the video sender to start generating data more quickly based on the assumption that the error correction rate is being reduced. If this reduction is in fact not going to take effect until the end of a block of packets, a temporary mismatch between adaptation in two components could occur. By using barrier synchronization, the error correction mechanism and video encoding component synchronize their adaptations, preventing any inconsistency from occurring. In systems that need distributed synchronization, the inter-host protocols can also join adaptation barriers at appropriate times to provide synchronization between local and remote components. After the completion of the *ControllerFinished* event and any additional barrier synchronization, each adaptable component implements the adaptation described by the controller in the appropriate control parameter.

3.6 Summary

Together, Cholla's controller architecture and runtime system provide a powerful framework for building configurable systems with multiple adaptable components. Cholla's rule-based approach to building controllers allows them to be customized by changing the rule sets that are used to construct the controller. When combined with the event-based notification, barrier synchronization, and coordination protocols in the runtime system, the Cholla architecture allows adaptation in multiple components to be coordinated for maximum effectiveness.

Of the prior work described in chapter 2, ARC [vDLS00] and SWiFT [Cen97] are the most closely related to Cholla. ARC provides adaptation coordination between multiple components in network protocol stacks, but is limited to coordinating adaptation between

strictly layered components. Cholla, in contrast, does not require adaptable components to be strictly layered, and addresses the issue of coordination between adaptable components in highly configurable systems. SWiFT, like Cholla, provides support for composing adaptive behavior from a set of smaller components, in the case of SWiFT small linear feedback components. SWiFT does not, however, provide any support for coordinating adaptation between multiple adaptable components in a system.

CHAPTER 4

CHOLLA IMPLEMENTATION

4.1 Introduction

The architecture described in chapter 3 provides a framework for constructing adaptable components in highly-configurable software systems. This chapter presents a prototype implementation that is designed to test the adaptation composition and inter-component coordination aspects of the architecture. Specifically, this prototype implements those elements of the Cholla architecture necessary for performing inter-component coordination in the context of a framework for building highly-configurable network protocols.

Figure 4.1 illustrates the general system model around which the prototype is designed. In this model, different network protocols are layered on top of each other and a variety of protocol stack configurations may be constructed from the network protocols available in the system. These protocols may themselves be internally structured as collections of smaller subcomponents. Instances of network protocols, called *sessions*, encapsulate the protocol-specific state associated with a particular network conversation. This state might be, for example, the congestion control information shared between communicating local and remote transport protocol instances. These sessions may contain subcomponents, as shown in figure 4.1. Sessions communicate with each other by *pushing* outgoing network messages down to lower-level sessions and by receiving information *popped* upward from lower-level sessions. Protocols are shown using dotted lines in figure 4.1, while sessions are shown using solid lines.

The adaptable components Cholla controls in this model are protocol sessions and their subcomponents. Each Cholla controller is responsible for controlling a group of (usually related) sessions. As shown in figure 4.1, for example, a single controller can be used to

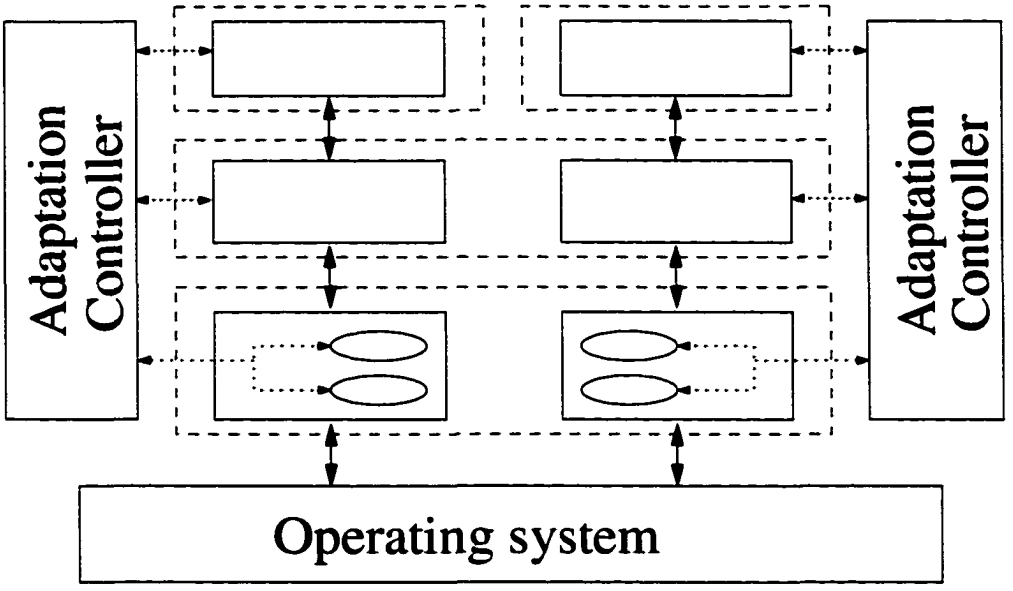


FIGURE 4.1. Architecture of the Cholla Prototype

control all of the network sessions associated with a single network connection, as well as the subcomponents inside these sessions. A single controller can also be used to control two different stacks of network protocol sessions that cooperate to perform a single function. For example, a Cholla controller can coordinate adaptation in two network stacks being used to send a multimedia presentation, where one stack transmits video and the other transmits the associated audio data.

The Cholla prototype described in this chapter implements this model in Cactus, a framework for constructing and composing highly-configurable network protocols. The prototype uses composable adaptation controllers to control network protocols written in Cactus, and uses the existing Cactus event mechanism to fulfill Cholla's event notification requirements. The prototype also uses Cactus events as the basis for an event-structured barrier synchronization primitive. The prototype does not, however, include inter-host coordination protocols, since these were not necessary to test the composition or inter-component coordination aspects of the system. Such inter-host coordination protocols have been implemented elsewhere [CHS01, RBH+98].

The remainder of this chapter describes this prototype implementation. Section 4.2 describes the Cactus framework in which this prototype is implemented. Section 4.3 then describes how adaptable components built in Cactus are controlled using Cholla, while section 4.4 describes the implementation of the Cholla runtime in Cactus. Section 4.5 describes the implementation of adaptation controllers, including the format and composition of rule sets, the fuzzy control engine that execute rule sets, and the interface used by adaptable components to communicate with the controller. Section 4.6 illustrates the interactions between different portions of the prototype by describing the implementation of a wrapper for a video transmission protocol. Finally, section 4.7 summarizes the features and limitations of the prototype.

4.2 The Cactus Framework

Cactus is a system for constructing highly-configurable protocols for networked and distributed systems. A novel feature of Cactus is its support for a two-level model for composing modular network software. First, individual protocols in Cactus are constructed from fine-grained software modules that interact using an event-driven execution paradigm. Each of these fine-grained modules implements a different property or function of the protocol. Next, protocols are layered on top of each other to create a protocol stack. This two-level approach has a high degree of flexibility, yet provides enough structure and control that it is easy to build collections of modules realizing a large number of diverse properties. Cactus also includes a dynamic message abstraction optimized for highly-configurable systems [HWS01].

Each protocol in Cactus contains a set of *protocol functions* that are used by sessions of the protocol to process incoming and outgoing data. These functions are invoked in a specific protocol session whenever new data is available for that session to process. For example, the `Push()` function is invoked in a session whenever a higher-level session function wants to send data to that session. Likewise, the `Pop()` function is called in a

session whenever a lower-level session has received data intended for that session. Other protocol functions exist for opening new sessions, closing existing sessions, and demultiplexing an incoming message to the appropriate protocol session.

Protocols in Cactus are built by implementing protocol functions such as `Push()` and `Pop()`. There are two main classes of Cactus protocols: *monolithic protocols* and *composite protocols*. Monolithic protocols implement specific protocol functionality by overriding the default implementation of protocol functions with code that implements the semantics particular to that protocol. Sessions of monolithic protocols contain the persistent state necessary to process network data using the protocol's customized protocol functions.

Composite protocols implement more generic flexible protocol functionality than monolithic protocols by decomposing protocol processing into smaller modules called *microprotocols*. Each microprotocol is responsible for implementing a particular property or functional component of the composite protocol. The specific functionality implemented by a composite protocol session depends on the microprotocols configured into that session. By configuring a composite protocol session in this way, the semantics of the session can be customized based on the needs of the user, application, and hardware. The persistent state in a composite protocol session consists of both global state shared by all microprotocols in that session, and state local to each of the *microprotocol instances*. This structuring enables multiple microprotocols in the same session to directly share data structures contained in the session state.

Unlike monolithic protocols, composite protocols and microprotocols do not normally customize the implementation of protocol functions by overriding their default implementation. Instead, microprotocols implement *handlers* that execute in response to *events* raised by the default protocol function implementations. For example, the `Pop()` protocol function in Cactus raises the MESSAGEFROMNETWORK event inside the session for which the message is intended. Each microprotocol that needs to perform an action upon receipt of a message from the network registers a handler for the MESSAGEFROMNETWORK operation.

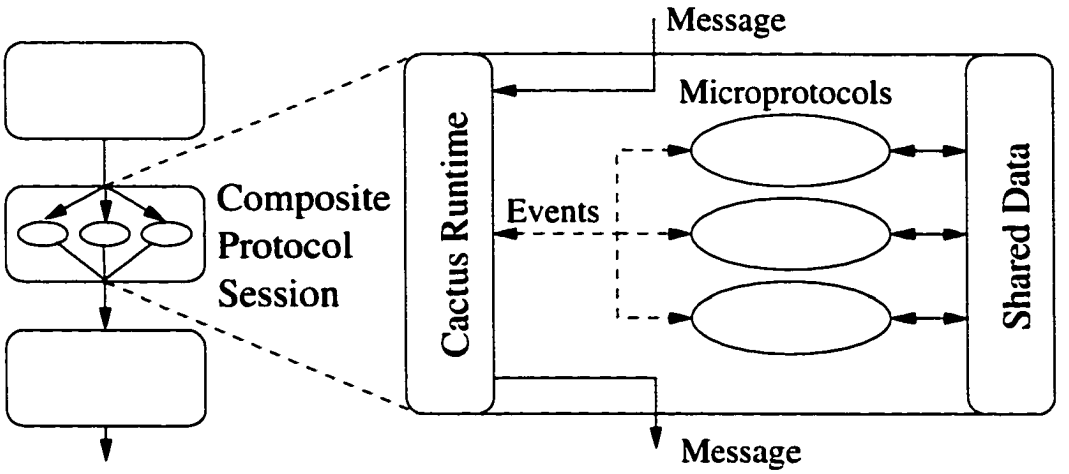


FIGURE 4.2. Composite Protocol Sessions in Cactus

Other predefined events can also be handled by microprotocols to customize the actions performed in response to invocations of `Push()`, `Open()`, and other protocol functions.

Cactus provides a number of operations for controlling event and handler execution inside a session. Events are created using `CreateEvent()`. There are two main operations on events: `BindHandler()`, which specifies a handler function to be executed when a given event occurs, and `RaiseEvent()`, which causes the handler functions associated with a specified event to be executed. Any function with access to the event object may call `RaiseEvent()`, even functions not in the session that created the event. Event handlers are passed the event that was raised, the session in which that event was created, and any additional arguments that were specified in the bind or raise operations. Other operations are available for deleting events, halting execution of the current event, and unbinding a handler from an event.

Figure 4.2 illustrates how sessions, microprotocol instances, and events interact in Cactus. Composite protocol sessions provide network service to those sessions higher in the protocol stack. Whenever a message is received by a composite protocol session from a higher layer, the default implementation of the `Push()` protocol function raises a MESSAGEFROMUSER event containing the message received. This event is processed by event

handlers defined in various microprotocols and the associated message is eventually sent to a session lower in the protocol stack, and finally over the network to a corresponding protocol stack on a remote machine. Similarly, whenever a message is received by a composite protocol session from a lower protocol layer (be it a monolithic protocol or a composite one), the default implementation of the `Pop()` protocol function raises a MESSAGEFROMNETWORK event in the receiving composite protocol. This event is handled in turn by the appropriate microprotocols, which may potentially raise other events and eventually pass the processed message on to sessions higher in the protocol stack.

Cactus includes a *dynamic message abstraction* with two unique features that make it useful in highly-configurable protocols: *attribute-based data storage* and *hold bits*. Data items in a dynamic message are stored in a dynamic set of named attributes with scopes corresponding to the composite protocol (*local*), the protocols on a single machine (*stack*), and the peer protocols at the sender and receiver (*peer*). The value associated with an attribute is set by calling `SetMsgAttr(message, scope, attribute, data)`, and retrieved by calling `GetMsgAttr(message, scope, attribute)`. This attribute-based data storage allows microprotocols to add their own custom information to outgoing messages as a new attribute without affecting the accessibility of data stored by other microprotocols.

Hold bits are a mechanism that consists of sets of boolean flags associated with messages. These sets are used to allow multiple microprotocols to agree when a particular action can be performed on a dynamic message. Microprotocols that need to hold a message `request()` a hold bit for that message, and then `set()` that hold bit when they agree that the action associated with their hold bit can be performed. Dynamic messages by default include two kinds of hold bits: *send bits* that are used to determine when participating microprotocols all agree that a message can be sent, and *deallocate bits* to determine when a message can be deleted. After every send bit associated with a message is set, the `SendMsg()` protocol function is called on that message. Similarly, after every deallocate bit associated with a message is set, the `DeleteMsg()` protocol function is called on that

state and control parameters to the controller. Section 4.5 describes how components cause the controller to run and how the controller interacts with components using the Cactus event mechanism. Cactus has no mechanism for exporting data from a composite protocol to external elements, however. To address this problem, we have added an interface for exporting data from Cactus sessions. This interface can also be used to provide a more structured form of data sharing between microprotocol instances than the current Cactus implementation.

The interface is centered around named *variables*—essentially an association between a name and a data address. Microprotocol instances that measure system state or export control parameters call `BindVariable()` to associate a variable name with a data location. System elements that need to access this variable, including other protocols, microprotocols, and Cholla adaptation controllers call `LookupVariable(session, name)` to gain access to the address associated with a variable in a particular session. The names associated with variable data locations are dot-separated names of the form `ComponentName.VariableName`. For example, the name of the video send rate in the video encoder component of a multimedia coder/decoder protocol might be `VideoEncoder.VideoRate`.

Adaptation wrappers for composite protocol sessions are implemented as wrapper microprotocols in the Cholla prototype. A wrapper microprotocol is a new, separate microprotocol responsible for interfacing a session to Cholla even though the session was not originally designed to work with Cholla. Figure 4.3 shows a session interfaced with Cholla using a wrapper microprotocol. The wrapper microprotocol connects the composite protocol session to Cholla using the appropriate API routines and then manages all interaction by the session with Cholla. This includes exporting the session's shared data and control parameters to the controller, measuring any additional system state that the controller might need, and handling events raised by the Cholla runtime and controller.

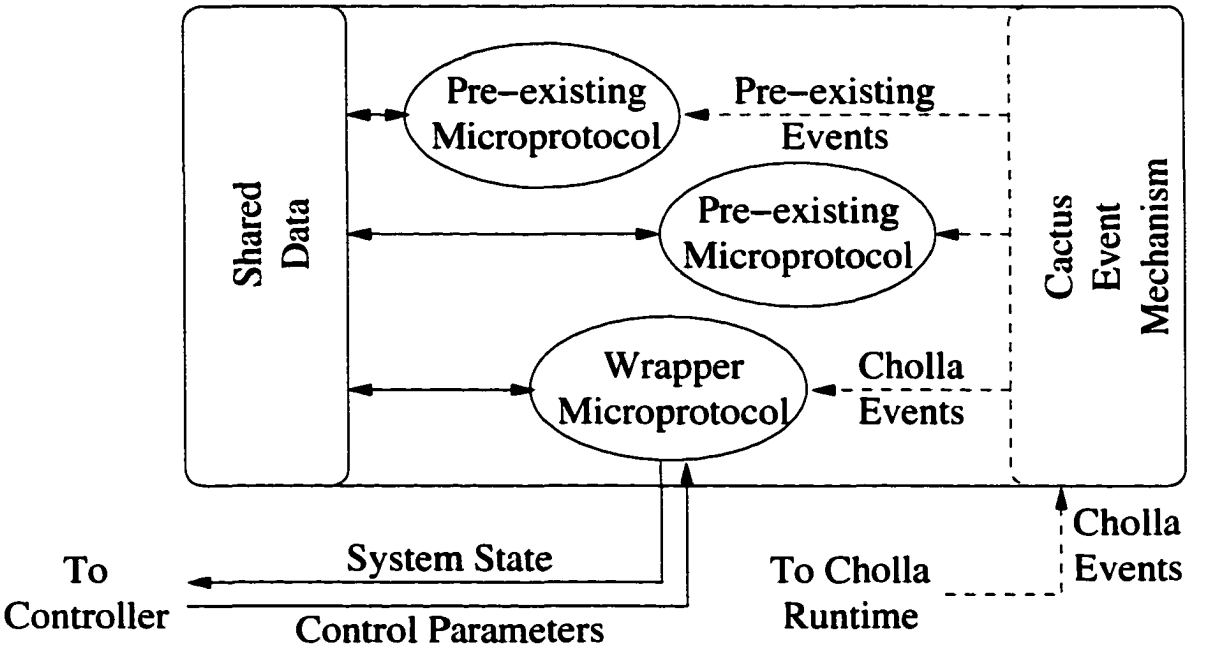


FIGURE 4.3. Composite Protocol Session containing a Wrapper Microprotocol

4.4 Cholla Runtime Implementation

As described in chapter 3, the Cholla runtime includes three main components: an event mechanism, a barrier synchronization mechanism, and inter-host coordination protocols. This prototype only focuses on those components necessary for evaluating Cholla's approach to inter-component coordination, and as such does not include inter-host coordination. In addition, the prototype's runtime reuses significant parts of the Cactus runtime, particularly the Cactus event mechanism, for implementing Cholla's event and barrier synchronization mechanisms.

Events in the Cholla architecture are used in precisely two situations: notifying components that the controller is preparing to run or has finished running, and notifying barrier members of barrier completion. These events are system-wide—any component in the system can bind a handler to a Cholla event and be notified when that event is raised. Events in Cactus, in contrast, are local to individual sessions.

Unlike many barrier synchronization implementations, the Cholla prototype described in this chapter implements an *event-structured barrier*. In most barrier implementations, members of the barrier block until every member joins. The barrier in this Cholla prototype, however, exploits the existing event mechanism to implement barriers without requiring components to block. When a member joins the barrier, the call to join immediately returns, regardless of whether or not every other member has already joined the barrier. Each member of the barrier is instead responsible for waiting to perform the action guarded by the barrier until signaled using the event mechanism.

Because `BarrierJoin()` does not block, components can continue operating while waiting for barrier completion so long as their operation does not affect the action being synchronized by the barrier. In addition, this allows barrier members to back out of a barrier they have joined before the barrier is complete. If multiple participants in the barrier can repeatedly join and back out, however, the barrier may never complete. Because of this, the `BarrierBackout()` routine must be used with caution.

As noted above, inter-host coordination protocols are not implemented in this prototype, but the elements in the existing prototype were designed to be easy to interface with such protocols. Specifically, inter-host coordination protocols would interact with the controller in a manner similar to adaptable components, namely by exporting state representing the status of other hosts to the controller and by responding to events raised by the controller. Similarly, inter-host coordination protocols could join and leave barriers to synchronize components on the local host with components distributed on other hosts.

4.5 Adaptation Controller Implementation

The primary issues in implementing adaptation controllers in Cholla are the interface between the controller and adaptable components, the language for specifying rules and rulesets, and the engine for executing rulesets. This section describes the interactions between the controller and adaptable components, along with the environment in which rules and

```
typedef struct _controller controller;
enum controllerEventType {CNTLR_FIRING, CNTLR_FINISHED};

int CreateController( controller *, char *name );
int DestroyController( controller * );

int RegisterSessionController( session *, char *name,
                               controller * );

int AddControllerEvent( controller *, event *,
                        enum controllerEventType);
int DeleteControllerEvent( controller *, event *,
                           enum controllerEventType );

int SetClockRate( controller *, double clockRate );
int RunController( controller * );
```

FIGURE 4.5. Adaptation Controller API

rulesets are written and composed. This section also describes the limitations of this implementation of the Cholla controller architecture.

4.5.1 Controller Interface

The prototype's adaptation controller supports an interface for adaptable components that allows them to tie into the controller through the event notifier and variable interface. This interface includes routines for creating and configuring new controllers, for associating protocol sessions with a controller, and for gaining access to the events associated with the controller. Figure 4.5 shows the programming interface that adaptable components use to create and interface with adaptation controllers.

Controllers in the prototype are represented by pointers to the `controller` structure, which contains state associated with the controller. Instances of this type are initialized using the `CreateController()` routine, which allocates internal controller data structures. Conversely, the `DestroyController()` routine can be used to free internal

controller data structures prior to deallocation of a particular controller. Protocol sessions obtain pointers to the controller structure either by allocating and initializing controllers themselves, or by being passed a controller by another protocol session when a new session is created.

Once a session has access to a controller, it requests that the controller manage it using the `RegisterSessionController()` operation. The session passes this operation the generic name of the session, such as "`Codec`" or "`CTP`". The controller then generates a unique name for the session by appending a number to this name, yielding, for example, "`Codec0`". Once a session has registered with the controller, the controller uses the variable interface to lookup and monitor the variables associated with that session's system state and control parameters. It performs this lookup based on the name of the session provided to the controller during session registration and the names of the variables in the rules that are configured into the controller. Section 4.6 provides an example of controller initialization and variable binding.

As described in section 4.4, events in the Cholla prototype are implemented indirectly using Cactus events. Cactus microprotocols use the `AddControllerEvent()` to associate a Cactus event with the Cholla CONTROLLERFIRING and CONTROLLERFINISHED events. The Cactus events associated with these Cholla events are then raised by the controller before and after the controller fires, respectively. Similarly, instead of exposing to adaptable components the internal event that triggers the controller, the controller interface exports a routine that directly cause the controller to run (`RunController()`) and a routine that requests a minimum rate at which the controller should run (`SetClockRate()`). If multiple components set the controller rate, it runs at the fastest of the requested rates.

4.5.2 Implementation of Control Logic

The prototype described here uses the JFS fuzzy control system to implement fuzzy control [Mor00]. Fuzzy controller specifications in JFS are written using a custom programming

language for the definition of fuzzy sets and the expression of both fuzzy and non-fuzzy control rules. These rules are compiled into a bytecode format that can then either be interpreted or translated into C code by JFS tools or libraries. JFS also includes facilities for analyzing control rules and for tuning controllers based on a variety of machine learning techniques, although these facilities are not used in the current Cholla implementation.

Rule sets in the prototype are implemented as fragments of JFS specifications residing in separate files. In the current implementation, a Python [PYT] script is given a list of the rule sets to include in the controller. This script combines the specified rule sets together into a single JFS controller specification. It also generates C code to initialize the controller at system startup and interface code that allows the generated JFS controller to lookup and set system state and control parameters using the Cholla API. The JFS specification is then translated into C and the resulting control function and interface code are compiled and linked into the program.

4.5.3 Prototype Limitations

One limitation of the JFS system is that it does not include support for fuzzy mathematical operations, such as assignment of one fuzzy set to another, direct addition of two fuzzy numerical values into a third fuzzy value, and similar operations. Instead, JFS performs operations on values represented as fuzzy sets by converting the value to a crisp numeric representation and then performing the operation on the crisp representation of the number. This complicates the implementation of rule sets that contain mathematical operations instead of fuzzy if-then inference statements. In particular, since Cholla uses fuzzy composition for combining different rule sets, the lack of support for fuzzy mathematical operations makes composing rule sets that contain mathematical operations and operate on pure mathematical values somewhat awkward. The current prototype implements composition of rule sets that are not based on if-then rules by using JFS's limited support for mathemat-

ical operations on values represented as fuzzy sets along with macros and additional type information in fuzzy sets.

A more serious limitation in the current prototype is that a controller's control logic is static. Controller specifications are constructed at compile time and compiled into static functions that are linked into applications. Because of this, the rule sets used in a particular controller cannot easily be changed while the system is running. This means that large-scale changes to the control strategy in response to radical system reconfiguration cannot be easily implemented in the current prototype. In addition, rule sets refer to specific sessions such as *"Codec0"* at compile time, even though the binding of Cactus session to session name is determined only at runtime. This means that controller must know the number and the order in which sessions are created at compile time, so that it can correctly associate rule sets with sessions.

