A Software Feedback Toolkit and its Application in Adaptive Multimedia Systems

Shanwei Cen
B.S. in Computer Science, Tsinghua University, Beijing, China, 1987
M.S. in Computer Science, Tsinghua University, Beijing, China, 1989

A dissertation submitted to the faculty of the
Oregon Graduate Institute of Science and Technology
in partial fulfillment of the
requirements for the degree
Doctor of Philosophy
in
Computer Science and Engineering

October 1997

The dissertation "A Software Feedback Toolkit and its Application in Adaptive Multimedia Systems" by Shanwei Cen has been examined and approved by the following Examination Committee:

Calton Pu
Professor
Thesis Research Adviser

Jonathan Walpole
Associate Professor
Thesis Research Adviser

David Maier
Professor
Oregon Graduate Institute

Mayer Schwartz
Principle Engineer
Tektronix, Inc.

Tom D. C. Little
Associate Professor
Boston University

Dedication

To my wife Jinrong and my son Stanley.

Acknowledgements

I am most grateful to my advisors Calton Pu and Jonathan Walpole. They have been extremely helpful in identifying the thesis topic, developing the ideas, implementing the prototype systems, publishing the papers, and writing the thesis. Their appreciation of my work and encouragement were important in completing my thesis. I like the close relationship and frequent and inspiring discussions with Calton and Jonathan. Working with them has been a great pleasure to me.

I would like to thank the members of my thesis committee, David Maier, Mayer Schwartz and Tom Little, for their discussions on the ideas in the thesis, and careful reading and insightful comments on my thesis. Special thanks to Mayer, who was also my mentor during my internship at Tektronix, Inc. His encouragement enabled me to do my thesis at full speed while still gaining industry experience by working part-time.

The Distributed Systems Research Group (DSRG) has been the best group I have known of. The members of the group, Jonathan Walpole, Calton Pu, David Maier, Crispin Cowan, Dylan McNamee, David Steere, Charlie Krasic, Walt Leyland, Ryan Day, Jon Inouye, Roger Barga, Rainer Koster, Veronica Baiceanu, Dan Revel, Liujin Yu, Qian Zhang, Tong Zhou, Ashvin Goel, and others, all have been very helpful to me. I couldn't help appreciating the cooperative, open, aggressive and productive atmosphere in the group. Special thanks to Jon Inouye, for his unselfish help on all the tools and systems, and his interesting discussions on many topics.

I would also like to thank Jim Hook, a member of my Student Program Committee (SPC), for his advice on my program, encouragement on improving my skill in English, and his appreciation and comments on my thesis work.

The staff of the Data Intensive Systems Center (DISC), Jo Ann Binkerd and Jeff West, have been very helpful in making the workplace a comfortable, convenient and productive

one. The systems staff, including Marion Hakanson, Mark Morrissey, Liesl Andrico, and others, owned the credits for keeping the computers in the department running seamlessly all the time.

I am also grateful to Professor Karsten Schwan of Georgia Institute of Technology, for providing an account at Georgia Tech, which has been used in the Internet experiments for the thesis.

It is a pleasure to have good friends like Paul Benninghoff, Brian Mak, Sanjay Nadimpal, Ke Zhang, Javed Anwar and Radhika Reddy. They brought a lot of fun to my student life at OGI.

This thesis work is supported in part by grants from DARPA under grant N00014-94-1-0845 and contract MDA904-95-C-5547, and by Tektronix, Inc. and Intel Corporation.

Finally, special thanks to my wife Jinrong and my son Stanley for their love and pressure. They are the only reason why I have the strong motivation and energy to finish my thesis while having a half-time job at the same time.

Contents

List of Tables

List of Figures

Abstract

A Software Feedback Toolkit and its Application in Adaptive Multimedia Systems

Shanwei Cen

Supervising Professors: Calton Pu and Jonathan Walpole

As modern computer and network technologies develop, computer systems, especially distributed multimedia systems across the Internet, become more and more complex and dynamic. Among the problems of complex and dynamic computer systems are heterogeneity and a high degree of dynamics and unpredictability in their environment, the difficulty in precise modeling, and potential lack of convergence to desirable stable states. These problems call for mechanisms that can control complex and dynamic computer systems effectively without relying on insight into their internal structure nor precise models of their behavior. Furthermore, such mechanisms should be able to adapt across a wide range of possible changes in the heterogeneous software environment.

Software feedback, a software technique that uses feedback mechanisms similar to those in hardware feedback systems such as phase-lock loops, plays an important role in making complex and dynamic computer systems adaptive. It already exists in many forms, such as network flow and congestion control, clock synchronization between Internet hosts, intra- and inter-stream synchronization in distributed multimedia streaming systems, and adaptation in multimedia presentation quality. However, the existing feedback mechanisms are implemented in custom ways. They suffer from arbitrary structure and wasted effort due to repeated redesign and re-implementation of logically similar components.