Neither of these are inherent limitations, however. JFS supports interpretation of byte-code controller specifications and constructing and changing these specifications while programs are running. These facilities could potentially be extended to allow controllers to be composed and rule sets to be associated with sessions at runtime. This would allow the control logic itself to be adapted at runtime if necessary, for example, in response to large-scale system reconfiguration. It would also eliminate the current difficulties stemming from associating rule sets with sessions at compile time when sessions are only created at runtime. The only shortcoming of this approach is that interpreting controller logic at runtime could potentially be slower and have higher overhead than the static compilation method that our prototype currently employs. Dynamic code generation techniques could be used to avoid even this shortcoming, although this would require extensive changes to JFS or the implementation of a new fuzzy system.

4.6 Interactions Between Sessions and Cholla

The interface between Cactus composite protocols and Cholla is straightforward, since Cholla-specific code can usually be isolated into a wrapper microprotocol that specializes in interfacing a composite protocol and its other microprotocols with the framework. In this section, we provide an example of the code in a wrapper microprotocol used to connect a composite protocol session to Cholla and, in doing so, describe the interactions between this session and Cholla.

Figure 4.6 shows the source code for a simple microprotocol responsible for connecting a Cactus video encoder protocol with Cholla. In this example, the video encoder sets its rate using the value stored in `pcs->videoRate`. This variable is controlled by a Cholla adaptation controller that is interfaced to the video encoder using this wrapper microprotocol.

The wrapper microprotocol starts by exporting the current video rate so that the adaptation controller can use it as an input, and exporting `pcs->videoRateAdapt` to store the output of the controller. The wrapper then registers the current session with the controller using the controller reference passed to the initialization routine as an argument. The protocol session containing this wrapper could obtain the controller reference in a variety of ways, although it would frequently be passed to the session as an argument when the session was opened. Upon session registration, the controller assigns the name "`CODEC0`" to the session; this is the first CODEC session registered with the controller. The controller then searches its rules for references to the `CODEC0.VideoEncoder.VideoRateIn` and `CODEC0.VideoEncoder.VideoRateOut` variables, and associates the variables just exported by the wrapper microprotocol with these predefined rules.

After registering with the controller, the wrapper creates a Cactus event to use for signalling that the controller has executed. The wrapper binds a handler to this event, and requests that the controller raise this event after controller execution finishes. In addition, the wrapper enrolls inself as a member of the adaptation barrier used to synchronize with

```
void wrapperInit(session *ps, controller *pc,
                 componentstate *pcs)
{
    /* Export appropriate variables from the component */
    BindVariable(ps, "VideoEncoder.VideoRateIn",
                 &pcs->videoRateIn);
    BindVariable(ps, "VideoEncoder.VideoRateOut",
                 &pcs->videoRateOut);

    /* Register this session with the controller */
    RegisterSessionController(ps, "CODEC", pc);
    pcs->pevControllerFinished = CreateEvent(ps);
    BindHandler(pevControllerFinished, controllerHandler);
    AddControllerEvent(pc, pevControllerFinished,
                       CNTLR_FINISHED);
    /* Become a member of the adaptation barrier and
     * add a handler to the associate event */
    pcs->nBarrierMember = BarrierNewMember(pcs->pb);
    pcs->pevBarrierComplete = CreateEvent(ps);
    BindHandler(pevBarrierComplete, barrierHandler);
    BarrierAddEvent(pcs->pb, pevBarrierCompleted);

    /* Run the controller at least twice per second */
    SetClockRate(pc, 2);
}

static void controllerHandler(csession *ps,
                              componentstate *pcs)
{
    /* We're always ready to adapt. Enter the barrier */
    BarrierJoin( pcs->pb, pcs->nBarrierMember);
}

static void barrierHandler(csession *ps, componentstate *pcs)
{
    /* Copy the controller's output to the variable that
       controls the rate, causing adaptation to happen */
    pcs->videoRate = pcs->videoRateOut;
}
```

FIGURE 4.6. Code for Connecting a Video Encoder to Cholla

other components, and registers an event with the barrier. Finally, the wrapper requests that the controller run at least twice per second.

After the controller runs, the wrapper's `controllerHandler` event handler is executed. This handler immediately joins the associated adaptation barrier since the video encoder can adapt its sending rate immediately. After all adaptable components have entered this barrier, the wrapper's `barrierHandler` function is called. This handler copies the controller's output from the `videoRateOut` variable into the `videoRate` variable, which results in a change in the rate at which the video encoder generates data.

4.7 Summary

The Cholla prototype described in this chapter implements many of the key elements of the coordination architecture described in earlier chapters. In particular, it implements those elements necessary for testing the adaptation composition and inter-component coordination aspects of Cholla. These features are implemented in the Cactus framework, allowing these features to be tested in the context of highly-configurable network protocols. The following chapters describe several protocols and applications implemented using this prototype, including an adaptable network transport protocol, a video transmission application, and a customizable proxy for wireless networks.

CHAPTER 5

ENHANCEMENT AND CONTROL OF A CONFIGURABLE TRANSPORT PROTOCOL

Transport protocols are used by a wide range of applications to carry data over networks. As described in chapter 1, the semantics needed by different applications for this purpose can vary widely and may need to change based on changing network conditions. Because of this, transport protocols contain many opportunities for composing adaptation policies and coordinating adaptation between multiple mechanisms inside a single protocol. In addition, when applications that use adaptable transport protocols are themselves adaptive, there are also opportunities for coordinating adaptation between transport protocols and applications. This chapter focuses on the control of a highly-configurable transport protocol called CTP using the Cholla prototype described in chapter 4. In addition, it describes a number of improvements to CTP that were orthogonal to controlling CTP with Cholla; these enhancements make CTP more suitable for use in adaptive applications.

This chapter is organized as follows. Section 5.1 provides a general overview of salient features of CTP, including the microprotocols and events it uses. Section 5.2 then describes our enhancements to CTP, and section 5.3 describes how the enhanced CTP implementation is connected to and controlled by Cholla. Finally, section 5.4 presents an experimental evaluation of the control of CTP using Cholla that demonstrates how Cholla can be used to provide fine-grained configuration to the control of a highly-configurable network protocol. Later chapters in this dissertation illustrate how Cholla can also be used to coordinate adaptation between CTP and specific applications.

5.1 CTP Overview

5.1.1 CTP Architecture

CTP is a highly-configurable, message-oriented composite transport protocol implemented using the Cactus framework [WHS01]. CTP sessions are given *messages*, arbitrarily-sized pieces of data, by sessions higher in the protocol stack. Each CTP session then converts these messages into *segments*, data units sized for transmission by network protocols and devices, that are then sent to a remote CTP session using a network protocol such as IP. The remote CTP session reconstructs application messages from the segments it receives and delivers these messages to the session above it in the protocol stack. The specific semantics of a CTP transport session such as connection-establishment, reliability, and flow control depend on the microprotocols configured into that session. Some semantics of CTP are fixed, however. Because CTP is a message-oriented protocol, for example, it always preserves message boundaries on the data it transports. [1]

Figure 5.1 shows an example CTP composite protocol session. CTP uses several events to control message and segment processing by a variety of microprotocols. Microprotocols respond to these events by manipulating message and segment attributes, changing shared data, and raising new events. These microprotocols and events are discussed in the following subsections.

5.1.2 CTP Microprotocols

The original CTP implementation described by Wong, *et al.* includes a number of different microprotocols, each of which implements a variant of some aspect of the protocol's functionality. Some of these aspects of transport protocol functionality are listed below, along with examples of CTP microprotocols that implement different variants of that functionality.

[1] In contrast, stream-oriented protocols such as TCP do not preserve message boundaries. This may allow them to deliver partial data more quickly at the cost of losing message framing information.

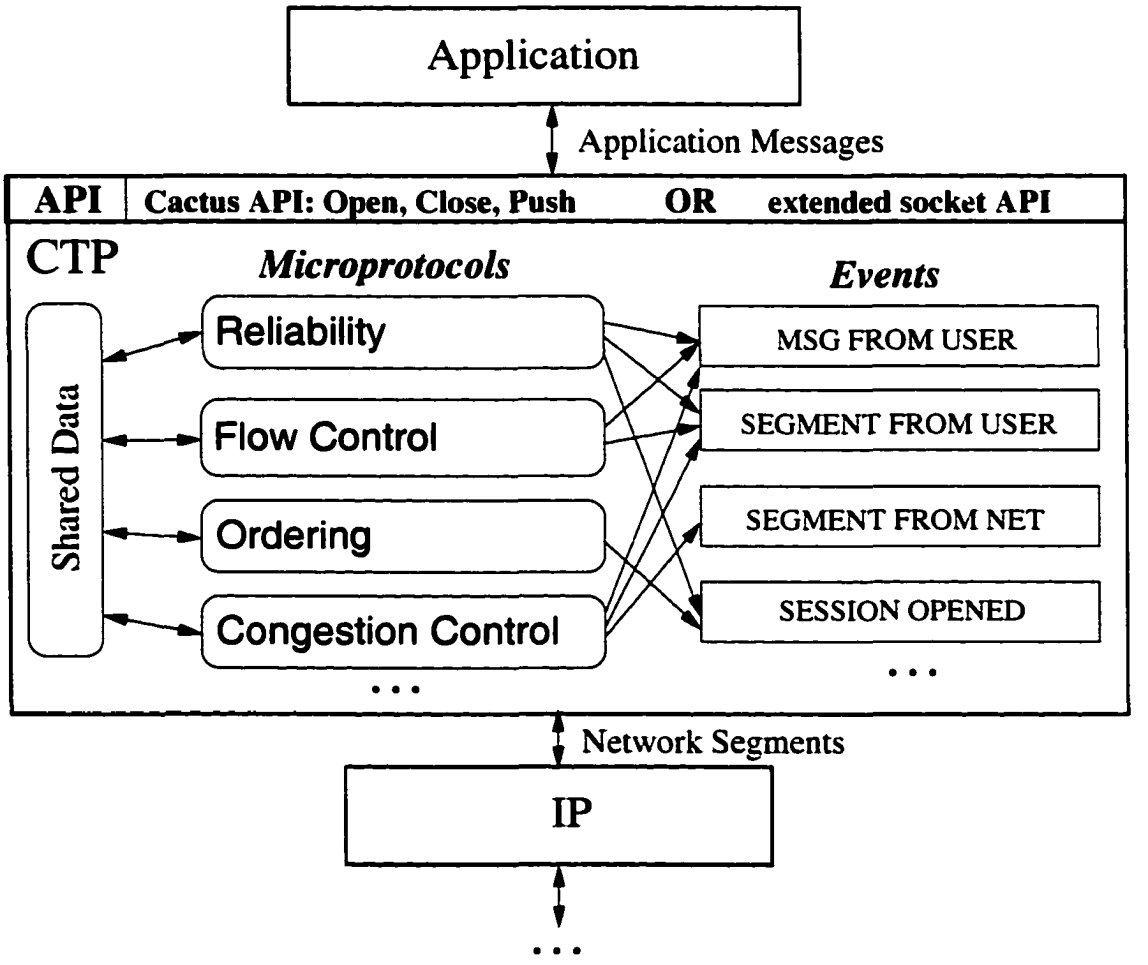


FIGURE 5.1. CTP Protocol Architecture

Message/Segment Conversion : FixedSize, Fragmentation

Flow and Congestion Control : WindowedFlowControl, WindowedCongestionControl

Reliability : ForwardErrorCorrection, PositiveAck

Message Ordering : ReliableFIFO, LossyFIFO

Miscellaneous Services : RoundTripTimeEstimation, SequencedSegments

Every CTP session must include a message/segment conversion microprotocol. Instances of conversion microprotocols package outgoing application messages for transmission over the network by converting them into segments. They are also responsible for converting incoming segments back into messages suitable for consumption by applications and higher-level protocols. Flow control, ordering, and reliability microprotocols such as those listed above can also be included in a CTP session if these features are needed. Finally, some microprotocols exist solely to provide commonly needed services to other microprotocols. For example, the PositiveAck microprotocol uses the SequencedSegments microprotocol to uniquely identify segments by sequence number and the RoundTripTimeEstimation microprotocol to keep track of end-to-end segment round-trip times.

5.1.3 CTP Events

Microprotocols respond to a variety of events, primarily ones associated with message handling. There are six main events in CTP:

MESSAGEFROMUSER : raised by the `Push()` protocol function when a new message is received from the application.

SEGMENTFROMUSER : raised by a message/segment conversion microprotocol after application messages have been packed into network segments.

SEGMENTFROMNET : raised by the `Pop()` protocol function when a new segment is received from the network.

MESSAGEFROMNET : raised by a message/segment conversion microprotocol after network segments have been converted into application messages.

SENDMSG : raised using send bits when a segment is ready to be sent over the network.

DEALLOCMSG : raised using deallocate bits when a message or segment is ready to be deleted.

Four of these events constitute the transmission and reception paths through CTP, with one event per path for messages and one for segments. The remaining events are used to manage message and segment state and are raised using hold bits.

In addition, some microprotocols use *private events*, events allocated and managed by a single microprotocol. These events are used for a variety of reasons, such as for a timer mechanism. Microprotocols that need to perform periodic actions, for example, may arrange for a private event to be raised at a fixed rate, while protocols that need to keep track of segment timeouts may create and raise a private event that fires when a segment times out.

5.2 Enhancements to CTP

While CTP provides a wide range of transport features, the original design and implementation of CTP has several limitations. Some CTP microprotocols, such as the PositiveAck microprotocol, implement several different facilities that are not closely related. In addition, the original CTP design did not separate congestion and flow control mechanisms from the policies that controlled these mechanisms. This section describes enhancements to CTP that address these shortcomings, as well as several other improvements that make

CTP more suitable for use in adaptable systems. Several of these enhancements use primitives developed for use in Cholla that have since been added to Cactus, such as the barrier synchronization primitive described in chapter 4.

5.2.1 Acknowledgments and Retransmission

One significant shortcoming of the initial CTP implementation is its use of a single PositiveAck microprotocol for performing two logically separate functions: tracking the status (*received/lost/timed out*) of transmitted segments, and reliable transmission of segments using timeouts and retransmissions. This overloading causes problems for applications that want segment status tracking but not retransmissions. Retransmissions are often undesirable because they can increase average end-to-end message latency and jitter. The increase in jitter is particularly troublesome in soft and hard real-time scenarios, such as streaming multimedia applications.

We have enhanced CTP with a new acknowledgment architecture that generalizes the status tracking provided by PositiveAck into a protocol-wide facility that can be implemented using a variety of different microprotocols. These enhancements allow CTP to be configured to use acknowledgments for feedback about segment arrival and loss without mandating the introduction of retransmissions and their negative effects on multimedia applications. This work divides the original PositiveAck microprotocol into a number of different microprotocols, each of which implements a portion of the functionality of the original PositiveAck microprotocol. The new PositiveAck microprotocol implements acknowledgments and segment timeouts, but nothing else. The DuplicateAck and NegativeAck microprotocols are now responsible for notification of packet loss, while the Retransmit microprotocol implements segment retransmission. This separation required changes to both the sender and receiver sides of CTP, as described below.

Sender-side Changes. The sender-side changes to CTP are centered around a new method of notifying microprotocols about changes to segment status. This method uses new Cac-

handles SEGMENTLOST and SEGMENTTIMEOUT events so that it can retransmit segments when necessary.

Receiver-side Changes. The predominant change to the receiver side of CTP is when acknowledgments are sent by PositiveAck. Traditional protocols use cumulative acknowledgments to inform the sender that the receiver actually received the acknowledged segment and *all prior segments*. PositiveAck, however, should be able to support feedback of packet status in both reliable and unreliable configurations, and the traditional meaning of cumulative acknowledgments is not appropriate in unreliable protocols. Such protocol configurations neither want nor need to guarantee the receipt of every segment and message, but still desire the feedback on packet status provided by acknowledgments.

To address this difficulty, the enhanced PositiveAck microprotocol generalizes cumulative acknowledgments so that they work well in CTP configurations that are not completely reliable. Specifically, an acknowledgment from the enhanced PositiveAck microprotocol signals that the receiver no longer needs or expects to receive the acknowledged segment or any segment sent prior to it, not that those segments were necessarily received. In reliable protocols, where the receiver expects to receive every packet, this behavior results in the standard acknowledgment behavior used in protocols such as TCP. In protocols that do not require complete reliability, however, the more general definition allows an acknowledgment for a packet to be sent even if some previous packets have not been received.

Several additional data structures are used to implement this acknowledgment behavior in the new version of PositiveAck. PositiveAck now manages a data structure on the receiver that tracks whether the *reliability constraints* of each listed segment have been met. In CTP configurations that do not include reliability microprotocols, the initial reliability constraint of each segment is RELIABILITY_MET. Microprotocols that implement reliability, however, can change the default reliability status of segments to RELIABILITY_UNMET. When PositiveAck receives a new segment, it sets the reliability constraint of that seg-

ment to RELIABILITY_MET. Other microprotocols can also set the status of segments to RELIABILITY_MET if they determine that a segment is no longer needed.

When PositiveAck receives a segment from the network, it acknowledges the highest numbered segment that has been received and for which the reliability constraints on all previous packets have been met. For example, consider the case where segment number 6 is received. PositiveAck sets the reliability status of segment 6 to RELIABILITY_MET and then checks the status of other received segments. If segments 1, 2, 3, 5, and 7 have all been received but segment 4 has not had its reliability constraints met, PositiveAck will send an acknowledgment for segment 3. If the reliability constraints on segment 4 have also been met, however, PositiveAck will acknowledge segment 7. When a microprotocol that implements complete reliability like Retransmit is included, this approach acknowledges the highest in-order segment that has been received. Without any reliability microprotocol, on the other hand, PositiveAck acknowledges the highest numbered segment that has been received, regardless of whether or not all previous segments have been received. This method also allows the use PositiveAck with microprotocols that desire *partial reliability*, i.e., the reliable delivery of only some segments. The current implementation of CTP does not contain such microprotocols, however.

5.2.2 Congestion Control

Another shortcoming of the original CTP design is that it did not separate congestion control mechanism and policy. Because there are a wide range of different congestion control policies that all use similar congestion control mechanisms [Jac88, Kes91, SCFJ96, MJV96, CPW98, RHE99, FHPW00, YL00], separation of mechanism and policy is important in a configurable transport protocol. We have designed and implemented CTP microprotocols that implement window-based and rate-based congestion control mechanisms, separated these mechanisms from the microprotocols that implement congestion control policy, and implemented several different common policies.

Congestion Control Mechanisms. CTP now contains two different CTP microprotocols that implement congestion control mechanisms, WindowedCongestionControl and RateBasedCongestionControl. The overall goal of both microprotocols is the same: to implement a mechanism that can be used by *policy microprotocols* to control how quickly CTP sends data based on indications of congestion received by these policy microprotocols. Both WindowedCongestionControl and RateBasedCongestionControl limit how quickly segments are sent using send bits, although the mechanisms they use for deciding when to set send bits differ. Both mechanisms are governed by parameters that can be modified to control how quickly segments are sent. These parameters are stored in CTP's shared data, allowing policy microprotocols to control segment transmission speed. It is also worth noting that our implementations of WindowedCongestionControl and RateBasedCongestionControl are orthogonal—both microprotocols can be used in the same session and a segment will only be transmitted by such a session when both microprotocols approve its transmission.

WindowedCongestionControl limits segment transmission speed using the sliding window approach taken by many other transport protocols, including TCP. This approach attempts to limit the number of segments that can be in the network at one time. In particular, it limits the number of transmitted segments that have not been acknowledged or timed out. It does this using three variables, *windowSize*, *lastSegmentSent* and *lastSegmentDone*. The range of segments between *lastSegmentSent* and *lastSegmentDone* are those segments currently outstanding in the network and comprise the *congestion window*. WindowedCongestionControl limits the size of the congestion window to no more than *windowSize*, the congestion window size, by preventing new segments from being sent. As segments are retired by acknowledgments or timeouts, *lastSegmentDone* increases, which in turn allows WindowedCongestionControl to send new segments, increasing *lastSegmentSent*. The segment transmission rate achieved using the sliding window algorithm described above is at most *windowSize/segmentRoundTripTime*. By changing the congestion window size used

by WindowedCongestionControl, policy microprotocols can control the number of outstanding segments in the network and in doing so limit segment transmission rate.

WindowedCongestionControl must be able to tell when a segment is no longer in the network and will not be in the network again so that it can advance the trailing edge of the congestion window, *lastSegmentDone*. This means that WindowedCongestionControl must be used with PositiveAck in the current CTP implementation, since PositiveAck is currently the only microprotocol that keeps track of segment acknowledgment and timeout. WindowedCongestionControl must also take into account PositiveAck's interactions with other microprotocols such as Retransmit, since Retransmit could reintroduce old segments into the network. If WindowedCongestionControl does not take these factors into account. it could incorrectly advance the congestion window prior to a segment retransmission, causing too many unacknowledged segments to be in the network at once.

WindowedCongestionControl handles these interactions by introducing a new SEGMENTDONE event on the sender-side of CTP. SEGMENTDONE is raised whenever a segment and all segments with lower sequence numbers no longer need to be retransmitted by any CTP microprotocol. This event is managed using the event-structured barrier abstraction described in section 4.4 and later added to Cactus. Microprotocols such as Retransmit that could cause a segment to be retransmitted register with this barrier, and join it when they no longer need to transmit a segment. A numeric argument is associated with the SEGMENTDONE barrier that is used to track which segment has completed. WindowedCongestionControl registers with the barrier and enters it whenever the SEGMENTACKED, SEGMENTTIMEOUT, or SEGMENTLOST events are raised. It also sets the associated numeric argument to the sequence number of the segment just acknowledged or timed out when it does so. Similarly, Retransmit registers with the barrier if included in a CTP session and enters the barrier whenever a segment is acknowledged.

As an example of how the barrier works in practice, consider what happens when the arrival of new acknowledgment is signaled by the SEGMENTACKED event. In such a case, WindowedCongestionControl updates the numeric argument associated with the barrier in

response to the event and then enters the barrier. Retransmit also enters the barrier in response to this event, causing the SEGMENTDONE event to be raised and the congestion window to advance. If a timeout or loss occurs instead, WindowedCongestionControl still enters the barrier and updates its associated numeric argument. Retransmit, if included in the CTP session, does not enter the barrier, however. This prevents the barrier from completing and the window from advancing because a segment in the window could still be reintroduced into the network. If Retransmit is not present in the CTP session when a timeout or loss occurs, on the other hand, the barrier would complete when WindowedCongestionControl entered it. This would allows the window to advance as is appropriate.

Unlike WindowedCongestionControl, the RateBasedCongestionControl microprotocol does not need to keep track of the status of individual segments. Rate-based congestion control schemes limit the rate at which segments are sent, and do not depend on the state of transmitted segments or on feedback from the receiver to clock the transmission of new segments. This means that rate-based congestion control schemes generally send segments more smoothly than window-based schemes, and may therefore be more appropriate for multimedia applications. This also makes rate-based congestion control schemes more appropriate for situations in which little or no feedback from the receiver is possible, such as in unidirectional applications like satellite data transmission. One minor limitation of the current implementation of RateBasedCongestionControl is that it does not limit retransmissions of segments that have already passed through the RateBasedCongestionControl microprotocol. This could occasionally allow segments to be sent at faster than the configured rate. Since these packets have previously been lost, however, the average rate at which packets are *received* by the peer should remain at or below the configured rate.

Congestion Control Policies. We have implemented several congestion control policies for the WindowedCongestionControl microprotocol. Each congestion control policy is implemented as a policy microprotocol that uses segment status events and information from microprotocols such as RoundTripTimeEstimation to set the congestion window size used

by WindowedCongestionControl. The TCPCongestionControl microprotocol, for example, implements the TCP additive-increase/multiplicative-decrease congestion control algorithm [Jac88]. The SCPCongestionControl microprotocol, in contrast, implements the SCP congestion control policy described by Cen, *et al.* [CPW98]. Unlike TCPCongestionControl, SCPCongestionControl was designed for use in multimedia protocols. Of course, other policy microprotocols could also be implemented, including rate-based congestion control policies.

While each of these microprotocols implements a complete congestion-control policy, each policy is still relatively coarse-grained. Fine-grained customization of these policies is difficult and requires implementation of an entirely new policy microprotocol. Section 5.3 describes an alternate fine-grained approach to implementing congestion control policies in CTP that uses the Cholla architecture described in chapter 3.

5.2.3 Other Enhancements

We have also implemented a number of other enhancements to CTP that increase its utility to adaptable applications. For example, in the original version of CTP, every CTP session in an application had to use the same set of microprotocols because microprotocol selection was performed at compile-time. We have supplemented CTP with a simple facility for specifying which microprotocols should be included in each CTP session when that session is created at runtime. This allows applications that send multiple types of data with differing transport requirements to use CTP sessions with differing semantics when necessary. In addition, we have added a microprotocol for coalescing several small messages into a single outgoing segment to save on transmission overhead, custom handling of Cactus dynamic messages to decrease transport protocol overhead, facilities for measuring the rate of segment loss in the network, and a simple adaptive forward error correction policy microprotocol.

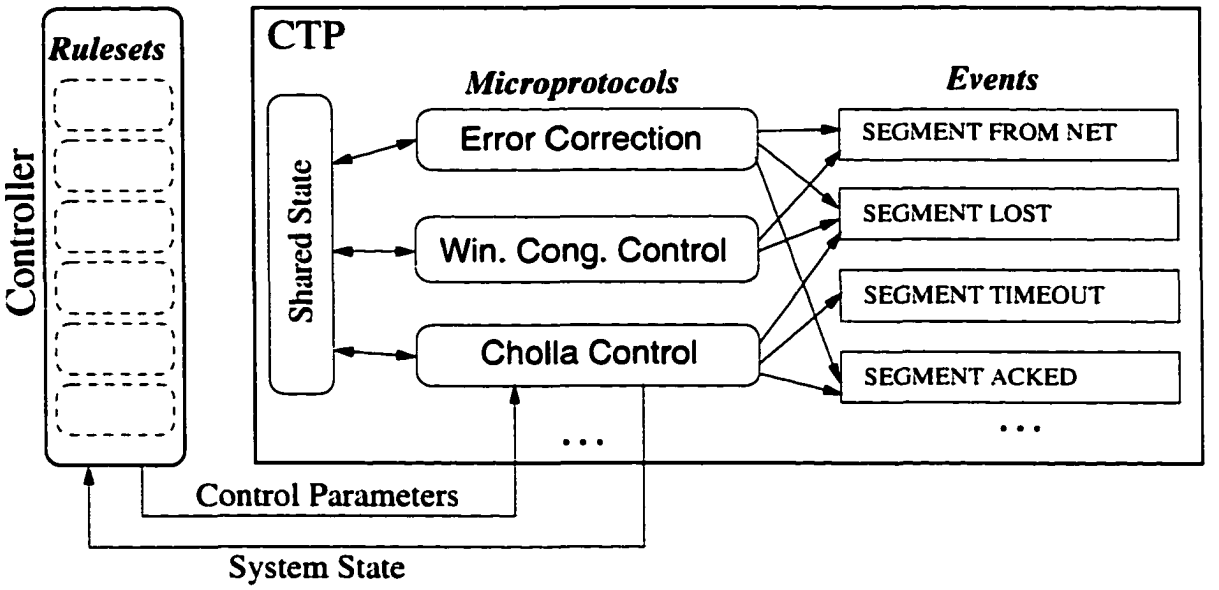


FIGURE 5.3. Controlling CTP using Cholla

5.3 Connecting CTP to Cholla

5.3.1 Overview

In addition to the changes described in section 5.2, we have modified CTP to use the Cholla prototype described in chapter 4 to control adaptation in CTP. Using Cholla to control adaptation in CTP allows its adaptation to be coordinated with adaptation in other protocols. It also allows adaptation policies that are currently implemented as standalone policy microprotocols in CTP to be decomposed into several Cholla rule sets instead. This decomposition allows adaptation policies to be customized easily and new policies to be prototyped more quickly. Figure 5.3 shows the relationship between the adaptable CTP microprotocols, the CTP wrapper microprotocol ChollaControl, the Cholla controller, and rule sets that comprise the controller. This section describes the adaptable CTP components controlled by Cholla, the inputs and outputs used to effect control over these components, the wrapper microprotocol used to connect CTP to Cholla, and rule sets that implement CTP adaptation policies.

5.3.2 Adaptable Components

Cholla is responsible for controlling two different adaptable components in CTP: the WindowedCongestionControl microprotocol and the ForwardErrorCorrection microprotocol. The adaptable mechanism used by WindowedCongestionControl is the sliding window algorithm described in Section 5.2.2. Cholla controls the size of the congestion window in this microprotocol in response to different indications of congestion in the network. The control strategies used by Cholla and rule sets that implement these strategies are described in section 5.3.5.

The ForwardErrorCorrection microprotocol implements a block-erasure forward error correction algorithm [Riz97b]. When ForwardErrorCorrection is used, N total segments are sent for every K data segments that CTP would normally transmit, where N and K are integers and $N \geq K$. The additional $N - K$ segments added by ForwardErrorCorrection contain redundant data that can be used to reconstruct the original K data segments as long as any K of the N transmitted data and redundant segments sent are received. This allows sessions that use the ForwardErrorCorrection microprotocol to tolerate up to $N - K$ segment losses from each block of N segments without losing any data segments. Details on the effects of changing N and K are given in section 5.3.4.

5.3.3 Cholla/CTP Interface

Adaptable components in CTP interface with Cholla using the ChollaControl wrapper microprotocol. This microprotocol contains most of the Cholla-specific portions of CTP, although minor changes are also required to the adaptable components to allow them to fully coordinate their actions using Cholla. Isolating as much of the Cholla-specific portions of CTP as possible in a wrapper microprotocol allows CTP to be used with Cholla when appropriate without requiring that Cholla be used in every CTP session.

CTP interacts with Cholla by performing five different tasks:

- Creating or acquiring a controller to control CTP;

- Recording any additional inputs required by Cholla rule sets that were not previously measured by CTP;
- Exporting control parameters and measurements of system state from CTP session state using the variable interface described in section 4.3;
- Running the controller when necessary using `RunController()` or `SetClockRate()`;
- Synchronizing adaptation in WindowedCongestionControl and ForwardErrorCorrection using an adaptation barrier.

The ChollaControl microprotocol is responsible for the first four of these actions. When a CTP session is opened, ChollaControl accepts a controller created by the higher-level session opening CTP as an argument, and uses this controller to manage adaptation in CTP. If a controller is not supplied when the session is opened, ChollaControl creates its own controller using `CreateController()`. This controller cannot then be used to coordinate adaptation in CTP with adaptation in other sessions, however, since other sessions do not have access to it.

ChollaControl also exports any CTP system state and control parameters needed by the controller. Since system state and control parameters are stored in CTP's shared session state, ChollaControl uses the `CreateVariable()` function to export this state to the controller. ChollaControl also maintains any internal measurements of system state that are not computed by CTP. For example, no CTP microprotocol currently stores the number of acknowledgments that have been received since the controller last ran, so ChollaControl measures this value for use by the controller.

Finally, ChollaControl is responsible for running the controller at the appropriate time. ChollaControl supports two different disciplines for executing the controller: *fixed-rate control* and *unclocked control*. When using fixed-rate control, ChollaControl simply calls `SetClockRate()` to set the rate at which the controller executes. While this reduces

the computational overhead that the controller imposes on the system, it can also limit how quickly the system responds to changes in the environment.

In unclocked control mode, in contrast, ChollaControl fires the controller explicitly using `RunController()` in response to receipt of the SEGMENTACKED, SEGMENTTIMEOUT, and SEGMENTLOST events. This allows the system to respond more quickly to changes in the system environment, since there is no delay between event handling and controller execution. Unclocked mode may, however, place more computational load on the system since the controller could run very frequently when many events are being received. The results of experiments comparing unclocked and clocked control are given in section 5.4.3.

The remaining element in interfacing CTP with Cholla is using an adaptation barrier to coordinate adaptation in the ForwardErrorCorrection microprotocol with adaptation in other protocols. This barrier resides in CTP's shared state and is used by the ForwardErrorCorrection microprotocol to avoid any potential coordination problems caused by the delay between when the controller finishes running and ForwardErrorCorrection is able to adapt, as mentioned in section 3.5.2. Such coordination requires changes toForwardErrorCorrection, since the ChollaControl wrapper microprotocol does not have access to the internals of this microprotocol and cannot always know when it is able to adapt. However, the implementation of these changes allows ForwardErrorCorrection to continue to work when CTP is used without Cholla and the ChollaControl microprotocol.

5.3.4 System State and Control Parameters

Controller Inputs. Cholla has access to a number of different inputs from CTP to use in making control decisions. Because CTP variable names can be long, variables are usually referred to using shorter abbreviated names in the Cholla rule sets. Figure 5.4 shows the names of the input variables used in Cholla, along with the name under which they are exported from the ChollaControl microprotocol.

Cholla controller sets this variable in response to indications of congestion. The rule sets that control this variable are given in section 5.3.5.

Similarly, the ForwardErrorCorrection control parameters N and K are exported to the controller as `ForwardErrorCorrection.N` and `ForwardErrorCorrection.K`. By controlling these variables, Cholla can control what percentage of the data transmitted by CTP is redundant, and therefore how much segment loss in each block the CTP connection can tolerate. For a particular value of N, decreasing K increases the protection against loss at the cost of transmitting more redundant data. As N increases, more data segments are protected in a given block of N packets, increasing the protection against the entire block of N packets being erased due to a burst of loss and allowing more flexibility in the choice of K (since K is an integer and $N \geq K > 0$). Like decreasing K, increasing N has associated costs, since larger sequences of segments require more CPU cycles to generate redundant data for than shorter sequences. Larger values of N also result in more data segments being sent before redundant data is sent, potentially increasing *reconstruction delay*, the time between when a lost segment would normally be received and when it is reconstructed using the last block of redundant data. Because of this, applications that use forward error correction must balance their bandwidth and CPU demands and constraints on reconstruction delay with the number of segments in each error correction block and the amount of redundant data in each such block.

5.3.5 Control Design

In designing the Cholla control strategies for WindowedCongestionControl and ForwardErrorCorrection, we had several different goals. In particular, we wanted control strategies that allowed for:

- construction of common transport control policies
- customization of transport policies for new system environments

- Ack-based Congestion Detection
- Drop-based Congestion Detection
- Timeout-based Congestion Detection

(a) Input synthesis rule sets

- TCP congestion control
- SCP congestion control
- TCP slow start
- Bernoulli FEC Adaptation

(b) Adaptation selection rule sets

FIGURE 5.5. CTP Rule Sets

coordination is done between protocol layers, which means that such rule sets are specific to the particular protocols included in the system.

Congestion Control Rule Sets. As described above, CTP's congestion control rule sets are based on existing congestion control policies, namely those used in the TCP and SCP transport protocols [Jac88, CPW98]. Because congestion is not easily measured, Cholla's congestion control policies make use of input synthesis rules to measure the amount of congestion in the network. The results of those rules are used by rules in the adaptation selection phase to determine how to adapt to network congestion.

The input synthesis congestion control rule sets are responsible for synthesizing multiple inputs to characterize network congestion. These inputs include the number of segments that have been acknowledged and lost, how many segments have timed out, the current round trip time, and a variety of other pieces of information. While each of these individually provides only a rough indication of congestion, together they can provide a relatively fine-grained qualitative measure of the actual congestion level in the network.

This qualitative measure is expressed as a fuzzy set with members ranging from *very low*, meaning the network is likely very underutilized, to *very high*, meaning that the network connection is in danger of completely collapsing from overutilization. In addition, the change in network congestion is estimated as a fuzzy set with members ranging from *negative large* to *positive large*. The use of fuzzy sets gives the controller a uniform quali-

% Timeout based Congestion Detection rule set:
if **numTimeout** is *true* then **deltaCongestion** is *positive large*
if **recentTimeout** is *true* then **congestion** is *very high*

% Drop Detection Based Congestion Detection rule set:
if **numDrop** is *true* then **deltaCongestion** is *positive medium*
if **recentDrop** is *true* then **congestion** is *high*

% Ack-based Congestion Detection rule set:
if **numAck** is *true* and (**congestionWindowIn** >= **safeWindow**)
    then **deltaCongestion** is *negative small*
if **recentAck** is *true* and (**congestionWindowIn** >= **safeWindow**)
    then **congestion** is *low*
if **numAck** is *true* and (**congestionWindowIn** < **safeWindow**)
    then **deltaCongestion** is *negative medium*
if **recentAck** is *true* and (**congestionWindowIn** < **safeWindow**)
    then **congestion** is *very low*

FIGURE 5.6. Congestion Control Input Synthesis Rule Sets

tative representation of the level of congestion in the network, something that is otherwise difficult to measure.

A number of alternative policies have been developed, each of which is appropriate in different situations. These rule sets are shown in figure 5.6. Note that all the variables in the rules are fuzzy—the crisp input variables have been fuzzified into fuzzy sets before these rules are executed. The rules are divided into these rule sets so that different combinations can be used to instantiate different policies. For example, when the underlying network is known to be lossy—for example, a wireless network—a message loss is not a reliable indicator of network congestion and thus, the second rule set should be omitted.

The adaptation selection phase uses the synthesized measure of congestion along with other inputs for deciding how to adapt to the measured amount of congestion. We have implemented several different policies for changing the size of the CTP congestion window in response to network congestion. In addition to TCP and SCP rules, simple rules are included that set the desired level of forward error correction based upon the level of

```
% Standard TCP Congestion Control rule set:
if deltaCongestion is positive large
    then congestionWindow ← 1
if deltaCongestion is positive medium
    then congestionWindow ← congestionWindowIn/2
if deltaCongestion is negative small
    then congestionWindow ←
        (congestionWindowIn + 1/congestionWindowIn)

% TCP Tahoe Slow Start rule set:
if deltaCongestion is negative medium
    then congestionWindow ← (congestionWindowIn+1)

% SCP-like Steady State rule set:
if congestion is not (very high or very low)
    then congestionWindow ← (ackRate × baseRTT) + 3.0
```

FIGURE 5.7. Congestion Control Adaptation Selection Rule Sets

Bernoulli loss in the network. These rule sets are shown in figure 5.7; in this figure, ← is used to denote fuzzy assignment, which expands into roughly five control rules. The variables lossiness and ackRate are input variables that indicate the observed loss rate of the underlying network and the rate at which acknowledgment are being received, respectively.

Similar to input synthesis, different policies can be created by configuring the adaptation controller with different combinations of rule sets. For example, SCP-style congestion control can be created by replacing TCP steady state rules with the SCP steady state rules. Note that when used together, some of these rule sets may propose different values for the size of the congestion window. This can be handled explicitly in the adaptation coordination phase, but this is usually not necessary because defuzzification automatically combines multiple output values into a single crisp output value as described in section 3.3. The experiments in section 5.4.2 demonstrate the composability of these rules.

Forward Error Correction Rule Sets. Figure 5.8 gives the adaptation selection rules used to control forward error correction in CTP. In contrast to the congestion control rule

% FEC adaptation rule set:
fecN ← **fecNin**
fecK ← (1 - **lossiness**) × **fecNin**

FIGURE 5.8. Forward Error Correction Rules

sets, the current forward error correction control rule sets are based on a simple model that attempts to correct for the average amount of loss in the network. Specifically, the rules shown in figure 5.8 set the amount of error correction used to average amount of loss in the network and do not attempt to control `ForwardErrorCorrection.N`.

This strategy has several limitations. Because most network losses are not evenly distributed and instead occur in bursts, these rules may underestimate the amount of error correction necessary to mask a series of losses. Correcting this would require measuring the average length of loss bursts, setting `ForwardErrorCorrection.N` to be significantly larger than this number, and setting `ForwardErrorCorrection.K` to be large enough to mask a burst of losses. Existing CTP microprotocols are not equipped to measure the average length of loss bursts, however. For this reason, we instead use the simple strategy shown in figure 5.8.

5.4 Experimentation

5.4.1 Setup

To determine the effectiveness of controlling CTP with Cholla, we evaluated CTP using a number of scenarios with different application requirements and underlying network characteristics. A scenario determines both the micro-protocol configuration in CTP and the rule sets used in Cholla. We considered the following three scenarios:

- **Bulk Data.** File transfer with TCP-style congestion control.
- **Wired Multimedia.** Video or audio transmission on a wired network with SCP-style congestion control policies used to supplement standard TCP congestion control.

- SequencedSegments
- PositiveAck
- NegativeAck
- RoundTripTimeEstimation
- WindowedCongestionControl
- ChollaControl

FIGURE 5.9. CTP Microprotocols used for Experimentation

(a) Bulk Data	(b) Wired Multimedia	(c) Wireless Multimedia
• Ack cong. detect	• Ack cong. detect	• Ack cong. detect
• Drop cong. detect	• Drop cong. detect	• Timeout cong. detect
• Timeout cong. detect	• Timeout cong. detect	• SCP cong. control
• TCP cong. control	• SCP cong. control	• TCP slow start
• TCP slow start	• TCP slow start	

FIGURE 5.10. CTP Rule Sets used in Test Scenarios

- **Wireless Multimedia.** Video or audio transmission on a wireless network with SCP-style congestion control policies used with the subset of TCP congestion control policies that behave well in networks with intermittent losses.

Experiments were run between two 200MHz Pentium Pro machines, each with 128MB of RAM, connected by 10 megabit per second twisted-pair Ethernet. Both machines were running RedHat Linux release 6.2. The system was built using the C version of Cactus 2.0, with the *x*-kernel used as the means of composing protocols. JFS was used as the fuzzy inference engine [Mor00].

CTP was configured to include the microprotocols listed in figure 5.9 while the rule sets used in each scenario are shown in figure 5.10. CTP was driven by a simple test protocol that buffers and transmits data, which was sufficient to test the different congestion control behaviors that are appropriate for the three scenarios listed above. Chapter 6 explores coordination and control of CTP in a multimedia transmission application and a proxying application.

5.4.2 Composition of Congestion Control Rules

We evaluated the composability and configurability of rule sets by examining the behavior of CTP's congestion control mechanisms when configured with policies that would be used in different adaptive scenarios. Each scenario was run in unclocked mode using the same set of components and mechanisms; only the rule sets governing adaptation policy were changed between scenarios.

Rule set behavior was tested by sending packets containing 1250 bytes of random data as fast as the congestion control mechanism would allow. This was done by pausing the CTP congestion control mechanism by artificially setting the size of the congestion window to 0, buffering sufficient packets in CTP so that it would not starve on connection startup, and then resetting the congestion control window to 1.0, causing CTP to start sending data. An additional microprotocol was also added to CTP that buffered an additional packet every time a packet was successfully transmitted, guaranteeing that the transmission queue never starved for data while placing minimal additional load on the memory system, CPU scheduler, or event mechanism.

Each scenario was run 10 times for 120 seconds per run. The average rate at which data was sent, the average size of the congestion window, and the standard deviation of the size of the congestion window were measured. A trace of the size of the congestion window over the course of each run was also collected. A lossy network was simulated for each scenario by dropping every 10th packet (10% of the data) received by each machine. While this is not an accurate model of the loss process in a real wired or wireless network, it is sufficient to study how the behavior of the congestion control mechanism changes under different policies.

Table 5.1 shows the results of the experiments. The TCP-like congestion control mechanisms used for **Bulk Data** had the largest average window size and standard deviation. This is expected since the TCP congestion control rules are fairly aggressive in looking for available bandwidth. Despite this, however, the **Bulk Data** rules only sent marginally

Scenario	Packet Send Rate (Mb/s)		Avg. Congestion Window Size (pkts)		Avg. Congestion Window Std. Dev.	
Added Loss	0%	10%	0%	10%	0%	10%
Bulk Data	6.9	6.5	69.6	4.0	15.4	0.8
Wired Multimedia	7.4	7.1	5.6	5.0	0.1	0.8
Wireless Multimedia	7.3	7.2	5.5	5.3	0.1	0.1

TABLE 5.1. Send Rate and Congestion Window Size for Various Configurations

slower than the multimedia-oriented rule sets; this is most likely due to the increased load that the large amount of queuing used by this method places on the network, especially the device queue and drivers on the receiver.

In contrast, the **Wired Multimedia** experiment with a rule set derived from SCP uses a much smaller average congestion window size. Despite this, this rule set was able to send marginally faster than the rules used in the **Bulk Data** scenarios. The variance in the window size is also much smaller, implying that this rule set results in a much more even data transmission rate. However, the **Wired Multimedia** control rules are much less aggressive in looking for available network bandwidth than the **Bulk Data** rules; in networks where the amount of available bandwidth changes rapidly such as the Internet, the **Bulk Data** rules may be more appropriate.

When periodic drops are introduced to simulate a lossy network, the results change. The standard **Wired Multimedia** sends more slowly because it backs down quickly in the face of congestion, falsely indicated by intermittent drops. Even worse, the standard deviation of the congestion window size increases, indicating increased variation in the rate at which data is sent. The **Wireless Multimedia** scenario, which does not include rule sets that treat losses as congestion indications, does not have any additional variation in sending rate.

In addition, traces of the congestion window size demonstrate the expected behavior; these traces are representative of congestion control behavior in the different scenarios. Note that figure 5.12 covers a much smaller timescale so that packet-by-packet variations in window size can be seen after losses are added. Figure 5.11 shows the behavior of the **Bulk Data** and **Wired Multimedia** scenarios on a network with little loss. The **Bulk Data**

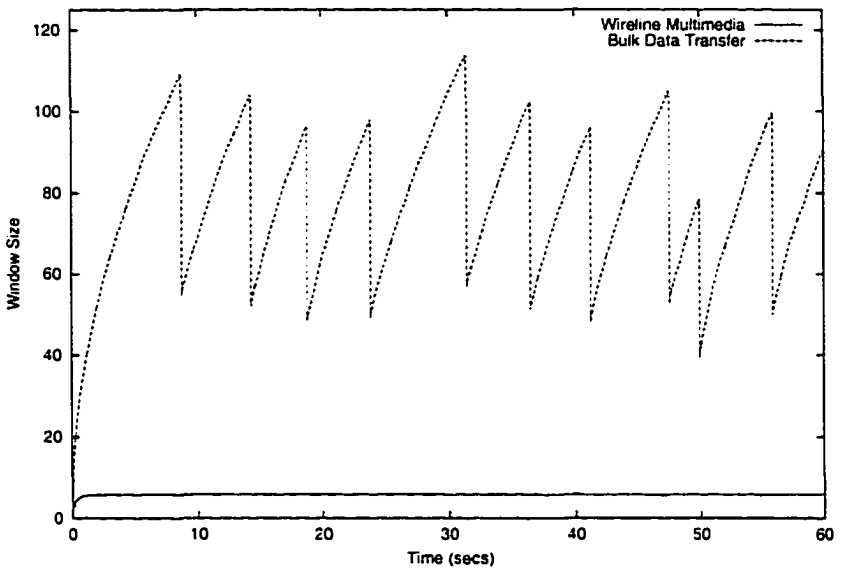


FIGURE 5.11. Congestion Control Behavior With Different Rule Sets

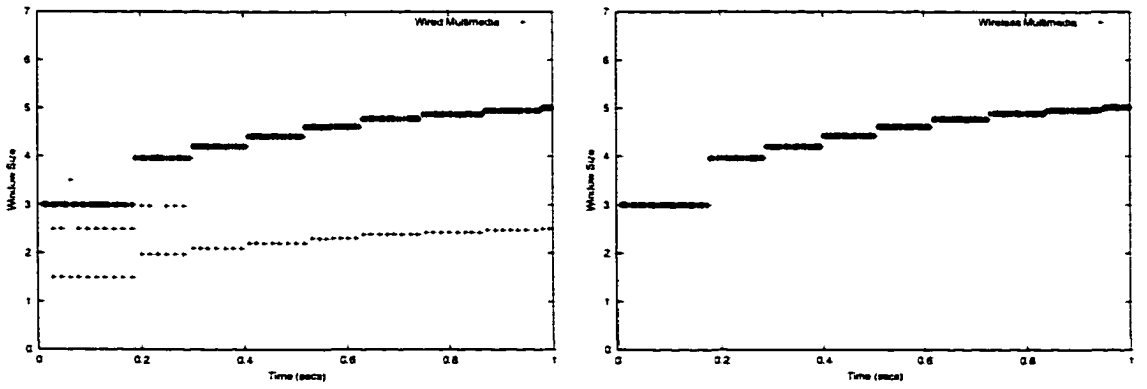


FIGURE 5.12. Multimedia Congestion Control Behavior Under Loss

scenario shows the classic TCP "sawtooth" behavior, while the **Wired Multimedia** scenario settles at an even sending rate.