To address the issue of building software feedback mechanisms systematically and efficiently for complex and dynamic computer systems, in this thesis, we present a software feedback toolkit and discuss its application in adaptive multimedia systems. We propose a methodology for hierarchical composition of feedback systems on top of simple building blocks. We also introduce the concepts of guard-based meta-adaptation, guarded feedback components, and dynamic component replugging for composing feedback mechanisms that adapt across a wide range of system dynamics based on simple feedback policies with limited domains. We implement the toolkit in C++, with a library of building blocks and a set of tools for simulation and instrumentation. This software feedback toolkit facilitates the development of highly modular, adaptive and extensible feedback systems, and helps the reuse of existing feedback components. Then we demonstrate the application of the software feedback toolkit for building adaptive real-time packet rate control mechanisms, network flow and congestion control mechanisms for multimedia streaming, and an adaptive real-time distributed video player.

Chapter 1

Introduction

1.1 Motivations

As modern computer and network technologies develop, computer systems and applications become more and more complex and dynamic. The Internet is a typical example of the state of the art. It is a huge network, composed of millions of heterogeneous computers connected through a wide variety of network links, and numerous kinds of applications. Furthermore, hosts, network links and applications are added or removed constantly. Each application can be envisioned as a dynamic system living in an ever-changing environment. Problems of complex and dynamic systems include difficulty of modeling them precisely, and potential lack of convergence to desirable stable states. There are many reasons why a dynamic system may not be able to stabilize on an expected state by itself. Either there is disturbance from the environment, or the limit of its structure makes it impossible to converge automatically. These problems of dynamic systems call for mechanisms that can control them effectively without relying on detailed insight into their internal structure nor precise models of their behavior. Furthermore, such mechanisms should adapt quickly to changes in the environment.

Software feedback is a software technique that uses feedback mechanisms similar to those in hardware feedback systems, such as phase-lock loops [3, 5]. A feedback mechanism continuously monitors the output of the system under control (the target system), compares the result against preset values (goals of the feedback control), and feeds the difference back to adjust the behavior of the target system. One beneficial property of feedback mechanisms is that they can control complex dynamic systems effectively even

when they have no, or only partial, knowledge of the target system's internal structure or when they do not have a precise model of it. This property suggests that feedback mechanisms could be used in controlling complex dynamic systems that must operate in highly complex and unpredictable environment, hence making the dynamic systems adaptive to the diverse and ever-changing conditions.

Software feedback mechanisms already exist in many forms in computer systems. They have been used in inter-host clock synchronization, network flow and congestion control, quality-of-service control in multimedia presentations, process and thread scheduling, system adaptation, and many other fields. Just a few examples are the flow control mechanisms in TCP [23], the clock synchronization mechanisms in Network Time Protocol (NTP) [35], video frame rate control mechanism in Berkeley continuous media player [53], feedback-based scheduling in the Synthesis operating system [32].

These existing feedback mechanisms are generally implemented in a custom manner, and are hard-coded for a particular application in a particular environment. Consequently, they suffer from arbitrary structure and wasted effort due to repeated design and implementation of logically similar components. Their parameters are usually hard-wired and difficult to tune. They also lack the flexibility of easy extension for adaptation to new environments.

On the other hand, various control theories and toolkits already exist, and have been successfully used in the development of traditional hardware and embedded control systems, including feedback systems. In classical control theories, linear systems theory [3, 5] provides formal specification and analysis of linear systems, and nonlinear systems theory [14] helps design nonlinear systems based on various forms of linearization. Modern control theories such as fuzzy [34, 67] and neural [16, 67] control offer guidelines on dealing with dynamic nonlinear systems, though they generally are based only on intuition and experience, and lack a rigorous method. There are also toolkits, such as Matlab [60, 61] and MATRIX$_X$ [21], which are based on the above control theories. In these toolkits, there are libraries of control-theory conforming components, and tools to help the composition, simulation and analysis of control systems.

The origin and target of the control theories and toolkits in engineering fields are

feedback-based dynamic control of hardware and embedded systems. The target systems have relatively well-defined or understood range of dynamics (or domain). Many of them may also be modeled precisely. For example, one of the basic assumptions of (linear and nonlinear) control-theory-based feedback mechanisms is that the target system is *gradual* in the transition of its state. In other words, the system state usually does not jump suddenly. Thus *gradualness* is an important property in the domain of a control-theory-based feedback system. Fortunately, hardware and embedded systems usually have this gradualness in their dynamics for straightforward application of control-theory-based feedback mechanisms. As a result, the control theories and toolkits do not need to provide a means to handle the case when the actual system dynamics is out of the domain of the feedback mechanisms used. In case the dynamics of a target system goes beyond the domain of the feedback mechanism, the feedback tends to either suffer from performance degradation or to fail without detection. Furthermore, hardware feedback systems are implemented in capacitors, inductors and resistors. Embedded systems usually are programmed in low-level languages. For these reasons, the existing control-theory-based toolkits do not facilitate software implementation of feedback systems in high-level programming languages very well. They either do not generate executable code, or the code they generate is tightly coupled with their run-time environment, making it hard for the generated feedback systems to be efficiently incorporated into software systems.