Figure 5.12 shows the behavior on a network with extra losses. The **Wired Multimedia** configuration sends at a more erratic rate when extra loss is added; every time a negative acknowledgment is received, the congestion window is halved and then restored when a new acknowledgment is received. The **Wireless Multimedia** configuration, in contrast, sends at an even rate because it does not include the rule sets that use loss to indicate congestion.

5.4.3 Effect of Fixed-rate Control

The experiments in the previous section were run in unclocked control mode to prevent any interaction between controller clocking and rule set dynamics. It can be useful, however, to run the controller in clocked mode to reduce CPU usage by the controller. This is especially true on slow machines with frequent control events, because these events could cause the controller to run very often. Running the controller at fixed intervals, however, can also reduce the speed at which the controller is able to respond to changes in the environment. Ideally, the clocked controller should approximate the behavior of the unclocked controller.

To determine how closely different controller clock rates approximate the behavior of an unclocked controller, we measured the throughput, average window size, and variation in window size at various clock rates while using the **Bulk Data Transfer** rules. The test setup was identical to that used in the previous experiments, with one exception—we reduced the drop frequency to 1% in the tests that included extra loss. Without this change, a packet drop would occur between almost every controller invocation except at extremely high clock rates. Because packet loses in the **Bulk Data** scenario dominate controller behavior by reducing the window size, high drop rates would mask differences in the behavior of different clock rates. By setting the drop rate to 1%, packet drops occur at least 3 times per second (3 Hz) when the window is 1 segment and approximately 300

Controller Clock Rate	Packet Send Rate (Mb/s)		Avg. Congestion Window Size (pkts)		Avg. Congestion Window Std. Dev.	
Added Loss	0%	1%	0%	1%	0%	1%
1 Hz	3.6	3.0	41.3	1.0	57.3	0.0
5 Hz	6.9	3.4	65.9	1.4	15.1	2.2
10 Hz	6.9	6.3	67.0	15.5	14.5	13.1
25 Hz	6.9	7.1	66.7	11.7	14.2	3.4
50 Hz	6.9	7.1	66.7	12.2	14.4	2.8
Unclocked	6.9	7.1	69.6	12.7	15.4	2.3

TABLE 5.2. Bulk Data Send Rate and Congestion Window Size for Different Clock Rates

packets per second are exchanged, and no more than approximately 7-8 times per second (7-8 Hz) when the congestion window is larger, and on the order of 700 packets per second are being exchanged.[2]

Table 5.2 shows the throughput, average congestion window size, and standard deviation of congestion window size using various clock rates. At low clock rates, control behavior can vary significantly from behavior in the unclocked system because the dynamics of the system are faster than the rate at which the controller fires. In the case of no added packet loss, the controller differs dramatically only at 1 Hz, when the controller holds the window open for a significant amount of time even after a packet has been lost, causing additional packet losses and timeouts. As the control rate increases to 5 Hz and higher, however, the control rate is sufficient to approximate the behavior of the unclocked system relatively closely, even though the unclocked controller runs on the arrival of every acknowledgment, approximately 700 Hz.

When 1% packet loss is added, however, the 1, 5, and 10 Hz controller rates all differ significantly from the behavior of the unclocked controller. At extremely low rates such as 1 Hz and 5 Hz, the congestion window is either never or rarely opened before a drop causes the controller to reduce the window size. The 10 Hz clock rate was fast enough to occasionally raise the congestion window to a relatively large value and hold it at that

[2]With 1250-byte packets, 700-800 packets per second is approximately 7-8 megabits per second, roughly as much as throughput as can be expected sent on a 10 megabit per second link when using per-packet acknowledgments.

value until the next time that the controller ran. This happens at 10 Hz but not slower speeds because 10 Hz is the first controller rate marginally faster than the maximum packet loss rate of approximately 8 Hz. The standard deviation in window size at 10 Hz with 1% loss is remarkably high because the controller frequently sees one of two kinds of inputs:

- New acknowledgments and no packet loss, causing the controller to increase the window size and hold the window at that level even if a loss occurs immediately after the controller fires.
- A packet loss, causing the controller to set the window to a low level for an entire control interval (0.1 sec, in this case), even though new acks are received.

As controller clock rates increase, however, the interval during which the controller holds the window artificially high or low decreases. As a result, control rates of 25 and 50 Hz closely approximate the behavior of the unclocked controller, while running much less frequently than the 700 Hz at which the unclocked controller runs. Our testbed is fast enough that the difference in CPU utilizations at different controller clock rates was not measurable. On less powerful machines such as palmtops, PDAs, and embedded devices, however, the difference could be significant.

5.5 Summary

Controlling CTP using the Cholla prototype described in chapter 4 allows applications fine-grained control over how adaptation is performed in CTP. Similarly, our improvements to the acknowledgment, retransmission, and congestion control portions of CTP make CTP more configurable and flexible. This is useful for applications that need both reliable and unreliable data transmission sessions or for applications that need custom protocol semantics not implemented by existing transport protocols. Instead of using multiple transport protocols or implementing custom protocol handling, these applications can instead simply use appropriate CTP configurations. The next chapter describes two applications that

take advantage of the enhanced facilities for adaptation, coordination, and configurability in CTP.

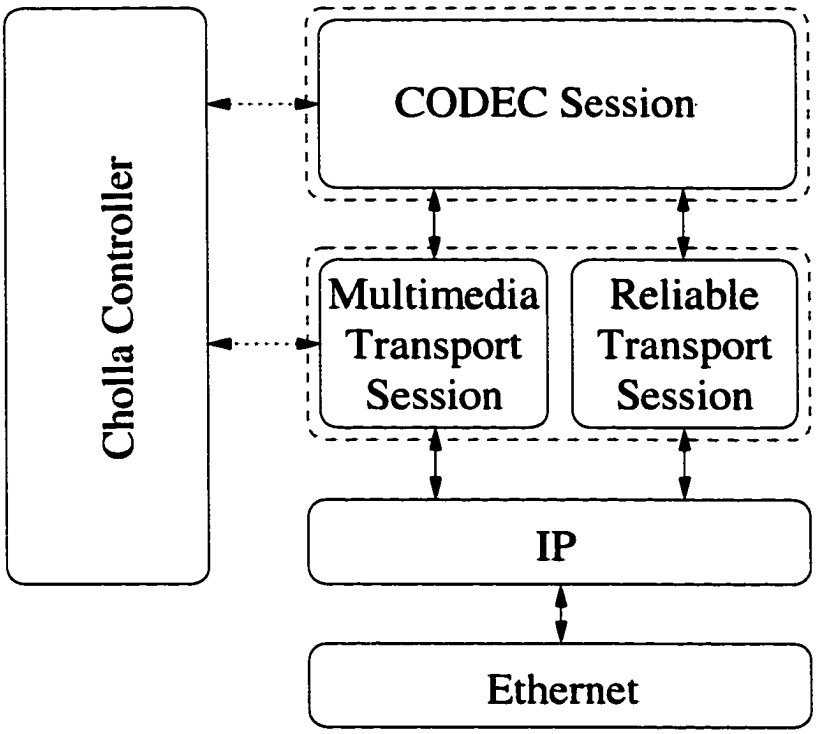


FIGURE 6.1. Multimedia Transmission Application Architecture

network, while the Cholla framework controls and coordinates adaptation in CODEC and CTP sessions.

CODEC sessions use two different types of transport protocol sessions: a multimedia-oriented transport session and a reliable transport session. The multimedia-oriented session is used to transport media data with soft real-time constraints to remote machines, while the reliable transport session is used to transfer non-real-time configuration information. When CTP is used as the transport protocol, as is the case for all of the configurations used in this chapter, the multimedia-oriented transport protocol is controlled by Cholla, and adaptation in the CODEC session is coordinated with adaptation in the multimedia transport session. Because the reliable data transport session is only used to transport non-real-time configuration data, it can be controlled either using Cholla or using microprotocols such as TCPCongestionAvoidance.

The following subsections describe in detail the various components of the multimedia application. Sections 6.1.2 and 6.1.3 describe the architecture of the CODEC composite protocol and the various CODEC microprotocols, and section 6.1.4 describes the various adaptable mechanisms in these microprotocols. The general approach used by Cholla to control adaptation in these mechanisms is described in section 6.1.5, while sections 6.1.6 and 6.1.7 describe the specific rule sets that implement this approach to control of adaptation in the audio-related and video-related mechanisms, respectively.

6.1.2 CODEC Protocol Architecture

Like CTP, the CODEC protocol is implemented as a Cactus composite protocol, where each microprotocol instance is responsible for implementing a portion of the behavior of that session. This allows the CODEC to be configured to work with a wide range of input devices, compression modules, data transport methods, and output destinations. Such configurability is common in multimedia-oriented applications, although it is usually implemented using dynamic link libraries (DLLs).

Each CODEC session includes microprotocols for performing five separate tasks, as follows.

- **Data acquisition microprotocols** acquire multimedia data from an external source, such as a camera, microphone, or data file, for transmission by the application.
- **Media encoder microprotocols** compress or otherwise encode data for transmission over a network.
- **Media streaming microprotocols** send and receive multimedia data using an appropriate transport protocol session, normally a Cholla-controlled CTP session.
- **Media decoder microprotocols** decode received multimedia data for display or archival.

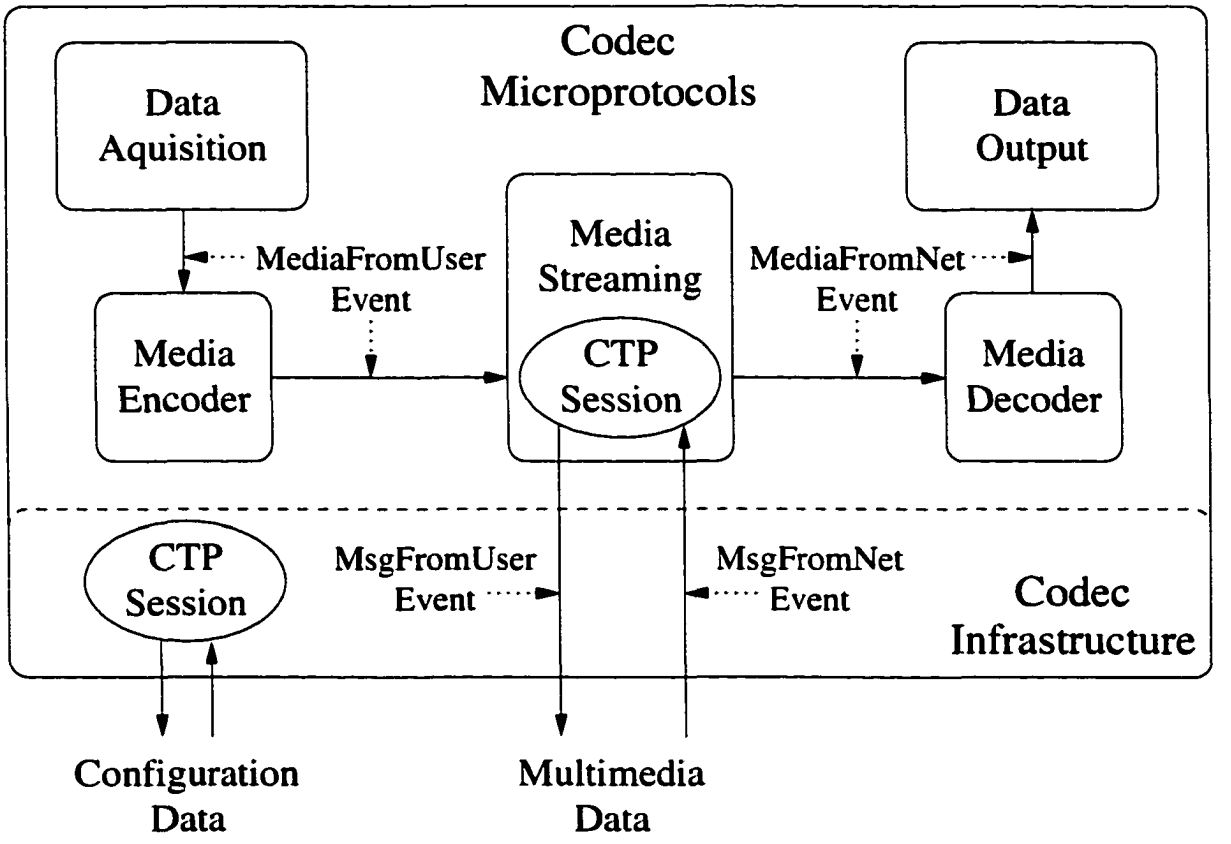


FIGURE 6.2. CODEC Protocol Architecture

- **Data output microprotocols** send received and decoded information to external destinations, such as display screens, speakers, or files.

While most CODEC microprotocols implement just one of these tasks, some may implement more than one. A microprotocol that can handle both compression and decompression of a particular media format, for example, performs both the media encoding and media decoding tasks listed above. Section 6.1.3 presents specific examples of the different types of CODEC microprotocols.

Figure 6.2 shows each type of microprotocol and how they interact using Cactus events. Data acquisition microprotocols are responsible for acquiring multimedia data, such as video frames or audio samples, from an external source and creating messages that contain this data using the Cactus dynamic message abstraction. These microprotocols also

add attributes to each message that describe the format of the data in the message. The **Video4LinuxAcquisition** microprotocol, for example, uses Linux's video API to acquire images from a video camera and formats those images into dynamic messages. The microprotocol also adds `video_format`, `video_resolution`, and `video_sampletime` attributes to the message, which describe the sample's video format, resolution, and when the sample was acquired from the video device, respectively.

After a sample has been acquired and formatted, the acquisition microprotocol raises a MEDIAFROMUSER event. This event is handled by media encoder microprotocols and then finally by a media streaming microprotocol. Media encoder microprotocols, such as the **H263FastEncoder** microprotocol, recode the data for transmission, frequently using some form of compression. As encoder microprotocols change the data in a message, they also change the attributes associated with the message, for example changing the `video_format` attribute from `YUV420P_FORMAT` to `H263_FORMAT`. A media streamer microprotocol, for example the **VideoStreamer** microprotocol, then uses the multimedia transport session to transmit the message to a CODEC session on a remote machine.

When a CODEC session receives a message from a multimedia transport session, a media streaming microprotocol such as the **VideoStreamer** microprotocol handles the MSGFROMNET event raised by the Cactus runtime. This microprotocol then formats the received message by attaching message attributes describing the format of the sample, if necessary.[1] After formatting the sample, the streamer microprotocol raises a MEDIAFROMNET event to notify other microprotocols that a new sample has been received. This event is handled by media decoding microprotocols and media output microprotocols. Media decoding microprotocols such as **H263Decoder** decompress and change the formatting information associated with the message as necessary, while media output microprotocols such as **X11ImageDisplayer** send these decoded media samples to appropriate output devices.

[1] Such formatting is usually *not* necessary, however, as these attributes are typically transmitted by the remote CODEC along with the media sample.

Microprotocol Type	Specific Microprotocols
Data Acquisition	FileAudioAcquisition FileVideoAcquisition Video4LinuxAcquisition
Media Encoder Microprotocols	H263FastEncoder H263FullEncoder
Media Decoder	H263Decoder
Combined Encoder/Decoder	FLACAudioCodec VorbisCodec
Media Streaming	AudioStreamer VideoStreamer AudioVideoStreamer
Data Output	AudioOutput X11ImageDisplayer VideoArchiver

TABLE 6.1. CODEC Microprotocols

6.1.3 CODEC Microprotocols

We have implemented a number of variants of the different types of microprotocols needed to acquire, compress, transmit, decompress, and output audio and video in real-time. These are shown in table 6.1 and described below.

Data Acquisition. The FileAudioAcquisition microprotocol reads audio from data files and periodically raises MEDIAFROMNET events that carry dynamic messages containing audio samples and associated formatting information. This microprotocol uses the Silicon Graphics Audio File library, allowing it to read a wide variety of different audio file formats [CM96]. The FileVideoAcquisition microprotocol serves a similar function for files containing a sequence of raw video images, although it does not include support for the wide range of formats supported by the FileAudioAcquisition microprotocol. The remaining data acquisition microprotocol, Video4LinuxAcquisition, was mentioned in section 6.1.2. This microprotocol uses the Video4Linux API to read video from Linux video devices, such as video cameras and TV tuners [V4L].

Video Encoding and Decoding. We have implemented several different video encoding and decoding microprotocols for processing the video data acquired using the microprotocols listed above. These microprotocols use the ITU-T H.263 video compression standard [ITU], a lossy video compression standard, to greatly reduce the amount of bandwidth required to send real-time video over the network. The H263Decoder microprotocol uses the H.263 reference library implementation to decode H.263-encoded video, while two different microprotocols, H263FastEncoder and H263FullEncoder both implement H.263 encoding.

Although they provide the same basic functionality, H263FastEncoder and H263FullEncoder use different compression libraries with different performance characteristics. H263FastEncoder uses an encoding library optimized for encoding speed but that only supports a subset of H.263 [Aal]. While this library does not use all of the features available in the H.263 standard, it can perform software encoding of video on relatively modest hardware. The H263FullEncoder library, on the other hand, uses Telenor's reference implementation of the H.263 video encoder. This library implements most of the features of the H.263 standard, allowing it to achieve better compression than the speed-optimized library used by the H263FastEncoder microprotocol. Unfortunately, this library is also slower, and frequently cannot encode data quickly enough to supply a real-time data stream. When real-time encoding is more important than high levels of compression, H263FastEncoder is more appropriate. On extremely fast hardware or in situations with no real-time constraints, however, H263FullEncoder is generally more suitable because it provides superior compression.

Audio Encoding and Decoding. For audio encoding and decoding, we have implemented two different microprotocols. The FLACAudioCodec uses the Free Lossless Audio Codec library to encode audio for lossless transmission over networks [FLA] . The VorbisCodec microprotocol, on the other hand, uses the lossy Ogg Vorbis compression standard as implemented by the reference implementation to encode and decode audio samples at a

variety of bitrates [OGG]. When initialized, VorbisCodec sets up 3 different encoding levels that can be used to transmit audio at configurable transmission rates, normally 64kbps, 128kbps, and 256kbps. By default, the 128kbps data rate is used for data transmission, although the encoding rate used can be changed dynamically if necessary. Section 6.1.4 describes the adaptation mechanism implemented by this microprotocol in more detail, and section 6.1.5 describes the rules used by the Cholla controller to select the appropriate audio encoding rate.

Media Streaming. The two main media streaming microprotocols in the current CODEC implementation are VideoStreamer and AudioStreamer. Both microprotocols handle MEDIAFROMUSER events and, if the media sample is the appropriate type, send the sample over a network connection to a remote CODEC session that contains matching media streaming microprotocols. These microprotocols also handle MSGFROMNET events so that they can raise MEDIAFROMNET events when they receive a new sample on a network connection. The VideoStreamer microprotocol uses a Cholla-controlled CTP connection that includes SCP congestion control rules, lossy FIFO packet delivery order, and no reliability microprotocols. The AudioStreamer microprotocol uses a similarly-configured CTP connection, but also includes the ForwardErrorCorrection microprotocol in the CTP sessions it creates. In addition to these streaming microprotocols, we have also designed an AudioVideoStreamer microprotocol that multiplexes audio and video samples on the same connection.

Data Output. Finally, we have developed three different microprotocols for outputting media data to different destinations. The simplest of these microprotocols, the VideoArchiver microprotocol, stores received video data as uncompressed YUV images on disk. The X11ImageDisplayer microprotocol, in contrast, sends received video frames to an X11 window for immediate playback. Finally, the AudioOutput microprotocol can send received audio data to either a live hardware device for immediate playback or to disk for archiving.

values are ignored by VorbisCodec, but are used by the Cholla audio control rules described below in section 6.1.5. In particular, these fractional values allow the controller to slowly accumulate multiple small changes in `VorbisCodec.Codebook`, a change from 1.23 to 1.31, for example, even though such changes do not immediately affect the actual codebook used.

Of course, additional audio adaptation mechanisms could be added to the CODEC. For example, the audio acquisition microprotocols could adapt the rate at which audio samples are acquired or the size of these samples, or more aggressive voice-related audio compression algorithms could be used. In addition, once the reference implementation of the Ogg Vorbis encoder supports *bitrate peeling*, the ability to dynamically degrade a compressed stream with minimal loss of audio quality and without changing codebooks, the VorbisCodec microprotocol could be upgraded to use this method of adaptation instead of using multiple codebooks. This would remove many of the limitations on how and when this microprotocol adapts.

Video Adaptation Mechanisms. The video-related adaptation mechanisms in the CODEC are, in contrast, much more flexible than the existing audio mechanism. Four different video-related microprotocols contain mechanisms that can be used to adapt video behavior: Video4LinuxAcquisition, FileVideoAcquisition, H263FastEncoder, and H263FullEncoder. The FileVideoAcquisition and Video4LinuxAcquisition can both change the rate at which they generate video frames, resulting in a change in the amount of data to be compressed and transmitted by other CODEC microprotocols. The FileVideoAcquisition microprotocol accomplishes this by discarding incoming video frames occasionally, while Video4LinuxAcquisition accomplishes this by changing the rate at which the attached camera acquires frames. Cholla sets the desired framerate using the `ImageAcquisition.TargetFramerate` variable. Video4LinuxAcquisition can also change the resolution of the acquired images, something that FileVideoAcquisition cannot do; video resolution is controllable using the `VideoAcquisition.Resolution` variable.

The H263FastEncoder and H263FullEncoder microprotocols both adapt by changing the parameters used to compress video data. As already mentioned, both of these microprotocols use the compression algorithm defined by the H.263 compression standard. This algorithm employs two different types of compression: *spatial compression* and *temporal compression.*

Spatial image compression compresses each individual image by grouping pixels that are close together in the image into $N \times N$ pixel *blocks* and compressing each block. Blocks are compressed by transforming the data in each block into the frequency domain using a discrete cosine transformation and then discarding the image coefficients associated with high-frequency data. The size of each block, N, is the amount of *quantization* of the image; this parameter can be changed from image to image. As quantization increases, more pixels are grouped together into each block and fewer blocks are generated in each image, resulting in more compression but more discarded image information and poorer image quality. Quantization is exported to Cholla though the `Encoder.Quantization` variable.

Temporal compression in H.263 works by exploiting similarities between sequences of video image frames. When encoding a temporally compressed image, the video encoder predicts what it expects the next image to look like based on the previous images it has encoded. It then compares this prediction with the actual image being compressed and generates data describing the difference between the predicted and actual frames. Images that have been compressed with both spatial and temporal compression are referred to as *p-frames*, or predictive frames, while images compressed using only spatial compression are referred to as *i-frames*, or image frames.

To decode a p-frame, a H.263 decoder must know the contents of the previous image upon which the prediction was based. This requires it to see at least one i-frame before the first p-frame. When frames are lost, p-frames are decoded using older images, which can cause errors in the decoded p-frames. These errors will manifest as small visual differences between the encoded and decoded frame, and this error can accumulate as the

new, slightly incorrect frame is then used to decode another incoming p-frame. Because of this, the CODEC's H.263 encoding microprotocols periodically force the transmission of an i-frame to resynchronize the decoded image at the receiver with what is encoded at the transmitter. The rate at which these i-frames are generated is configurable through the `Encoder.IFrameRate` variable; higher i-frame rates result in higher data rates as less temporal compression is used. Increased rate of i-frame transmission does, however, also prevent the accumulation of image errors due to lost p-frames. In high-bandwidth, lossy networks, high i-frame rates are desirable, while in low-bandwidth, relatively reliable networks, low i-frame rates are desirable.

6.1.5 General CODEC Control Strategy

We have implemented rule sets for controlling the audio and video adaptation mechanisms described in the previous section and for coordinating adaptation in those mechanisms with adaptation in CTP. Our current control strategies focus on fully utilizing bandwidth provided by the multimedia transport session, while avoiding generating data more quickly than it can be transmitted. This is important because if the CODEC supplies data too slowly, it may be sending lower-quality data than the network connection could support. On the other hand, if CODEC microprotocols generate data too quickly, this data will be queued for transmission by CTP, resulting in increased end-to-end latency.

The rule sets we have designed so far examine the number of segments queued for transmission between the CODEC and CTP, and adjust the rate at which the CODEC generates data based on the difference between the actual and desired number of segments queued. We currently use 50 segments as the target, although this set-point could easily by changed. Several variables contain information about the length of this queue:

- `QueueError` contains the difference between the desired and actual number of segments in the queue. Unfortunately, this measurement is relatively noisy because

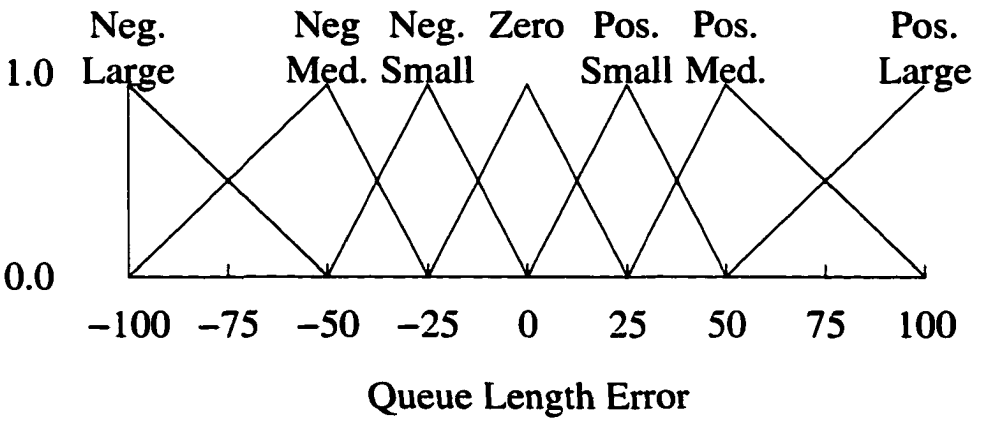


FIGURE 6.3. `QueueError` Fuzzy Set Membership Mapping

the queue is continually being filled and drained as new data is generated and new acknowledgments arrive in CTP.

- `QueueErrorInt` and `QueueErrorDrv` contain the integral and derivative of `QueueError`, respectively.

- `QueueDrainRate` contains a running average of the rate at which segments are being removed from the queue.

The audio control rules reference one of these variables, `QueueError`, as a fuzzy set, while the video control rules use the numeric values of these variables, not their fuzzy representation. Figure 6.3 shows the fuzzy set elements that comprise the `QueueError` variable, along with the mapping from crisp error values to set memberships.

CTP-specific input variables can also be used to coordinate CODEC data generation with CTP transmission either implicitly or explicitly. `SegmentTransmissionRate`, for example, contains a measurement of the rate at which CTP is sending segments. Other CTP-specific variables, such as `ForwardErrorCorrection.N`, have already been described in chapter 5.

The coordination rule set shown in figure 6.4 is responsible for modifying `VorbisCodec.Codebook` to take into account changes in the amount of forward error correction used by the CTP multimedia transport session. Because forward error correction requires sending extra packets containing redundant data, changes in the amount of forward error correction directly affect the amount of bandwidth available for transmitting CODEC segments. The coordination rule set shown in figure 6.4 computes the percent change in sending rate caused by the requested change in FEC level, and then converts this percentage into an appropriate change in `VorbisCodec.Codebook`.

Together, these rule sets implement an adaptation strategy for Vorbis audio compression that attempts to select the correct compression level to use available network bandwidth. Because of limitations in the adaptation mechanism, this adaptation strategy is relatively conservative, although it does include rules for coordinating CODEC adaptations with CTP adaptation. Section 6.3.2 presents experiments that show the effectiveness of the audio adaptation and coordination rules presented in this subsection.

6.1.7 Video Control Strategy

General Approach. Compared to the control parameters in the VorbisCodec microprotocol, the various adaptation mechanism in the video-related microprotocols are much more flexible. The CODEC's adaptable video-related microprotocols, the image grabber and H.263 compression microprotocols, can change video acquisition rates and compression parameters after every encoded video frame. In addition, small adjustments in the parameters associated with these microprotocols take effect immediately, unlike the codebook parameter used by VorbisCodec microprotocol, where small adjustments must accumulate over a time before causing any actual change in the rate at which data is generated.

This flexibility in the video adaptation mechanisms allows Cholla to use a much more aggressive strategy for controlling the video transmission rate in which parameters are adjusted much more frequently. While the audio adaptation strategies described in the previ-

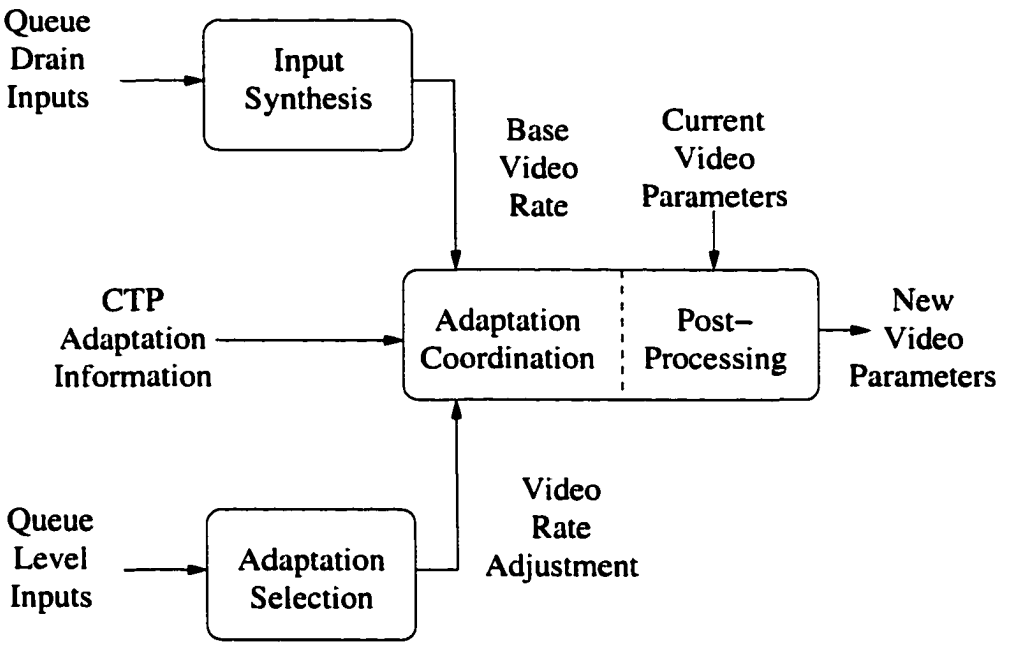


FIGURE 6.5. Video Adaptation Controller Phases

% Direct Queue Drain Rate Measurement Rule Set
BaseSegmentRate ← **QueueDrainRate**
% CTP-based Drain Rate Measurement Rule Set
PcntRedundantIn = 1 - (**fecKin** / **fecNin**)
BaseSegmentRate ← **CTPSegmentSendRate** × (1 - **PcntRedundantIn**)

FIGURE 6.6. Video Adaptation Input Synthesis Rule Sets

post-processing portion of the adaptation coordination phase then uses this value to derive appropriate video control parameters. The remainder of this subsection describes these phases in detail and presents the rules used in each.

Input Synthesis. The input synthesis phase of video control is responsible for one thing: determining the rate at which segments are being drained from the segment queue and using this to determine a base video transmission rate. Figure 6.6 shows the two different input synthesis rules used by Cholla. The first rule sets the base sending rate to the average rate at which CODEC segments have recently been drained from the segment queue. While this

% PI Queue Length Control Rule Set
Gain = -0.1
IntegralCoeff = 0.1
SegmentRateTrim ← **Gain** × (**QueueError** + **IntegralCoeff** × **QueueErrorSum**)

% Video Adaptation/CTP Forward Error Correction Coordination Rule Set
PcntRedundantIn = (1 - **fecKin**) / **fecNin**
PcntRedundant = (1 - **fecK**) / **fecN**
deltaSegmentRate = (**PcntRedundantIn** - **PcntRedundant**)/(1 - **PcntRedundantIn**)
SegmentRateBase = **SegmentRateBase** × (1 + **deltaSegmentRate**)

FIGURE 6.7. Video Adaptation Selection and Coordination Rule Sets

measurement is normally accurate, it does not always reflect recent changes in the rate at which the queue is being drained. For example, if CTP starts sending redundant data for error correction purposes, this will reduce the rate at which the queue is drained. It may, however, take several seconds for this change to be reflected in the average queue drain rate measurement. To avoid this problem, the second rule set shown in figure 6.6 can be used instead; this rule set uses information about the rate at which CTP is sending segments coupled with the current amount of forward error correction to estimate the rate at which the segment queue will be drained. The implicit coordination with CTP provided by this second rule set provides a more accurate measure of the rate at which the segment queue drains, although if the CODEC microprotocol is used with a transport protocol other than CTP, the first rule set is the only option.

Adaptation Selection and Coordination. Figure 6.7 shows the rule sets used for adjusting the base sending rate computed in the input synthesis stage and for coordinating this sending rate with adaptation in CTP. As already mentioned, the first rule set implements a PI-style linear control strategy to trim the base sending rate established by CTP. We use PI-style control over other linear control strategies such as proportional/integral/derivative (PID) control, because the noise filtering properties of PI control are most appropriate for dealing with the rapidly fluctuating queue level.

The second rule set shown in figure 6.7 is responsible for coordinating CTP forward error correction adaptation with video adaptation. In particular, this rule set adjusts the base sending rate determined by the input synthesis phase based on the new amount of forward error correction proposed by the controller. This explicit coordination rule set supplements the implicit coordination provided by the second rule set in figure 6.6—that rule set determines the base sending rate based on the previous amount of forward error correction used, while this rule set adjusts the base sending rate based on the amount of forward error correction that will be used after the controller runs.

The post-processing rules in the adaptation coordination phase combine the send rate base and the queue trim value into a desired segment rate, and then use this rate to compute video parameters that generate this desired rate. Equation 6.1 gives the approximate relationship between the segment rate (R_S), video frame rate (R_F), video quantization (Q), and i-frame rate (R_I),

$$R_S = R_F \frac{S_P + R_I(S_I - S_P)}{Q^2} \tag{6.1}$$

S_I and S_P are, respectively, the size in segments of i-frames and p-frames when no quantization is used. These constants can be estimated from the average size of i-frames and p-frames and the current quantization level, although such estimates are imprecise.

There are of course infinitely many values of R_F, R_I, and Q that yield a given segment rate, R_S. Currently, Cholla implements rule sets that solve for a new value of one video parameter and leave the other parameters unchanged from the current values. These rule sets are shown in figure 6.8.

More sophisticated post-processing rules could also be used. Such rules would take into account other inputs such as desired CPU utilization and network loss rate, and attempt to find video parameters that achieve the desired segment rate and CPU utilization while maximizing perceived video quality. Construction of such post-processing rules is one potential area for future work.

% Desired Segment Rate Computation Rule Set
SegmentRateOut = SegmentRateBase + SegmentRateTrim

% Controlled Framerate Postprocessing Rule Set
FrameSizeDiff = IFrameSizeIn - PFrameSizeIn
EstFrameSize = PFrameSizeIn + IFrameRateIn × **FrameSizeDiff**
Framerate ← **SegmentRateOut / EstFrameSize**
Quantization ← **QuantizationIn**
IFrameRate ← **IFrameRateIn**

% Controlled Quantization Post-processing Rule Set
IFrameOneSize = IFrameSizeIn × **QuantizationIn**^{2}
PFrameOneSize = PFrameSizeIn × **QuantizationIn**^{2}
UnquantFrameSize = PFrameOneSize
    + IFrameRateIn × **(IFrameOneSize - PFrameOneSize)**
Quantization ← sqrt(**FramerateIn** × **UnquantFrameSize / SegmentRateOut)**
Framerate ← **FramerateIn**
IFrameRate ← **IFrameRateIn**

% Controlled I-frame rate Post-processing Rule Set
DesiredFrameSize = SegmentRateOut / FramerateIn
FrameSizeDiff = IFrameSizeIn - PFrameSizeIn
IFrameRate ← **(DesiredFrameSize - PFrameSizeIn) / FrameSizeDiff**
Framerate ← **FramerateIn**
Quantization ← **QuantizationIn**

FIGURE 6.8. Video Post-processing Rule Sets

Even though they do not use sophisticated post-processing rules, these rule sets do implement an adaptation strategy for video acquisition and compression. In addition, these rules coordinate video adaptation with forward error correction in the underlying multimedia transport protocol. Section 6.3.3 presents experiments that show the effectiveness of these rules.

6.2 Wireless Network Proxy

In addition to the multimedia transmission application, we have also designed an adaptable proxy for wireless networks that uses Cholla. Wireless proxies are normally situated at the junction between wireless and wired networks. This places them in an ideal location for optimizing the performance of wireless applications by adapting the behavior of transport sessions that connect to the proxy and by recoding application data as it traverses the network. Such adaptation, however, requires careful coordination between a variety of system components. This section describes the motivation for and demands on wireless proxies, the architecture of a Cholla wireless proxy, and how adaptation in such a proxy could be controlled.

6.2.1 Motivation

Most standard transport protocols do not work well in a wireless environment but do work well on wired networks. Because of this, proxies are frequently used in mobile networking environments to deal with the differing requirements of wireless and wired connections [FGBA96, BPSK97, FGCB98]. In a proxy-based system, applications on wireless networks connect to a proxy instead of connecting directly to a server; the proxy then completes the connection by connecting to the appropriate server and forwarding packets received on each connection using the other connection. This allows wireless applications to use transport protocols optimized for wireless networks, while maintaining interoperability with existing services and wired networks where wireless-specific protocols are not appropriate.

A proxy can also perform services besides splicing together two different types of transport connections. For example, two common difficulties in mobile and wireless systems are the lack of battery and computational power on mobile devices, and the widely varying characteristics of wireless networks. A proxy can help with both of these problems. By off-loading a portion of transport-level forward error correction from the client, a proxy

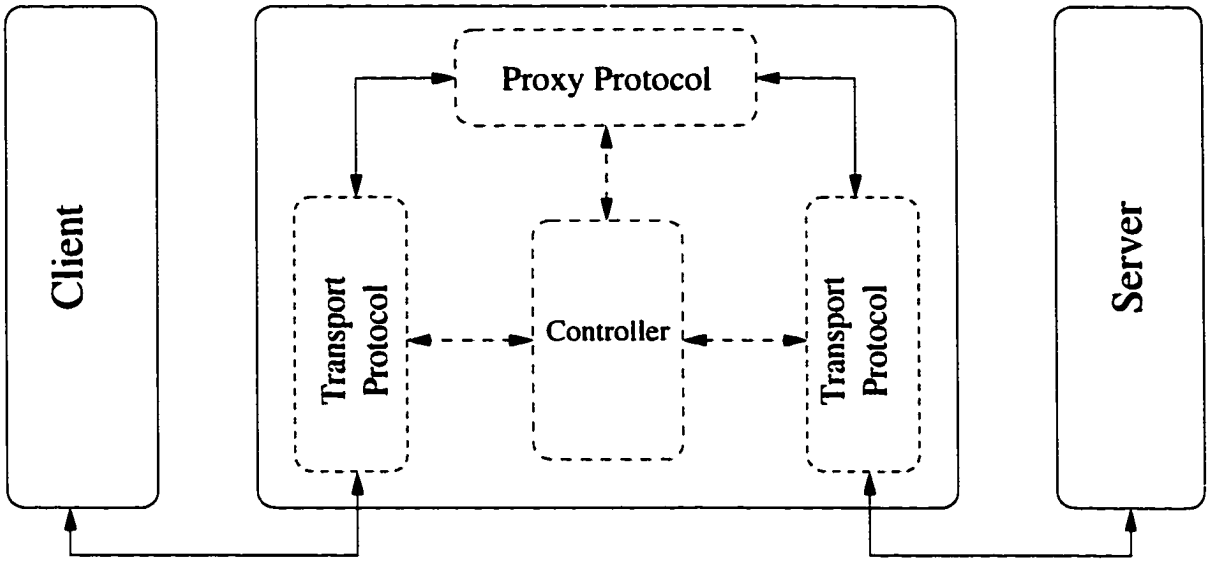


FIGURE 6.9. Proxy Overview

can help conserve CPU and battery resources on a mobile device; by recoding content like images, audio, and video, a proxy can also help conserve scarce wireless network bandwidth.

Cholla provides an ideal framework for implementing a general wireless proxy service. Such a service must be able to support a wide range of transport protocol semantics and application adaptations; Cholla's support for fine-grained configuration and adaptation is designed for exactly this purpose, as is its support for coordinating adaptation between multiple system components. Figure 6.9 shows an overview of such a proxy implemented using Cholla. In this architecture, Cholla is responsible for coordinating adaptation in multiple transport connections and in data recoding done by the proxy.

6.2.2 Cholla Proxy Architecture

The details of the proxy architecture are shown in figure 6.10. Clients connect to the proxy using one of several different possible protocols, typically TCP or CTP. These connections are handled by a proxy protocol that includes two different types of sessions, relay sessions and forwarder sessions. *Relay sessions* encapsulate general information about connection

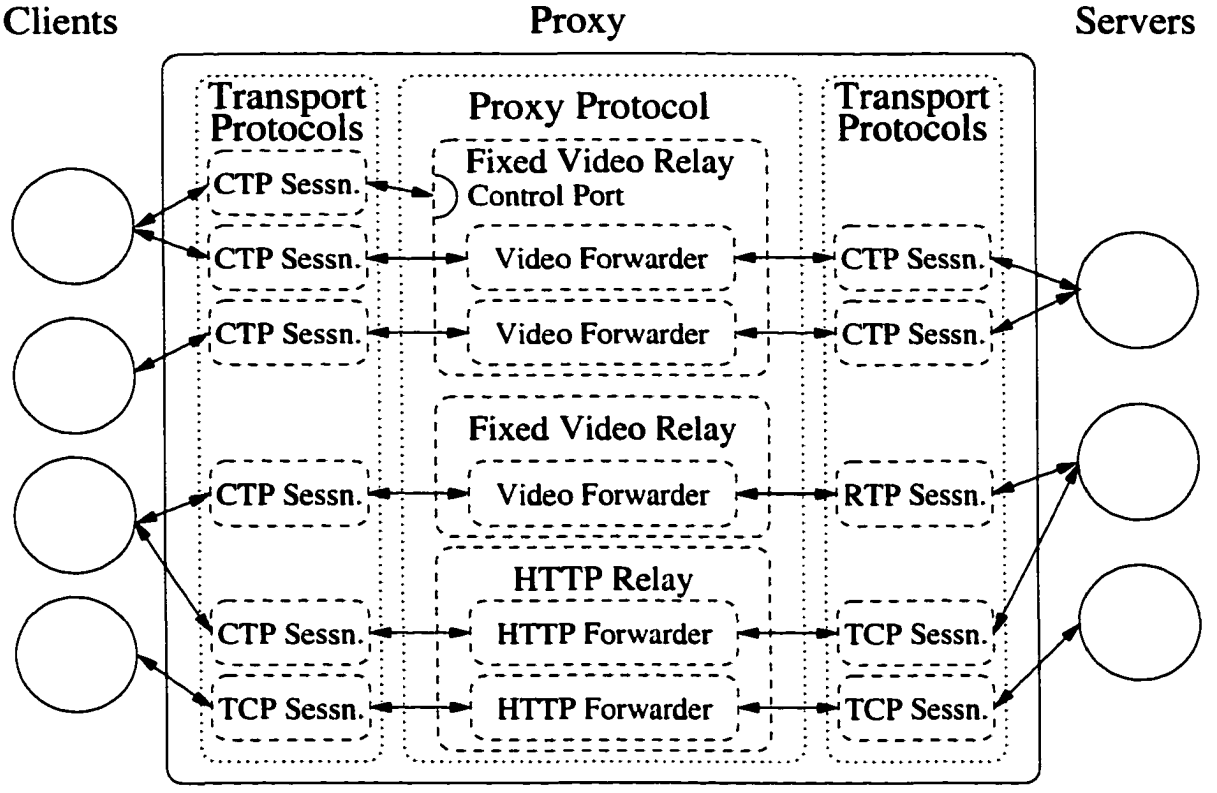


FIGURE 6.10. Proxy Architecture

creation and content management for a particular application. Applications create new relay sessions customized to their needs by connecting to the proxy control port and specifying what type of relay they need. The resulting relay session then listens for incoming connections on the ports specified by the application and creates *forwarder sessions* in response to new connections. These sessions process and forward the data associated with a particular connection and, as such, perform most of the actual work in the proxy.

Like CTP and the CODEC, relay and forwarder sessions in the proxy are structured as collections of microprotocol instances that interact using events. This allows the semantics of each relay and forwarder session to be customized by selecting the microprotocols it contains. The proxy contains several different kinds of microprotocols, as follows.

- **Connection management microprotocols** are used by relay sessions to complete a connection to the appropriate destination when a new connection is opened. The FixedConnection microprotocol, for example, opens a connection to a pre-configured destination when a new connection is established, while the HttpConnection microprotocol parses the incoming HTTP request to find the appropriate destination.

- **Data forwarding microprotocols** are used to forward application data in application-specific units. The MessageForwarder microprotocol, for example, immediately starts recoding and forwarding received messages, while HttpDocumentForwarder collects incoming data into complete HTTP documents before starting recoding or forwarding.

- **Data recoding microprotocols,** such as the HttpImageRecoder and VideoFilter microprotocols, recode data collected by the data forwarding microprotocols before it is forwarded.

An application that wishes to use the proxy starts by connecting to the proxy's control port. This creates a new relay session that the application can configure to serve as a template for the creation of future forwarder sessions. Using control messages, the application selects a connection management microprotocol for the relay session along with the microprotocols it wants in forwarder sessions created by the relay. It then instructs the relay to listen to a TCP or CTP port for new connections. When a new connection is made, the NEWFORWARDER event is raised in the relay session. The relay's connection management microprotocol handles this event, makes a connection to the appropriate destination, and adds this connection to the newly-created forwarder session. Other microprotocols in the relay session such as data management and recoding microprotocols create new instances of the same kind of microprotocol and add them to the forwarder session as well, so these microprotocols can process data received by that forwarder. Finally, the Cactus framework raises a MSGFROMNET event in the newly-created forwarder, which processes the message and forwards it to the appropriate destination.