Computer systems are becoming far more complex and dynamic than traditional hardware and embedded systems. They are also evolving rapidly in an unpredictable way. For example, Internet hosts range from low-end Internet appliances and PC's to workstations to mainframe supercomputers. Network links range from phone line at $28.8Kbps$, to Ethernet at 10 or $100Mbps$, to fiber links at up to $10Gbps$. Some of the Internet hosts are also mobile. They may switch between different network interfaces with totally different characteristics, such as between wireless PPP and Ethernet, while network applications are up and running. Applications with totally different requirements and behavior share the same Internet. They include bulk data transfer through FTP, interactive applications such as telnet, and real-time multimedia streaming applications. The Internet has been, and is still evolving in an unpredictable way. An exponentially-growing number of new

links and hosts are connected, and new types of applications are created constantly. No one can predict what the Internet will look like several years from now. Similarly, typical workstation operating systems must support a wide range of tasks, from background computational tasks, interactive editors, real-time multimedia applications, and system processes, to hard real-time tasks processing hardware interrupts. The mixture of the tasks also changes in an unpredictable manner.

To make software systems survive and work well in highly complex and dynamic computer environments, and adapt to unpredictable changes, we not only need to make use of software feedback mechanisms, but also need to make the feedback mechanisms themselves robust and adaptive. In contrast to a hardware and embedded feedback mechanisms that have a domain with the property of gradualness, a software feedback mechanism may have a domain with *discontinuity*. For example, a streaming video player on a mobile host will experience orders-of-magnitude *jump* in available network bandwidth upon switching between wireless PPP and Ethernet interfaces. The feedback mechanism in the player for network flow and congestion control should respond to the big jump quickly as well as adapting the playback quality to the network bandwidth variation caused by dynamic network traffic load. A domain with discontinuity can be seen as a union of multiple sub-domains, some of which may have the property of gradualness. Due to the holes between the sub-domains, the overall domain is *sparse* in some sense. A feedback mechanism with a sparse domain has a wide range of dynamics. It should not only work well within each sub-domain, but also needs to switch between these sub-domains dynamically when a jump between the domains happens. The feedback mechanism may need to change its parameters or internal structure dynamically in order to perform the switches. To reflect the characteristics mentioned here, a feedback mechanism of this type can be referred to as a *sparse-domain feedback mechanism*, a *wide-range feedback mechanism*, or an *adaptive-structure feedback mechanism*.

To build software feedback mechanisms efficiently, whenever possible, methodologies should be followed for composition from simple components, and software components already implemented should be reused to avoid waste of effort. There exists a big gap

between traditional control theories and toolkits, and the requirements of software feedback mechanisms and their development. This gap explains the fact that though control theories and toolkits are popular in the development of hardware and embedded control systems, they are unknown to most software engineers, and hardly used in the development of any software feedback mechanisms.

1.2 Contribution of the Thesis Research

To facilitate the design and implementation of wide-range software feedback systems[1], this thesis presents a *software feedback toolkit*. We propose a methodology for composition of software feedback systems, and implement a software feedback toolkit prototype consisting of a library of basic feedback components and a set of tools for feedback systems composition, simulation, and instrumentation. We also demonstrate the application of the software feedback toolkit in developing software feedback mechanisms for adaptive multimedia streaming applications. The contributions of the thesis research are as follows:

(1) A methodology for composition of wide-range software feedback systems

In the proposed methodology, software feedback components export a common interface. With this common interface, building blocks can be composed in a uniform way to form composite components. Feedback systems are built hierarchically based on existing building blocks and composite components. To help build wide-range feedback systems, the methodology makes guard-based meta-adaptation (a feedback system changes its own parameters or internal structure in order to adapt to wide range of dynamics in the target system) explicit, and introduces the concept of a guarded feedback component. A feedback component (or policy) has a domain in which it works well, and an associated set of assumptions on applicability, dynamics in input signals, performance and stability. A set of guards (each of which is a predicate testing if an assumption is valid or invalid) is placed around a feedback component against these assumptions. A guarded feedback

[1]Since the thesis will be focused on a toolkit for developing software feedback systems, when the context does not have ambiguity, terms such as "software feedback system", "software feedback", "feedback system", and "feedback mechanism" will be used interchangeably.

component is activated, deactivated, re-parameterized (change in parameters) or restructured (change in structure through dynamic replacing of one feedback component with another) upon triggering of its guards. A feedback system can be composed of multiple guarded components, each of which are in effect only when they are applicable. It switches between its components through guard-based meta-adaptation actions.