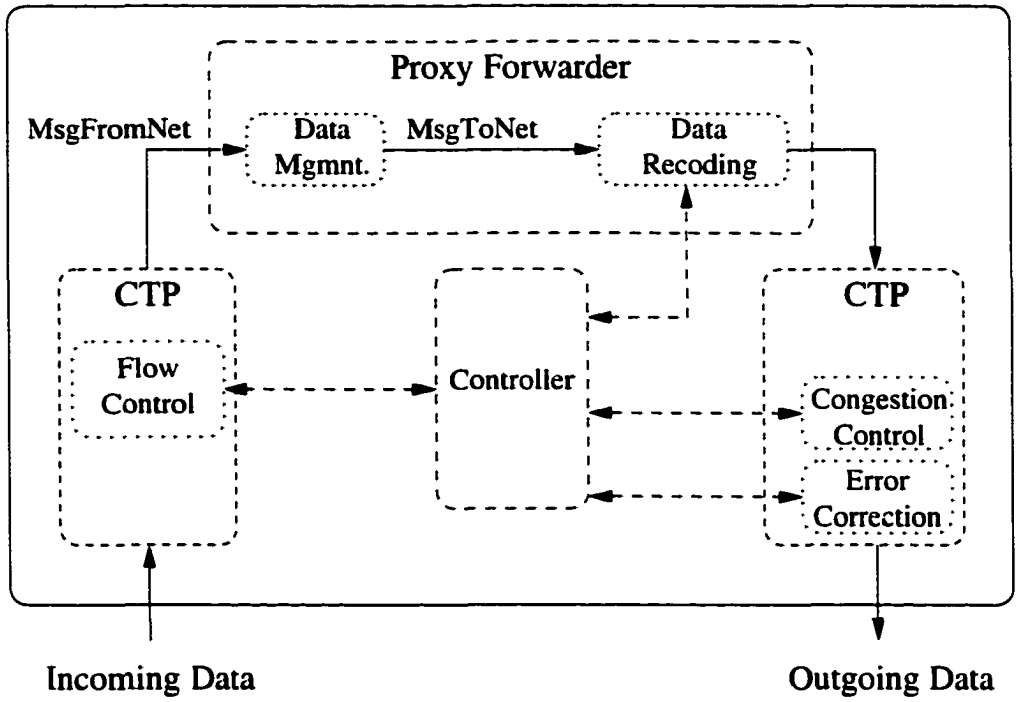


FIGURE 6.11. Example Forwarder Session

Figure 6.11 shows an example of how a forwarder session in the proxy processes data and interacts with other system components. The rate at which a proxy forwarder accepts incoming data is determined by CTP's flow control mechanisms. After a packet is received, a MSGFROMNET event is raised. These events are handled by data management microprotocols, which combine the incoming messages into units suitable for recoding by later microprotocols. After a complete unit is received, the data management microprotocol raises a MSGTONET event. Data recoding microprotocols recode the message associated with this event, and then set send bits on the message when recoding is complete. After all of the send bits are set, the recoded message is forwarded to its destination using another CTP session, which may queue the message prior to transmission. This CTP session can also perform congestion control or include additional error correction information as necessary.

Adaptation	Affects
Amount of error correction on outgoing connections	Proxy CPU, Client CPU, Available network bandwidth, Quality of data received by client
Congestion window on outgoing connections	Congestion in network, Number of segments buffered by CTP
Recoding parameters	Proxy CPU usage, Client CPU usage, Number of segments buffered for transmission by CTP, Quality of data received by client

TABLE 6.2. Adaptations in the Proxy

6.2.3 Adaptable Components

Adaptation of several of the components show in figure 6.11 can be controlled by Cholla. In particular, the CTP flow control, congestion control, and forward error correction microprotocols contain adaptable mechanisms, as do the data recoding microprotocols in the proxy. Cholla manages each of these mechanisms to control a number of different parameters, including:

- CPU utilization on the proxy
- CPU utilization on the client
- Number of segments buffered by CTP for transmission
- Quality of the data received by the client
- Rate at which new data arrives at the proxy

Table 6.2 shows the various adaptations controlled on each connection in the proxy, and what each adaptation affects.

The CTP congestion control and error correction mechanisms were described in chapter 5, and these mechanisms are controlled using the rule sets described in chapter 5. CTP's

% PI Control of Output Rate
TargetOutRate ← **QueueDrainRate** +
    Gain × (**QueueError** + **IntegralCoeff** × **QueueErrorSum**)

% Input Rate Adjustment
if **CPUUsage** is *high* then **TargetInRate** ← **CurrentInRate** × 0.75
if **CPUUsage** is *medium* or **CPUUsage** is *low*
    then **TargetInRate** ← **CurrentInRate** × 1.0

% Input/Output Mismatch Coordination
RateDifference ← **TargetRateIn** - **TargetRateOut**
if **RateDifference** is *high* then **TargetRateIn** = **TargetRateIn** × 0.8
if **RateDifference** is *medium* then **TargetRateIn** = **TargetRateIn** × 0.9

FIGURE 6.12. Example Proxy Rule Sets

the target output rate into recoder-specific parameters. The second rule set attempts to control the amount of CPU usage in the proxy by throttling incoming connections. Finally, the third rule set attempts to offload work from the proxy by slowing the sender if the difference between the proxy input and output rates is large.

Other rule sets that attempt to take into account CPU utilization on the client or quality measurements could also be used, as could coordination rules similar to those used for the multimedia transmission application. In addition, recoders with large CPU requirements will have to have their CPU usage coordinated with CPU utilization by CTP's error correction mechanisms.

6.3 Experimentation

6.3.1 Setup

We ran a series of experiments to validate the control strategies for the multimedia application described in section 6.1 and to study the effectiveness of the adaptation coordination provided by Cholla. Experiments were run between two 600MHz Pentium III machines,

each with 128MB of RAM, connected by 100 megabit per second twisted-pair Ethernet. Each experiment was run several times on unloaded machines to ensure that system behavior was consistent, and a representative graph of the system behavior was then constructed.

We used the Linux tool `tc` and the token bucket filter (TBF) kernel module to limit outgoing network bandwidth when necessary. As in the experiments described in chapter 5, both machines were running RedHat Linux release 6.2, and the system was built using the C version of Cactus 2.0. JFS was again used as the fuzzy inference engine [Mor00]. The Cholla controller was set to run in unclocked mode but at a minimum rate of 30 times per second. For the audio adaptation experiments, this results in changing the audio parameter no more than once per second because of limitations in VorbisCodec. For video adaptation, on the other hand, unclocked control results in changing the video parameters prior to encoding of each frame—potentially as often as 30 times per second.

6.3.2 Audio Adaptation and Coordination

Audio Adaptation Control. For the audio adaptation mechanisms and policies, we examined how adaptation of the codebook used by the VorbisCodec microprotocol affected the number of segments queued between the CODEC protocol and the CTP multimedia transport session. In particular, we focused on how effective explicit coordination rules were for coordinating VorbisCodec adaptation with adaptation in the amount of forward error correction used by CTP.

For these experiments, we streamed the 3 minute and 10 second song "I Am a Man of Constant Sorrow", track seven from the soundtrack to "O Brother, Where Art Thou?". This clip was encoded using the VorbisCodec microprotocol and sent over a network connection with the transmission rate limited to 500 kilobits per second, since the various Vorbis codebooks compressed data to between 64kbps and 256kbps depending on which codebook was used. This rate limit was sufficient to allow 64kbps and 128kbps compression rates used by codebooks zero and one to flow easily, but to sometimes cause queuing at the

- FileAudioAcquisition
- VorbisCodec
- AudioStreamer
- AudioArchiver

(a) CODEC Microprotocols

- Vorbis Codebook Adaptation
- Vorbis Codebook/CTP FEC Coordination

(b) CODEC Rule Sets

FIGURE 6.13. CODEC Microprotocols and Rule Sets used for Audio Experiments

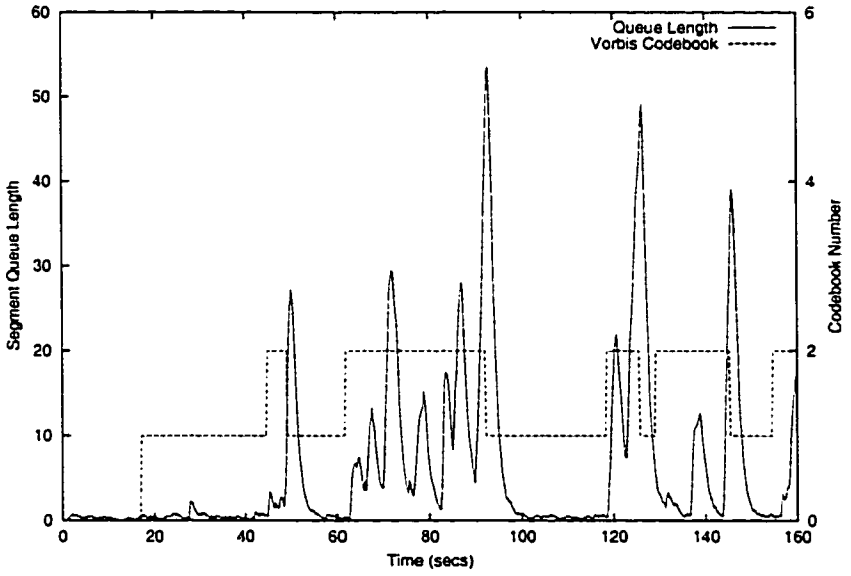


FIGURE 6.14. Audio Control with No Added Loss

256kbps compression level used by codebook two, especially when the forward error correction microprotocol sends redundant data. The CODEC protocol was configured to use the microprotocols and rule sets shown in figure 6.13, while the CTP multimedia session was configured to use the same microprotocols and rule sets as in the wireless multimedia scenario described in chapter 5.

Figure 6.14 illustrates the behavior of the CODEC audio adaptation policy in a network with no added loss. When few packets are queued, Cholla occasionally attempts to shift to a codebook that generates higher-quality output at the expense of increased data rates. When the network connection is unable to immediately transmit segments encoded at a

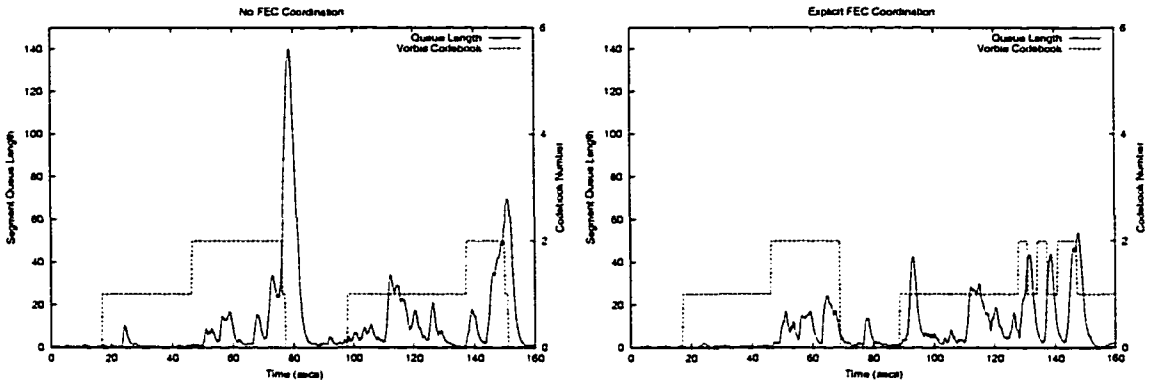


FIGURE 6.15. Audio Control with and without Coordination Rules

higher data rate, these segments are queued for transmission. This eventually causes the audio encoder to drop back a lower-quality compression level.

Audio/CTP Coordination. In addition to queuing caused by increased compression levels, queuing can also be caused by changes in the available network bandwidth, such as when CTP's ForwardErrorCorrection microprotocol transmits redundant data segments. To explore the effectiveness of Vorbis Codebook/CTP FEC Coordination rule set, we ran tests both with and without this rule set and added additional loss to the network part way through each run. Figure 6.15 shows the average queue length and codebook used during a run when the receiver starts dropping every fourth packet after 70 seconds and stops dropping packets 60 seconds later, at approximately t=130 seconds. The graph on the left shows the behavior of the system when no FEC coordination is used, while the graph on the right shows system behavior if the explicit FEC coordination rule is included in the rule sets used to control audio compression.

After extra loss is added to the network, the receiver reports this change to the sender, which then increases the amount of forward error correction used to include 2 redundant segments out of every 10 sent at $t = 70$ seconds, and then increases this to 3 redundant data segments out of every 10 at $t = 75$ seconds. The Cholla configuration that includes explicit coordination with changes in CTP forward error correction anticipates this loss of

- FileVideoAcquisition
- H263FastEncoder
- H263Decoder
- VideoStreamer
- VideoArchiver

(a) CODEC Microprotocols

- CTP Drain Measurement
- PI Queue Length Control
- Video/CTP Coordination
- One post-processing rule set

(b) CODEC Rule Sets

FIGURE 6.16. CODEC Microprotocols and Rule Sets used for Video Experiments

available bandwidth and immediately changes to a lower-quality compression codebook. The configuration that does not include the explicit coordination rule, however, changes codebooks only after queuing up a large number of extra segments, potentially resulting in large transient changes in end-to-end latency at the receiver.

6.3.3 Video Adaptation and Coordination

We have also run experiments to study the effectiveness of the CODEC's video control strategy. This strategy, which focuses on controlling the rate at which adaptable video components generate segments for transmission, can control a variety of different video parameters, depending on the post-processing rules used. We examined how effectively this strategy controlled framerate, video quantization, and i-frame rate in response to changes in the level of the segment queue between the CTP and CODEC protocols. In addition, we examined the effectiveness of implicit and explicit coordination when CTP uses forward error correction.

Figure 6.16 shows the microprotocols and rule sets used for the video experiments. Along with the rule sets listed in this figure, the video experiments also used the CTP's SCP congestion control rule sets, listed in figure 5.10. For video experiments that used compression, we limited the network bandwidth to 500 Kbps and set the default video parameters to 30 frames per second, quantization of 3.0, and an i-frame rate of 0.2 (one

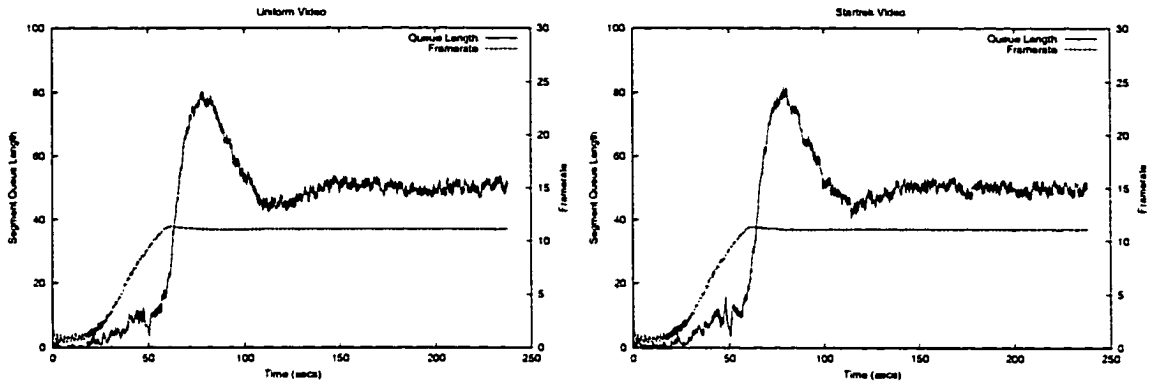


FIGURE 6.17. Framerate Control in Uncompressed Video

i-frame out of every 5 frames compressed.) For experiments that did not use video compression, we removed the H263FastEncoder microprotocol and instead limited network bandwidth to 5 Mbps.

We used two different videos to test video transmission. This first video, the *uniform* video, consists of a single repeated QCIF-resolution (176 by 144) video frame, with no motion or variation between frames. This video was used to factor out variations in the compressibility of video from variations due to control actions. The second video, *startrekds9* is a 46-second QCIF-resolution clip from the introduction to Star Trek: Deep Space Nine television show that includes segments with little motion or detail, segments with panning and other motion, and several scene changes. For tests longer than 46 seconds, this video was cycled through multiple times.

Video Adaptation Control. Figure 6.17 shows the how the video control strategy described in section 6.1.7 controls the video frame rate when sending uncompressed video. Note that the behavior of framerate control on the *startrekds9* video is essentially identical to that of the uniform video shown in figure 6.17, since no compression is used and the size of an uncompressed YUV420-encoded QCIF video frame is always 38,016 bytes. Because we use PI-control, there is an expected mild overshoot of the control point at startup. After this, however, the queue level settles at the desired 50-segment level. In contrast, figure

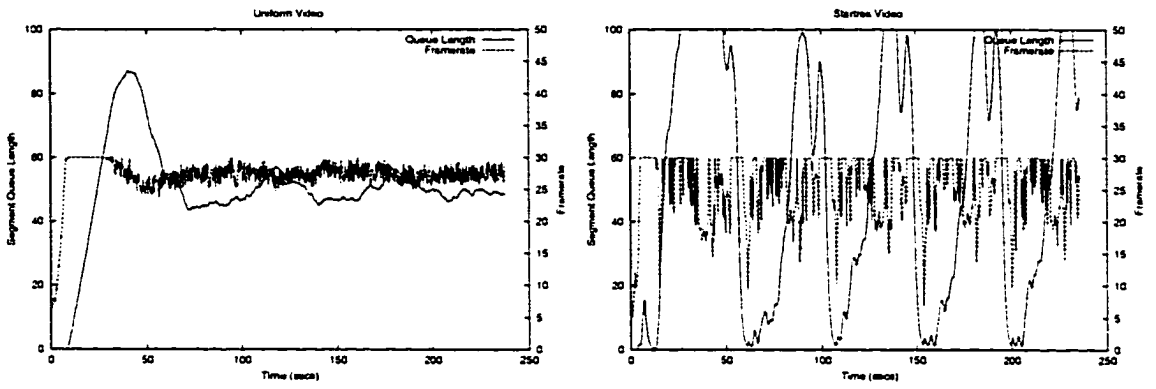


FIGURE 6.18. Framerate Control in Compressed Video

6.18 shows the results of framerate control in compressed video with both the *uniform* and *startrekds9* videos. Control of the *uniform* video is very similar to that of the uncompressed video shown in figure 6.17; the *startrekds9* video shows much larger variation that repeats every 46 seconds as the sender cycles through the video.

The variation in the queue length when controlling framerate in *startrekds9* is due to frame-to-frame variations in the compressibility of the video and because the controller cannot increase the framerate beyond 30 frames per second. Because Cholla cannot predict how compressible the next frame will be, it cannot compensate for this variation. Despite the variation in queue length, the variation in frame rate is not large, so buffering on the receiver can normally absorb this variation, although absorbing variation does require more buffering at the receiver than when such variation is not present. It is also important to note that without any sort of adaptation, the multimedia application would normally either underutilize the network connection or cause even greater queuing to occur when it attempted to continually send faster than the connection allowed.

Figure 6.19 shows the effects of controlling, respectively, the quantization and i-frame rate used by the H263FastEncoder microprotocol for encoding the *uniform* video. Unlike framerate control, queue length control in these cases results in oscillatory behavior. In the case of quantization, this oscillation is relatively mild, with the quantization used after startup ranging between none and using 3x3 blocks. This oscillation occurs in quantization

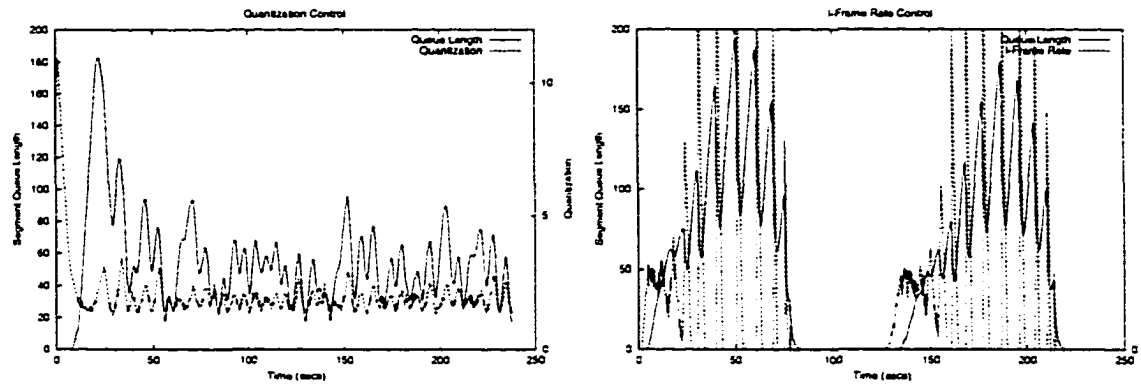


FIGURE 6.19. Quantization and I-Frame Rate Control

control because the relationship between these variables and the segment rate is more complicated than the linear relationship between the video frame rate and the resulting segment rate. In particular, changing the amount of quantization used changes the average size of i-frames and p-frames. The video control rule sets must estimate the size of unquantized i-frames and p-frames from the average size of current i-frames and p-frames and the current quantization level, so changes in the quantization can introduce errors in these estimates. These errors then result in errors in the control decisions made by the controller, resulting in the oscillatory behavior shown in figure 6.19.

The i-frame rate shows much more dramatic oscillation. This oscillation occurs because changes in i-frame rate also change the average rate that CTP sends segments. In particular, changes in the i-frame rate cause more p-frames to be sent. Most p-frames occupy only one segment and these segments, unlike segments containing i-frame data, are frequently much smaller than the maximum segment size of 1500 bytes. Variations in the rate at which short segments are sent cause changes in the rate at which CTP sends segments. This in turn causes changes in the rate at which i-frames are generated, leading to the oscillatory behavior shown in figure 6.19.

Unfortunately, such feedback effects are difficult to model and control. The most straightforward approach to dealing with these effects would be to lower gain used in the PI control strategy; this, however, would lower the responsiveness of the system and the

overall effectiveness of framerate control. Other possible approaches include using more conservative approaches to quantization and i-frame rate control, or using other less volatile parameters to control quantization and i-frame rate, with only the framerate used to adjust the level of the segment queue.

Video/CTP Coordination. In addition to testing the effectiveness of general video adaptation, we also tested the effectiveness of implicit and explicit coordination of these adaptations with CTP forward error correction. While forward error correction is not normally used with video transmission because the human eye is not as sensitive to lost video data as lost audio data, it can still be useful in networks with large amounts of loss. In these experiments, we compared control of the framerate with no coordination rules when sending the *uniform* video to control when using the implicit and explicit coordination rule sets shown in figures 6.6 and 6.7.

Figure 6.20 shows the effect of using implicit coordination rules to assist video framerate control. In these experiments, additional loss was added at the receiver after the framerate control had achieved steady state. Specifically, the receiver started dropping every fourth segment at $t = 130$ seconds. At this point, CTP starts sending 3 redundant segments out of every 10 segments, and increases this to 4 redundant segments out of every 10 at $t = 135$ seconds. Note that the graphs start at $t = 100$ seconds, long enough for the queue to have stabilized at 50 segments prior to testing coordination behavior.

Both with and without implicit coordination, a temporary *increase* in the rate at which CTP sends packets occurs when additional loss is added to the network, causing the segment queue in both experiments to empty. This occurs because the receiver is actually CPU-limited by video decoding, causing it to acknowledge packets only as quickly as it can actually decode video frames. With additional loss added, however, dropped frames do not have to be decoded, causing the receiver to be able to acknowledge segments more quickly.

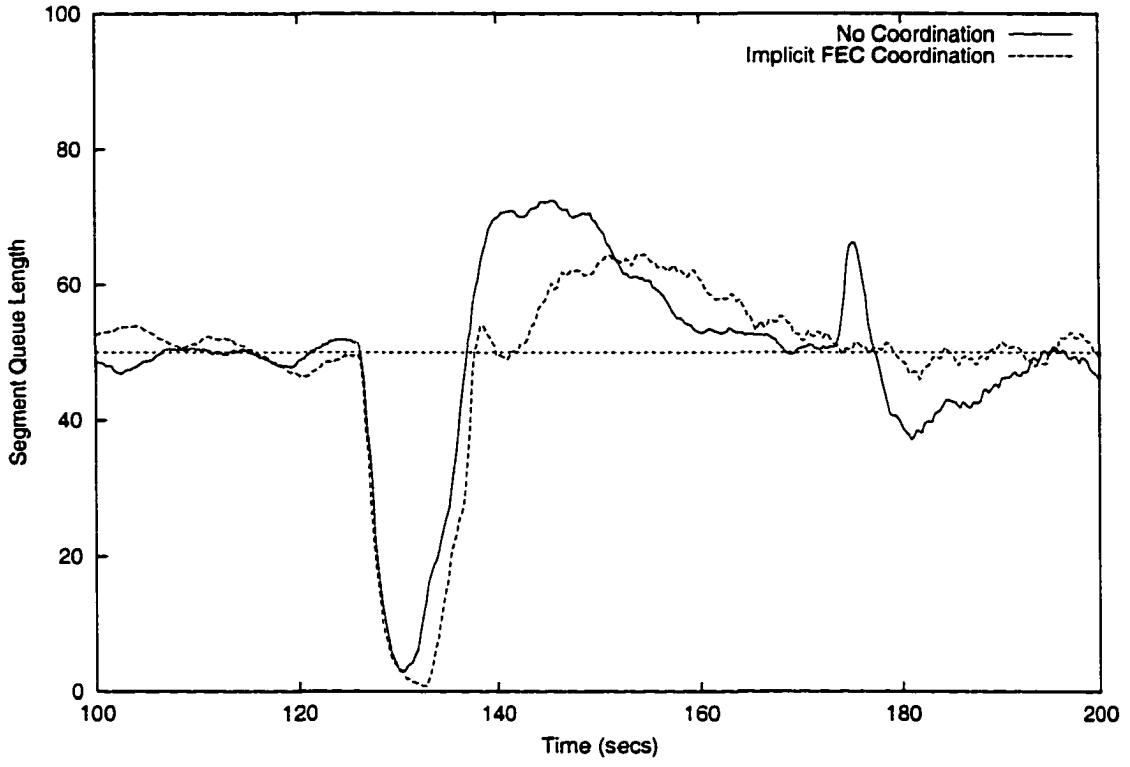


FIGURE 6.20. Framerate Control with and without Implicit Coordination Rules

After this temporary drop in the queue level, both uncoordinated and implicitly coordinated control rules again increase the queue level based primarily on their estimate of the rate at which CTP is draining the queue. The experiment without implicit coordination estimates this rate using a running average of the rate at which CTP drains the queue. In contrast, the implicit coordination experiment estimates this rate using the current amount of forward error correction used by CTP and a running average of the rate at which CTP is sending segments. Because the CTP sending rate changes less after loss is added than the direct queue drain rate measurement, the experiment that included implicit coordination converges to the correct queue level with less overshoot faster than when no coordination is used.

Figure 6.21 shows the effect of supplementing the implicit coordination rules with explicit coordination rules. Control in this situation is dominated by estimation of the queue

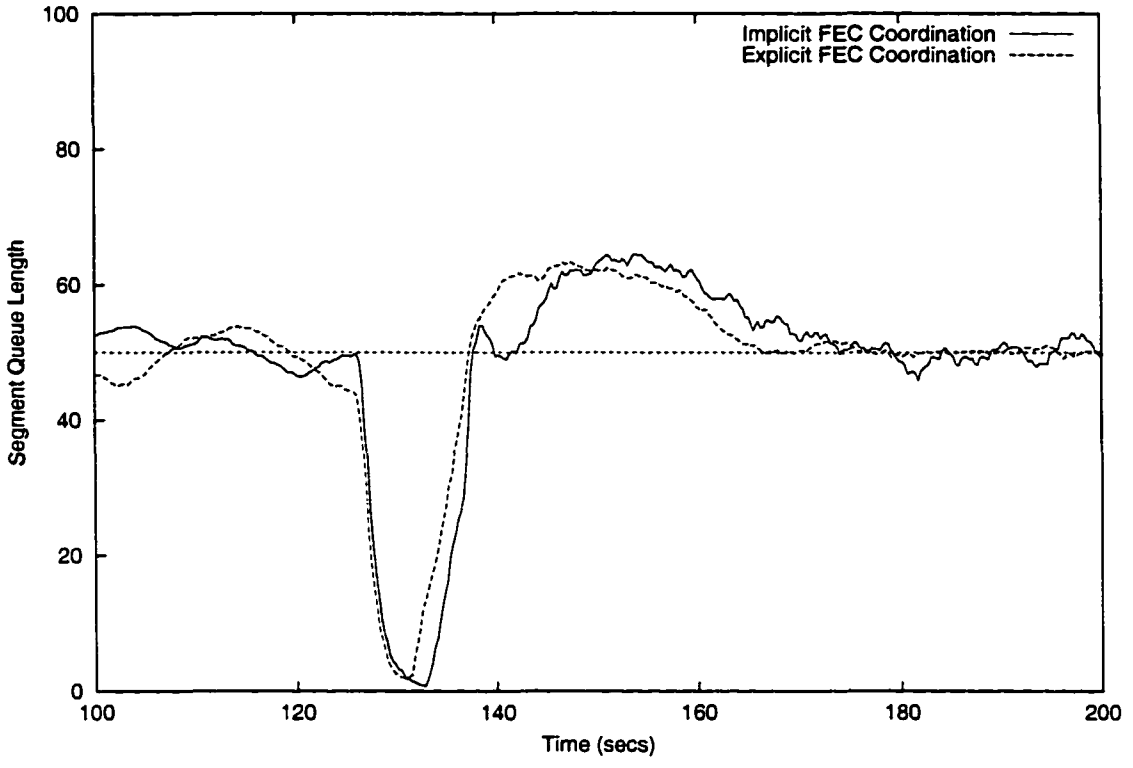


FIGURE 6.21. Framerate Control with and without Explicit Coordination Rules

drain rate, so explicit coordination provides only marginal improvements. Nonetheless, explicit control does return the sending queue to the proper level somewhat more quickly than if no coordination is used. In situations where the receiver is not CPU-limited and additional loss does not result in changes in the rate at which the queue is drained, the benefits of explicit coordination should also be greater.

6.3.4 Summary

Overall, these experiments show how Cholla can be used to control an adaptable multimedia application and to coordinate adaptation between multiple components. Explicit coordination in audio adaptation is very successful in preventing excess queuing of packets when forward error correction changes, and implicit and explicit coordination in video transmission is also useful, although not to the degree seen in audio transmission. This dif-

ference is due to differences in the overall control strategies—the video control examples use an aggressive linear control strategy that is able to quickly compensate for large-scale changes in system behavior. Audio control is much more sensitive, however, and must be controlled much more conservatively. Explicit coordination is very useful in this case for adapting to changes that cannot be easily compensated for using audio's standard control strategy. We believe that this is true in general—explicit coordination is most useful in cases where large-scale changes in system behavior can happen and conservative control strategies are normally used because of limitations of the system being controlled.

6.4 Related Work

Other researchers have implemented applications similar to those described in this chapter, although none that we are aware of provide the kind of inter-component coordination or fine-grain configurability that are available using Cholla. For example, ARC has been used to implement adaptive video transmission, and also provides inter-component coordination between layers in network stacks using an iterative approach [vDLS00]. ARC does not, however, provide any mechanisms for fine-grain composition of adaptation policies, or for coordination between non-layered components. ARC also does not explicitly separate adaptation policy from adaptation mechanism, making it more difficult to modify and tune adaptation policies.

The SWiFT toolkit has also been applied to adaptation in multimedia systems [Cen97]. Like the control policies for multimedia transmission presented in this section, SWiFT uses linear control policies to control adaptation in a multimedia transmission system, although SWiFT does not address coordination between multiple adaptable components. Similarly, work on adaptation of tunable application [BNBH98a, BNBH98b, KHP99, CK01] and on adaptation in the Fugue system [CNW01] has also implemented adaptation in multimedia transmission systems, although none of these systems address fine-grain composition of control policies or coordination of multiple adaptable components.

Various wireless proxies have also been implemented. Systems have been developed for building scalable content-recoding proxies [FGBA96, FGCB98] and for deploying HTTP content recoders into the World Wide Web [SP02]. Proxies have also been used for improving the characteristics of transport connections[BPSK97]. None of these systems, however, address how to control or coordinate adaptations.

6.5 Summary

The applications presented in this chapter demonstrate how adaptation control and coordination are performed in Cholla, and illustrate the effectiveness of this approach. In the future, we plan to extend the video transmission application to take into account other system parameters besides queue length, including CPU utilization and network lossiness. We also plan to do additional work investigating more sophisticated post-processing rules and general control strategies. The proxy application, for which implementation and experimentation are in progress, offers even more opportunities to explore coordination between multiple system components while building a robust, configurable service that can handle many different types of data.

CHAPTER 7

A NETWORK EMULATION ENVIRONMENT

Wireless systems are one of the most important environments for adaptable applications and protocols. Large-scale wireless systems are, however, expensive to build, laborious to maintain, and difficult to control. Because of this, *network simulators* and *network emulators* have become important tools for studying mobile and wireless systems. Network simulators computationally model the behavior of a set of computers connected by networking devices, while emulators make existing computers and applications act as if they were connected by a different network than the one physically being used. Network emulators normally support more complex applications and protocols than network simulators, and as such, are generally more suitable for testing complex adaptive applications. Most previous approaches to emulation have, however, introduced more latency into the emulated network than would be present in a real network of the type being emulated, been restricted to emulating only high-latency networks, or have required the use of simulation results to drive emulation.

This chapter describes VWLAN, a network emulator that implements a virtual WaveLAN network device as a device driver in the Linux kernel. Unlike other emulators, VWLAN distributes responsibility for transmission and reception decisions to the individual machines participating in the emulation. This approach gives VWLAN more reasonable latency characteristics than other approaches, potentially allowing it to emulate lower-latency networks. However, it also introduces limitations that prevent it from accurately reproducing the bandwidth characteristics of the WaveLAN when the emulated network is heavily loaded.

This chapter is organized as follows. Section 7.1 provides a general overview of network simulation and emulation, compares the two approaches, and provides a general

overview of our approach to distributed network emulation. Section 7.2 follows with a description of the system on which the emulator is constructed. Section 7.3 then presents the detailed design of the emulator, while section 7.4 presents an evaluation of the emulator and its limitations, and presents possible directions for future work. Finally, section 7.5 summarizes the chapter.

7.1 Overview

Network simulators are powerful tools for studying network protocols. They give system designers complete control over the system being simulated and allow them to experiment with systems that would be impractical to construct in a lab. Network simulators do, however, have a number of limitations. They generally run slowly and usually only simulate the networking portion of the system instead of the entire system, for example. In addition, most simulators cannot share protocol or application implementations with real-world systems, and generally use simplified versions of the applications and protocols they do support. For example, protocols for the `ns` network simulator are written in a mix of C and OTcl, while protocols for the PARSEC and GloMoSim simulator are written in a language specific to that simulator [NS, ZRM98]. Each of these difficulties limit the usefulness of simulators for studying complex adaptive applications and protocols.

Emulators are one way of testing complex applications and protocols while avoiding some of the shortcomings of network simulators. In an network emulator, applications run on normal machines instead of in a simulator, but the network used by the machine is still simulated. This works reasonably well in low bandwidth, high latency networks where the emulator can keep up with the real-time constraints of emulation. Emulation does sacrifice some control over the state of the system compared to simulation, since a full-fledged operating system is used instead of the simplified one common in most simulators. This, however, allows the interactions between the operating system and complex applications to be studied in ways that are not normally possible in network simulators.

Despite these advantages, most emulators actually do little to make developing complex protocols easier, since they use simplified simulated protocols for communication instead of the actual protocols in the operating system networking stack. As in simulators, developing in this environment or on real systems may require writing a protocol multiple times. In addition, most emulators are written as centralized servers where the various hosts taking part in the emulation forward wireless packets to the server to have their behavior simulated [NS, DBCF95, KGM+01]. This can add several extra network hops to most network communication and dramatically alter the latency characteristics of the emulated network unless they are restricted to the emulation of high-latency networks. Other emulators require the use of simulation results to drive the emulator [NSNK97], reintroducing many of the limitations of network simulation.

The emulator described in this chapter, VWLAN, addresses these shortcoming in two ways. First, packet transmission and reception decisions are distributed to the machines in the cluster on which the emulator runs, and each machine is responsible for maintaining its own information about the state of the emulated shared radio medium. This allows each machine to make purely local decisions about if and when it should send or receive a packet, removes the need for a centralized emulation server, and lets our emulator preserve latency characteristics when emulating low-latency networks. Second, our emulator is implemented as a Linux network device driver, allowing all of the standard Linux network protocols to be used with it and new protocols built using the emulator to be used unchanged with other Linux network devices.

7.2 Emulator Environment

VWLAN emulates a collection of mobile devices employing the WaveLAN 2Mbps 900MHz direct-sequence spread-spectrum wireless LAN card using a cluster of Linux workstations. We emulate the 2Mbps WaveLAN as opposed to newer cards because the radio and MAC characteristics of this card are well-documented and the media access control (MAC) pro-

tocol for the WaveLAN is relatively simple. Specifically, the WaveLAN cards use the CSMA/CA MAC protocol, a variant of the standard CSMA/CD protocol used by ethernet cards. The newer IEEE 802.11 wireless cards, in contrast, use a comparatively complicated MAC protocol that includes per-packet acknowledgments, ready-to-send (RTS) packets, and clear-to-send (CTS) packets, as well as optional time-division multiplexing and power-saving modes. As will be described in section 7.3.4, accurately emulating even the relatively simple CSMA/CA MAC protocol is difficult.

The emulator runs on a cluster of 64 200 MHz Pentium Pro workstations connected using 100BaseTX switched ethernet. Because network emulation has fine-grain timing constraints, the emulator runs only on Linux kernels that include the Kansas University Real-Time (KURT) extensions [HSPN98]. These extensions give the emulator much greater precision in scheduling events in the kernel but do not affect the Linux network stack, which still lacks real-time processing guarantees. Because of this, we also increased the kernel clock rate to 1000 Hz, as opposed to the 100 Hz clock rate normally used in Linux kernels running on the x86 architecture. This increases system overhead slightly, but also allows the emulator to receive packets from the underlying ethernet device driver more quickly. In addition, machines in the cluster keep their clocks synchronized within approximately 50 microseconds using NTP, the Network Time Protocol [Mil92].

7.3 Emulator Design

7.3.1 Design Overview

As previously mentioned, the VWLAN emulator is distributed, with each machine in the emulator representing a wireless machine in a simulated world and having an associated virtual position in this world. Machines make transmission and reception decisions based solely on locally maintained information about the status of the radio channel around their virtual position. They maintain this information by receiving packets sent by machines nearby in the simulated network and using information about the radio signals those pack-

ets represent to update their model of the local radio state. Because the underlying network used to transport these packets is much faster than the network being emulated, each machine is able to maintain a reasonably accurate model of the state of the simulated radio channel.

Because radio is a broadcast medium, each machine must see all of the radio signal information contained in packets sent by machines that are geographically close in the simulated environment, even if those packets are not for that particular machine. VWLAN achieves this by dividing the simulated world into rectangular *regions* each of which has its own well-known associated ethernet multicast address. Each machine in a region sends its outgoing packets to the multicast address associated with that region, and receivers subscribe to the multicast addresses of all of the regions near their current simulated location. On machines with hardware ethernet multicast filtering (most modern ethernet devices), this can significantly reduce the load on some machines in large emulations compared to using, for example, ethernet broadcast.

Figure 7.1 shows the internal architecture of the emulator software that runs on each machine. VWLAN is, as already mentioned, implemented as a Linux device driver layered on top of an existing network card. To software above the device driver layer of the network stack, the emulator looks like any other ethernet device, although it does have a slightly smaller maximum transmission unit (MTU) than normal ethernet devices. To network device drivers, on the other hand, VWLAN looks like a network protocol that processes packets, allowing it to send and receive packets using these network devices.

A timeline of the emulator packet transmission and reception process is shown in figure 7.2. Once the transmitter has obtained media access at point A using a media access protocol, the transmitter gives the data to the ethernet driver (point B), which sends the packet. This packet is received by the destination machine's ethernet driver at point C and handed to the emulator on the receiver at point D. At point E, the signal represented by this packet would have finished with the radio medium and be received by a WaveLAN network card. Finally, point F represents the last point at which a packet representing a radio signal sent

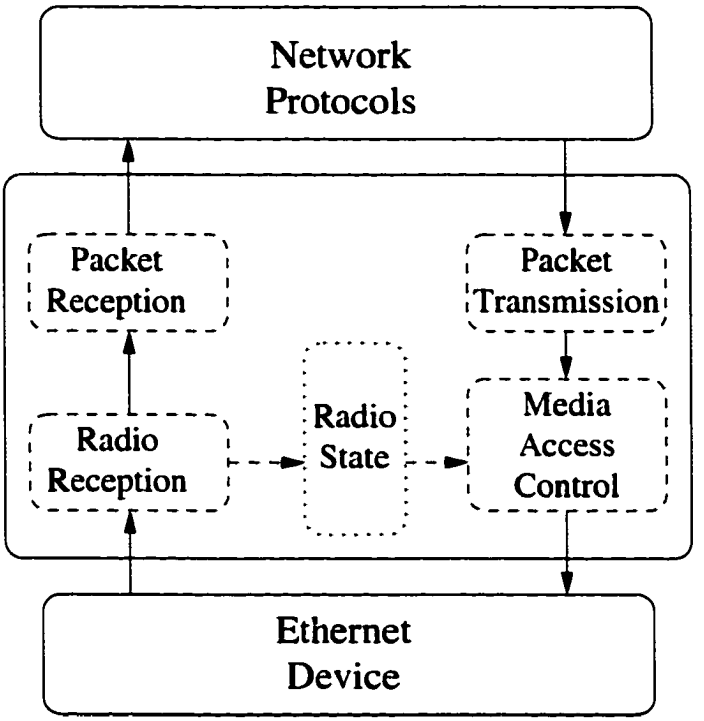


FIGURE 7.1. Emulator Software Architecture

between points B and E could be received by the destination machine and collide with the original signal sent at time B. The following subsections describe the transmission and reception process of the emulator in more detail and also describe potential sources of errors in the emulator and how the emulator attempts to compensate for them.

7.3.2 Packet Transmission

The transmission portion of the VWLAN driver shown in figure 7.1 has two components, a *packet transmission module* and a *media access control module*. The packet transmission module is responsible for preparing packets for transmission by prepending an additional header to the packet, shown in figure 7.3, that describes the radio signal associated with the packet. The `transmit_itime` field contains the time in microseconds since the start of the UNIX epoch at which the radio signal represented by the packet was transmitted. The location, power, and frequency fields describe the radio signal itself and are sufficient

```
struct emulator_header
{
    uint64_t transmit_time; /* Time the packet was actually
                             * sent */
    float sx, sy, sz;       /* x,y,z location of the
                             * transmitter*/
    float txPower;          /* transmission power*/
    float freq;             /* transmission frequency */
    unsigned char daddr[6]; /* Actual destination address
                             * of packet */
}
```

FIGURE 7.3. Additional Packet Header Information used by Emulator

the transmitter hands the packet to the underlying ethernet device for transmission, waits until the packet would be done sending on the emulated network, and then performs a final wait during which other machines can start transmission before it can again attempt to acquire the radio medium for transmission.

7.3.3 Packet Reception

The receive side of the VWLAN driver shown in figure 7.1 is responsible for updating the data structures that track the state of the simulated radio channel after each packet is received and determining whether packets destined for the local machine can be heard on the radio channel. Receivers determine which packets they should actually receive by computing the signal strength of every packet received using the information in figure 7.3 and comparing this power to the total amount of radio noise from other radio transmissions.

When a packet is received, the signal information included in the packet is entered into a time-sorted list of recent radio signals. It takes significantly longer for a 2Mbps WaveLAN packet to traverse the radio media than for it to be transmitted on a 100Mbps ethernet, so the received packet cannot yet be passed up the network stack as other packets could arrive that describe signals that collide with the current packet. In particular, although the packet

is received by the emulator at time D in figure 7.2, the radio signal described by that packet would not be done being received until time E because of the longer transmission time of the 2Mbps radio medium. In addition, another packet may arrive as late as time F and still logically collide with the current packet. Hence, the receiver waits until time F and then checks if any other radio signal arrived that collided with the packet being received. If no collision occurred, VWLAN passes the packet up the network stack. If a collision has occurred, on the other hand, both the packet being received and the packet with which it collided are dropped.

7.3.4 Sources of Error in the Emulator

The main source of error in the emulator is the delay in propagation of information from the ethernet card to the emulator driver. WaveLAN cards perform carrier senses in firmware or hardware, for example, and have almost instantaneous access to the current state of the radio medium. The emulator, on the other hand, has to wait several milliseconds from when a packet is sent (point B in figure 7.2) to when it is available in the emulator (point D in figure 7.2). This causes the emulator to occasionally make mistakes in carrying out the WaveLAN media access protocol. When such errors are unavoidable, the emulator favors receiving packets that should not have been received over dropping packets that should not have been dropped, as excess dropped packets can have a dramatic impact on, for example, TCP's congestion control mechanisms.

Carrier sense-based multiple access is particularly problematic for VWLAN. Because of the delay in receipt of radio signal information described above, the emulator sometimes has an incomplete view of the true state of the radio medium. This causes it to occasionally believe the radio medium is quiet when a radio signal has already been transmitted by another machine that would be audible to actual WaveLAN hardware. This mistake can cause the emulator to transmit in violation of the CSMA/CA MAC protocol, which in turn

can cause collisions at the receiver and excess dropped packets. Such behavior is exactly what the emulator seeks to avoid.

VWLAN attempts to work around this problem using *spurious collision detection* and *probabilistic carrier sense*. When using spurious collision detection, receivers detect when a packet was sent in violation of the media access control protocol and ignore collisions that would not have happened if both senders had correctly followed the MAC protocol. The receiver considers a potential collision spurious if it would not have occurred if both senders had obeyed the MAC protocol. Specifically, the receiver only considers two transmissions to have collided if they were sent at times or locations where neither sender would have been able to hear the other prior to sending. Any other potential collisions would not have happened if both senders had correctly followed the MAC protocol and are therefore spurious collisions that should be ignored. This, unfortunately, may allow the simulated media to be utilized at a significantly higher rate than is physically possible.

The emulator uses *probabilistic carrier sense* in its media access control module to try and correct for any media overutilization resulting from spurious collision detection. The essential insight is that media overutilization happens because looking only recently received packets underestimates the likelihood of sensing a carrier. Probabilistic carrier sense attempts to correct this error by artificially increasing the likelihood of sensing a carrier as follows. If, during media acquisition, the check for a real carrier in VWLAN's stored radio state does not detect a carrier, the emulator performs an additional check for a *probabilistic carrier*, a fake carrier that is not actually present. A fake carrier is randomly detected with a probability equal to the emulator's estimate of the chance of failing to detect a carrier when one was actually present. The emulator estimates this probability by periodically sampling the media during the normal carrier sense and then later looking at whether this sample was correct.

If a probabilistic carrier is detected, the transmitter assumes a carrier is actually present and waits the amount of time it takes to transmit an average length packet before attempting to continue media acquistion. While it might seem that the emulator should wait for half

of an average packet length on the theory that a random carrier sense will on average start halfway through a packet transmission, this is not generally true. Probabilistic carriers account for the case where a packet is sent in violation of the MAC protocol, that is, where they are sent very close to the start of another packet due to an incorrect carrier sense. Based on this, waits for the end of a probabilistic carrier last for the entire length of an average packet.

7.4 Evaluation

To determine the effectiveness of the mechanisms described in section 7.3.4, we performed tests that compared emulator configurations with and without spurious collision detection and probabilistic carrier sense to simulation results from the `ns` network simulator. In particular, we tested the following emulator configurations:

- The `vwlan-standard` emulator configuration includes neither spurious collision detection nor probabilistic carrier sense mechanisms.
- The `vwlan-spurious` emulator configuration includes the spurious collision detection mechanism but not the probabilistic carrier sense mechanism.
- The `vwlan-prob` configuration includes both spurious collision detection and probabilistic carrier sense.

Each emulator test consisted of placing a fixed number of machines in the same simulated region and instructing each machine to send data at a constant rate of 1Mbps to another machine. Up to ten machines can be used to offer load to the network, resulting in between 2Mbps and 10Mbps available load for the 2Mbps network. We then measured the average effective throughput rate actually provided by the network. Tests were run on the cluster described in section 7.2, with the average bandwidth measured over the course of a 10 minute run.

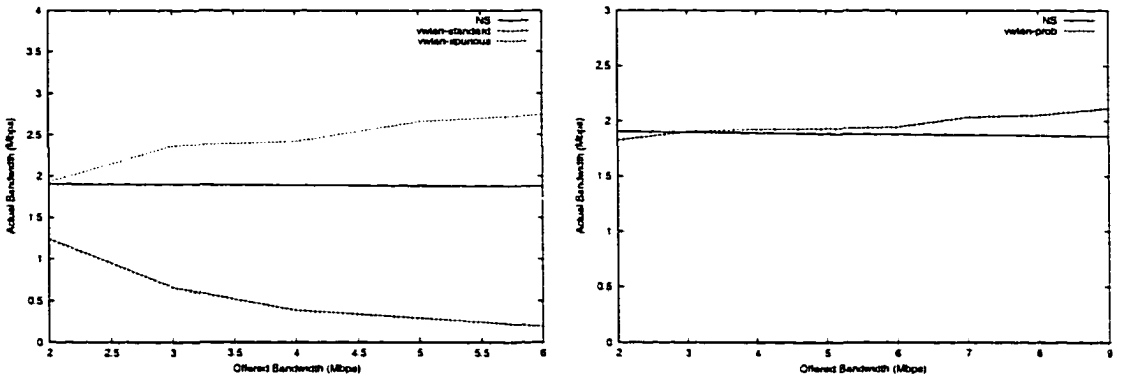


FIGURE 7.4. Throughput of Network in Different Emulator Configurations

Figure 7.4 shows the effective bandwidth versus offered load for each of the emulator configurations and the baseline `ns` simulator results. As shown in the graph on the left, the effective throughput in the `vwlan-standard` configuration drops much more quickly than it should due to spurious collisions caused by timing errors, while the `vwlan-spurious` configuration exhibits the opposite problem, overutilizing the network as machines are allowed to incorrectly send packets at the same time. The performance of the `vwlan-prob` configuration, shown in the graph on the right, performs almost identically to the `ns` simulations when the offered load is between 2Mbps and 5Mbps, but slowly starts to climb above the 2Mbps theoretical limit of the WaveLAN network when actual thoughput should be decreasing. This illustrates that our approach is reasonably accurate under relatively light load, but becomes increasingly less accurate as the media access control mechanisms are heavily stressed.

Because the difficulties in this approach appear to be related to the use of carrier sense-based MAC protocols, the distributed approach described above could be useful in low-latency networks that use time-division multiple access (TDMA) methods, token-passing techniques, or polling-based MAC protocols. Bluetooth, a recent wireless personal area network (PAN) technology, is one example of a network that uses polling-based MAC protocols [HM00]. Because these MAC protocols generally allow only one transmitter to

send at a time, the timing-based limitations of VWLAN's approach should be much less detrimental when emulating these networks.

Alternative emulator organizations could also solve the problem of emulating low-latency networks, particularly ones that use CSMA-based MAC protocols. One potential approach would be to immediately send every packet queued for transmission to every machine in the emulator, and then have every machine determine independently when that packet would actually be transmitted. This approach, however, has several problems. In particular, it would place a relatively large computational burden on every machine in the network compared to the approach described in this chapter, as every machine in the emulator would have to process every packet queued for transmission by any machine. In addition, most CSMA-based MAC protocols include random exponential backoffs that make them non-deterministic, while the approach described needs packet transmission decisions to be deterministic. This problem could potentially be avoided in a variety of ways, such as including a seed for a pseudo-random number generator in every packet that should be used for random decisions related to that packet, or by using distributed agreement protocols to decide when each packet was sent.

7.5 Summary

The emulator presented in this chapter provides the first step toward the accurate emulation of low-latency high-bandwidth wireless networks. Unlike approaches that rely on a centralized server and can only support high-latency networks [NS, DBCF95, KGM+01] or approaches that require simulation results to drive emulation [NSNK97], our approach allows for limited emulation of low-latency networks without developing protocols and applications separately on network simulators. The limitations of the VWLAN emulator currently restrict its usefulness to applications that lightly load the emulated network or do not require accurate emulation of network bandwidth when the network is heavily loaded, but the general approach could also be used to emulate wireless networks such as Blue-

tooth that use media access protocols with less demanding timing requirements than the WaveLAN CSMA/CA media access protocol.

CHAPTER 8
CONCLUSIONS

8.1 Summary

This dissertation has presented Cholla, a framework for composing and coordinating adaptations in system software. Adaptation and configuration have become important design techniques because of the proliferation of diverse hardware and applications environments such as small mobile devices, wireless networks, and multimedia applications. Without the ability to coordinate multiple adaptations in different components and on different machines, interactions between these adaptations may cause a system to adapt incorrectly or inconsistently. In addition, without the ability to customize the policies that control adaptation, adaptations will seldom match the exact needs of the hardware, applications, and users that rely on them.

In chapter 2, we described prior work on adaptation, configuration, and coordination. Specifically, we reviewed a number of recent and historical adaptable systems and configurable systems, as well as a handful of systems that support both adaptation and configuration. None of these approaches, however, address how to coordinate adaptation among multiple components in highly configurable systems. Only one of the systems described, the SWiFT toolkit, seeks to provide general support for constructing adaptation policies, although even it does not address coordination or decomposition of existing complex control policies [Cen97].

Chapter 3 described Cholla, a framework for coordinating adaptation in highly-cofigurable software. The Cholla architecture is comprised of adaptation controllers and a runtime system; these elements are together responsible for controlling and coordinating adaptation in the adaptable components in a system. Adaptation controllers make policy decisions for adaptable components on a given host, while the runtime system provides event notifi-

cation, controller invocation, and adaptation synchronization. By separating adaptation policies from the mechanisms they control, Cholla allows these policies to be analyzed, customized, and composed in ways that would difficult at best in other systems. Cholla controllers use fuzzy control rules to build adaptation controllers. This allows controllers to be created by composing modular rule sets, each of which describes a portion of an adaptation policy, and also allows controllers to make decisions based on both quantitative and qualitative information.

Chapter 4 followed with a description of a Cholla prototype designed to test the adaptation composition and inter-component coordination aspects of the architecture; inter-host coordination issues are not addressed in this prototype. The prototype is implemented in Cactus, a framework for constructing highly-configurable network protocols. In this prototype, Cholla is responsible for controlling and coordinating adaptation in network protocol stacks. These stacks contain composite protocols that are constructed by composing microprotocols, which constitute the adaptable components that Cholla is responsible for controlling.

Chapter 5 then demonstrated how this prototype could be used to control adaptation in a configurable transport protocol, CTP. Specifically, it showed how an existing highly-configurable network protocol could be connected to Cholla using an adaptation wrapper, and how standard transport protocol adaptation policies could be decomposed into fuzzy rule sets for use by adaptation controllers. Chapter 5 then demonstrated how changing the rule sets used by these controllers allowed fine-grained customization of the adaptive behavior of CTP. In addition, it also described several improvements to CTP that make it more suitable for use by adaptable and configurable applications.

Two applications designed around the Cholla prototype and the enhanced version of CTP were then described in chapter 6. A multimedia transmission application was used to demonstrate how Cholla can control and coordinate adaptation in multiple adaptable components. The coordination provided by Cholla was shown to be particularly useful when adaptation choices are limited and adaptation mechanisms are very sensitive, as is the case

in the audio adaptation mechanisms implemented in the multimedia transmission application. Coordination was also shown to be less useful for controlling video transmission rates, where the adaptation mechanisms were more flexible and simple linear control techniques could be used. The design for an adaptable, configurable proxy for wireless networks was also presented, along with a discussion of the kinds of control and coordination necessary in this application.

Finally, chapter 7 described the design of VWLAN, a distributed network emulator for the WaveLAN wireless card that avoids many of the limitations of other emulation approaches. This emulator has limitations, however, that prevent it from accurately reproducing the bandwidth behavior of a heavily loaded WaveLAN network. When the network is lightly loaded, however, VWLAN accurately emulates the low latency WaveLAN network on stock hardware without the use of a separate simulation process.

8.2 Future Work

The work described in this dissertation can be expanded in a number of different directions. This could include, for example, additional work on controlling other applications with Cholla, experimentation with coordinating distributed adaptation, or the exploration of automated techniques for creating coordination rules. The remainder of this section describes each of these possible directions in more detail.

As described in chapter 6, further work could be done on controlling multiple video parameters, especially H.263's quantization and i-frame rate parameters. Because framerate control is more effective than quantization or i-frame rate control in adjusting the length of the segment queue, a natural extension of our research would be to use framerate control for that purpose, and to use quantization and i-frame rate control to manage other less volatile parameters. Quantization could be adjusted, for example, to control CPU utilization, while the i-frame rate could be changed in response to information on the lossiness of the network. Because these parameters would be primarily controlling less volatile elements of

the application, very low gain control could be used, limiting the effect of these parameters on the segment rate. Changes in these parameters do, however, affect the segment rate, so coordination rules would be needed to account for side effects of quantization and i-frame rate control. In addition to work on the multimedia transmission application, implementing and experimenting with the Cholla wireless proxy described in section 6.2 would provide valuable experience with the types of coordination necessary in more complex systems.

The two applications described in chapter 6 could also be combined into a single large system that adaptively transmits and recodes multimedia data. Such a system could be used for studying inter-host coordination, for example. Other applications that could be used for studying inter-host coordination include distributed high-performance computing systems and distributed file systems. Others researchers have already examined composition and adaptation in file systems [GHM+90, KN93, REM+96, BOK+99, ZB99]; such work could serve as the foundation for research exploring inter-component and inter-host coordination in this area.

Finally, research into automated methods for creating and coordinating control policies would be very useful. As adaptable systems become larger and more complex, hand-written policies for control and coordination become more difficult to write, understand, and test. Because Cholla separates control policies from the mechanisms they control, reasoning about and modifying these control policies is generally easier than in monolithic systems that mix adaptation policy and mechanism. Machine learning techniques could be particularly useful for generating coordination rules or tuning the interactions between existing rule sets. A large body of work exists on machine learning, including work on tuning and learning in fuzzy systems [Wan97, COZ98]. Because Cholla controllers are built using fuzzy control rules, such work could readily be applied to Cholla. JFS, the fuzzy system used by the Cholla prototype described in chapter 4, in fact implements two different machine learning algorithms that could be applied to the tuning of Cholla controllers [Mor00].

REFERENCES

[Aal] R. Aalmoes. Roalt's H.263 page. http://www.xs4all.nl/ roalt/h263.html.

[ACO97] M. Allman, A. Caldwell, and S. Ostermann. ONE: The Ohio Network Emulator. Technical Report TR-19972, School of Electrical Engineering and Computer Science, Ohio University, August 1997.

[BHSC98] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, Nov 1998.

[BNBH98a] S. Brandt, G. Nutt, T. Berk, and M. Humprey. A dynamic quality of service middleware agent for mediating application resource usage. In *Proceedings of the 9th IEEE Systems Symposium (RTSS98)*, Madrid, Spain, Dec 1998.

[BNBH98b] S. Brandt, G. Nutt, T. Berk, and M. Humprey. Soft real-time application execution with dynamic quality of service assurance. In *Proceedings of the 1998 6th International Workshop on Quality of Service*, pages 154–163, Napa, CA, May 1998.

[BOK+99] A. Brown, D. Oppenheimer, K. Keeton, R. Thomas, J. Kubiatowicz, and D. Patterson. ISTORE: Introspective storage for data-intensive network services. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 32–37, Rio Rico, AZ, Mar 1999.

[BOP94] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proceedings of ACM SIGCOMM '94*, Sep 1994.

[BPSK97] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, Dec 1997.

[CCG+93] A. Campbell, G. Coulson, F. Garcia, D. Hutchison, and H. Leopold. Integrated quality of service for multimedia communications. In *Proceedings of IEEE INFOCOM '93*, pages 732–739, 1993.

[Cen97] S. Cen. *A software feedback toolkit and its applications to adaptive multimedia systems.* PhD thesis, Oregon Graduate Institute of Science and Technology, October 1997.

[CHS98] I. Chang, M. Hiltunen, and R. Schlichting. Affordable fault tolerance through adaptation. In J. Rolin, editor, *Parallel and Distributed Processing, Lecture Notes in Computer Science 1388*, pages 585–603. Springer, Apr 1998.

[CHS01] W.-K. Chen, M. Hiltunen, and R. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 635–643, Mesa, AZ, Apr 2001.

[CK01] F. Chang and V. Karamcheti. A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, 4(1):49–62, 2001.

[CM96] P. Creek and D. Moccia. *Digital Media Programming Guide*, chapter 7. Silicon Graphics, Inc., 1996. Document Number 007-1799-060.

[CMU98] The CMU Monarch Project. The CMU Monarch project's wireless and mobility extensions to ns. Available from http://www.monarch.cs.cmu.edu/, August 1998.

[CNW01] M. Corner, B. Noble, and K. Wasserman. Fugue: Time scales of adaptation of mobile video. In *Proceedings of Multimedia Computing and Networking 2001*, San Jose, CA, January 2001.

[COZ98] E. Cox, M. O'Hagan, and L. Zadeh. *The fuzzy systems handook.* Academic Press, Inc., 1998.

[CPW98] S. Cen, C. Pu, and J. Walpole. Flow and congestion control for Internet streaming applications. In *Proceedings of Multimedia Computing and Networking 1998*, 1998.

[CRS+98] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. Sanders, D. Bakken, M. Berman, D. Karr, and R. Schantz. AQuA: An adaptive architecture that provides dependable distributed objects. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, IN, Oct 1998.

[DBCF95] N. Davies, G.S. Blair, K. Cheverst, and A. Friday. A network emulator to support the development of adaptive applications. In *Proc. USENIX Symposium on Mobile and Location Independent Computing*, Ann Arbor, Michigan, April 1995.

[DFBC96] N. Davies, A. Friday, G. S. Blair, and K. Cheverst. Distributed systems support for adaptive mobile applications. *Mobile Networks and Applications*, 1:399–408, 1996.

[DHLB98] D. Dwyer, S. Ha, J.-R. Li, and V. Bharghavan. An adaptive transport protocol for multimedia communication. In *International Conference on Multimedia Computing and Systems*, pages 23–32, 1998.

[Dim87] N. Dimopoulos. Throughput and packet delay analysis for the H-network: CSMA/CD with adaptive and nonadaptive backoff protocols. *IEEE Transactions on Communication*, COM-35(11):1146–1152, 1987.

[dLWZ01] E. de Lara, D. Wallach, and W. Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, March 2001.

[DTB97] K. Dutton, S. Thompson, and B. Barraclough. *The Art of Control Engineering*. Addison-Wesley, 1997.

[FGBA96] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to network and client variation via on-demand, dynamic distillation. In *Proceedings of the 7th ASPLOS Conference*, Oct 1996.

[FGCB98] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications Magazine*, Aug 1998.

[FHPW00] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications. In *Proceedings of ACM SIGCOMM 2000 Symposium*, August 2000.

[FLA] FLAC — The Free Lossless Audio Codec. http://flac.sourceforge.net.

[FMS+98] D. Feldmeier, A. McAuley, J. Smith, D. Bakin, W. Marcus, and T. Raleigh. Protocol boosters. *IEEE Journal on Selected Areas in Comm.*, SAC-16(3):437–444, Apr 1998.

[GHM+90] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–71, Anaheim, CA, June 1990.

[GSPW98] A. Goel, D. Steere, C. Pu, and J. Walpole. SWiFT: A feedback control and dynamic reconfiguration toolkit. Technical Report CSE-98-009, Oregon Graduate Institute, 30, 1998.

[Hay98] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, Jan 1998.

[HM00] J. Haartsen and S. Mattisson. Bluetooth - a new low-power radio interface providing short-range connectivity. *Proceedings of the IEEE*, 88(10):1651–1661, Oct 2000.

[HMPT89] N. Hutchinson, S. Mishra, L. Peterson, and V. Thomas. Tools for implementing network protocols. *Software Practice & Experience*, 19(9):895–916, Sep 1989.

[HP91] N. Hutchinson and L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HPOA89] N. Hutchinson, L. Peterson, S. O'Malley, and M. Abbott. RPC in the *x*-kernel: Evaluating new design techniques. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, Dec 1989.

[HS00] M. Hiltunen and R. Schlichting. The Cactus approach to building configurable middleware services. In *Proceedings of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, Numberg, Germany, Oct 2000.

[HSNL97] D. Hull, A. Shankar, K. Nahrstedt, and J. Lui. An end-to-end qos model and management architecture. In *Proceedings of the IEEE Workshop on Middleware for Distributed Real-time Systems and Services*, December 1997.

[HSPN98] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus. Temporal resolution and real-time extensions to Linux. Technical report, Information and Telecommunication Technology Center, University of Kansas, 1998.

[HWS01] M. Hiltunen, G. Wong, and R. Schlichting. Dynamic messages: An abstraction for complex communication protocols. *Software: Practice and Experience*, 2001. Submitted for publication.

[ITU] International Telecommunication Union. Video coding for low bit rate communication. Recommendation H.263.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Proceedings of ACM SIGCOMM '88*, pages 314–332, Aug 1988.

[Kat94] R. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications Magazine*, 1(1):6–17, 1994.

[Kes91] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM '91*, pages 3–15, September 1991.

[KGM+01] M. Kojo, A. Gurtov, J. Manner, P. Sarolahti, T. Alanko, and K. Raatikainen. Seawind: a wireless network emulator. In *Proceedings of 11th GI/ITG Conference on Measuring, Modelling and Evaluation of Computer and Communication Systems*, Aachen, Germany, September 2001.

[KHP99] P. Keleher, J. K. Hollingsworth, and D. Perkovic. Exploiting application alternatives. In *The 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[KN93] Y. Khalidi and M. Nelson. Extensible file systems in Spring. In *Proceedings of the 14th Symposium on Operating Systems Principles*, Asheville, NC, Dec 1993.

[KRL+01] D.A. Karr, C. Rodrugues, J.P. Loyall, R.E. Schantz, Y. Krishnamurthy, I Pyarali, and D.C. Schmidt. Application of the QuO quality of service framework to a distributed video application. In *Proceedings of the International Symposium on Distributed Objects and Applications*, Rome, Italy, September 2001.

[KT75] L. Kleinrock and F. A. Tobagi. Packet switching in radio channels: Part 1 – carrier sense multiple access modes and their throughput-delay characteristics. *IEEE Transactions in Communications*, 23(12):1400–1416, 1975.

[LN99] B. Li and K. Nahrstedt. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications*, 17(9), Sept 1999.

[Mas92] H. Massalin. *Synthesis: An Efficient Implementation of Fundamental Operating System Services*. PhD thesis, Columbia University, 1992.

[Mil92] D. Mills. Network time protocol (version 3) specification, implementation, and analysis. Request for Comments RFC 1305, University of Delaware, March 1992.

[MJV96] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *Proceedings of ACM SIGCOMM '96*, pages 117–130, Stanford, CA, August 1996.

[MMMV01] S. Michiels, T. Mahieu, F. Matthijs, and P. Verbaeten. Dynamic protocol stack composition: Protocol independent addressing. In *Fourth ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2001)*, June 2001.

[MMO+94] A. Montz, D. Mosberger, S. O'Malley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operating system. In *Proceedings of the 1st USENIX Symposium on Operating System Design and Implementation (OSDI)*, Nov 1994.

[MMWV01] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. Position summary: Dips: A unifying approach for developing system software. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, May 2001.

[Mor00] J. Mortensen. JFS development environment. http://inet.uni2.dk/~jemor/-jfs.htm, Dec 2000.

[Mos97] D. Mosberger. Message library design notes. Technical Report 97-19, Department of Computer Science, University of Arizona, Tucson, AZ, Nov 1997.

[MP96] D. Mosberger and L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 153–168, Oct 1996.

[MP99] M. Margaritidis and G.C. Polyzos. Application-assisted adaptation of real-time streams over wireless links. In *Proceedings of the Second Annual UCSD Conference on Wireless Communications*, San Diego, California, March 1999.

[MST80] N. Meisner, J. Segal, and M. Tanigawa. An adaptive retransmission technique for use in a slotted-ALOHA channel. *IEEE Transactions on Communication*, COM-28:1176–1178, 1980.

[NIS] NIST Internetworking Technology Group. Nistnet network emulation package. http://www.antd.nist.gov/itg/nistnet/.

[NS] The LBNL network simulator, ns. http://wwwnrg.ee.lbl.gov/ns.

[NSN+97] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 276–287, Saint Malo, France, October 1997.

[NSNK97] B. Noble, M. Satyanarayanan, G. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proceedings of ACM SIGCOMM'97*, Cannes, France, September 1997.

[OGG] Ogg vorbis web pages. www.xiph.org/ogg/vorbis/.

[OMG98] Object Management Group. *The Common Object Request Broker: Architecture and Specification (Revision 2.2)*, Feb 1998.

[PC00] G. Parr and K. Curran. A paradigm shift in the distribution of multimedia. *Communications of the ACM*, 43(6):103–109, Jun 2000.

[PHOR90] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao. The *x*-Kernel: A platform for accessing internet resources. *IEEE Computer*, 23(5):23–33, May 1990.

[PMI88] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[Pos81] J. Postel. Transmission control protocol. RFC 793, Sep 1981.

[PYT] The Python Language Website. http://www.python.org.

[Rap96] T. S. Rappaport. *Wireless Communications, Principles and Practice*. Prentice Hall, 1996.

[RBF+95] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proceedings of the 14th ACM Principles of Distributed Computing Conference*, pages 80–89, Aug 1995.

[RBH+98] R. van Renesse, K. Birman, M. Hayden, A. Vaysburd, and D. Karr. Building adaptive systems using Ensemble. *Software Practice and Experience*, 28(9):963–979, Jul 1998.

[REM+96] D. Reed, C. Elford, T. Madhyastha, E. Smirni, and S. Lamm. The next frontier: Closed loop and interactive performance steering. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, pages 20–31, Bloomingdale, IL, August, 12 1996.

[RHE99] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *INFOCOM (3)*, pages 1337–1345, 1999.

[Rit84] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):311–324, Oct 1984.

[Riz97a] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *ACM Computer Communications Review*, 27(2):31–41, January 1997.

[Riz97b] L. Rizzo. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review*, 27(2):24–36, April 1997.

[SBK95] J. Short, R. Bagrodia, and L. Kleinrock. Mobile wireless network system simulation. *Wireless Network*, pages 451–467, 1995.

[SBS93] D. Schmidt, D. Box, and T. Suda. ADAPTIVE: A dynamically assembled protocol transformation, integration, and evaluation environment. *Concurrency: Practice and Experience*, 5(4):269–286, Jun 1993.

[SCFJ96] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A transport protocol for real-time applications. RFC 1889, Jan 1996.

[SESS96] M. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Operating Systems Design and Implementation*, pages 213–227, 1996.

[SLMP02] R. E. Schantz, J. P. Loyall, M. Atighetchi M, and P. P. Pal. Packaging quality of service control behaviors for reuse. In *Proceedings of ISORC 2002, the 5th IEEE International Symposium on Object-Oriented Real-time Distributed Computing*, Washington, DC, April 2002.

[SP02] J. Steinberg and J. Pasquale. A web middleware architecture for dynamic customization of content for wireless clients. In *Proceedings of WWW 2002*, May 2002.

[SS97] M. Seltzer and C. Small. Self-monitoring and self-adapting systems. In *Proceedings of the Sixth Workshop on Hot Topics on Operating Systems (HotOS-VI)*, Chatham, MA, May 1997.

[SVSB99] P. Sinha, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. WTCP: A reliable transport protocol for wireless wide-area networks. In *Proceedings of Mobicom'99*, Seattle, WA, Aug 1999.

[V4L] Video4Linux Resources. http://www.exploits.org/v4l/.

[vDLS00] H. van Dijk, K. Langendoen, and H. Sips. ARC: A bottom-up approach to negotiated QoS. In *IEEE Workshop on Mobile Multimedia Systems and Applications*, 2000, December 2000.

[Wan97] L. Wang. *A Course in Fuzzy Systems and Control.* PTR Prentice Hall, 1997.

[WCB01] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, Chateau Lake Louise, Canada, October 2001.

[WHS01] G. Wong, M. Hiltunen, and R. Schlichting. A configurable and extensible transport protocol. In *Proceedings of IEEE INFOCOM '01*, pages 319–328, Anchorage, Alaska, Apr 2001.

[WLAG93] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

[YL00] Y. R. Yang and S. S. Lam. General AIMD congestion control. In *Proceedings of ICNP 2000*, November 2000.

[Zad72] L. Zadeh. A rationale for fuzzy control. *Journal of Dynamic Systems, Measurement and Control*, 94(Series G):3–4, March 1972.

[ZB99] E. Zadok and I. Badulescu. A stackable file system interface for Linux. In *Proceedings of the 5th Annual Linux Expo*, pages 141–151, Raleigh, North Carolina, May 1999.

[ZBS96] J. Zinky, D. Bakken, and R. Schantz. Quality of service for objects. http://www.bbn.com/products/dpom/dualuse.htm, 1996.

[ZRM98] X. Zeng, Bagrodia R., and Gerla M. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of PADS'98*, Banff, Canada, May 1998.