Guard-based meta-adaptation also helps the application of control theories, especially linear systems theory, in the development of wide-range software feedback systems. Control theories can be applied in the specification and analysis of feedback systems composed of theory-conforming components. Guards are placed around those components to inform the upper layer for meta-adaptation in the case the system dynamics exceeds their domains.

(2) Software feedback toolkit prototype

A prototype of the software feedback toolkit is implemented in C++. Base classes are defined for feedback components, and a library of building blocks is implemented, including various filters, regulators, connectors, etc. The software feedback toolkit has facilities to help compose composite feedback components out of building blocks, and to help dynamic component replugging. It also contains a set of tools for feedback system simulation, and on-line instrumentation. With this toolkit prototype, we demonstrate that the feedback composition methodology can be reduced to practice. We provide a detailed design of components that others may follow. This toolkit prototype also makes the technology proposed in this thesis readily available to software developers.

(3) Application of the software feedback toolkit for developing wide-range feedback mechanisms in adaptive multimedia systems

In order to demonstrate the feasibility of the software feedback toolkit and to allow evaluation of it, the toolkit is applied in the development of several feedback systems for adaptive multimedia applications. In the case of adaptive packet rate control, two types of network pipelines are identified, and corresponding feedback policies and their guards

are designed and implemented. Upon triggering of the guards, appropriate feedback policies are plugged in and activated, or deactivated and unplugged, dynamically. Dynamic replugging of feedback components makes the packet rate control adaptive to changes in network conditions. It also makes it easy to extend the packet rate control feedback when new types of networks are identified.

A media streaming control protocol (SCP) is proposed and implemented. SCP is a rate- and congestion-window-based flow and congestion control protocol for real-time streaming of multimedia data. The design of SCP follows the methodologies of the software feedback toolkit such as hierarchical feedback composition, and guard-based meta-adaptation. The building blocks from the toolkit component library are used extensively. The simulation and instrumentation tools then greatly help visualizing the behavior of SCP, and facilitate tuning-up of policies and parameters.

Finally, the software feedback toolkit is used to make a distributed real-time MPEG video player adaptive. The player is designed to stream video and audio across the Internet in real-time. The ever-changing available bandwidth and transmission latency of the Internet requires that the real-time player be highly adaptive. The feedback-based quality-of-service control mechanism, along with the SCP protocol mentioned above, adapts to many factors, including client processing speed, server disk bandwidth, network bandwidth, network delay and delay jitter, user preference, etc. The adaptive client-server synchronization feedback ensures synchronization between the server and the client in the face of changing network and host conditions, and keeps buffering latency minimum while ensuring smooth video playback.

1.3 Structure of the Thesis

The rest of the thesis is organized as follows. Chapter 2 presents the proposed methodology for composition of software feedback systems. In Chapter 3, a prototype implementation of the toolkit is described. Next, Chapters 4, 5 and 6 are dedicated to the application of the software feedback toolkit in the development of several feedback systems for adaptive distributed multimedia applications: an adaptive packet rate control protocol, an media

streaming flow and congestion control scheme, and adaptive QoS and synchronization control feedbacks in a distributed multimedia player. Then Chapter 7 discusses how some previous or ongoing projects relate to the research in this thesis. Finally, Chapter 8 concludes the thesis and discusses some of the interesting research issues we want to address in the future.

Chapter 2

Software Feedback Composition Methodology

2.1 Introduction

A *feedback system* continuously monitors the behavior of the *target system* being controlled, compares the actual behavior against a specification of the expected behavior, and adjusts the target system accordingly, to ensure that the behavior or performance of the target system is within the specification. Figure 2.1 shows the overall structure of a feedback system and its interaction with the target system. There are two main functional components in a feedback system: a filter and a regulator. A *filter* takes the output signal of the target system, which usually contains noise caused by the environment or measurement process. It filters out the transient noise and extracts useful information reflecting the behavior of the target system. A *regulator* compares the measured current system behavior against a goal specification, and applies control laws to adjust the target system accordingly.

There are many flavors of software feedback systems in terms of complexity and adaptability. Some are simple, requiring no filter, and a regulator with only minimum functionality. Others are complex, such as the QoS control feedback for the distributed video player to be discussed in Chapter 6. Some target systems have limited dynamics ranges, so that the feedback systems do not need to be highly adaptive. Yet others, such as Internet applications, are known to have a highly dynamic environment. Furthermore, Internet

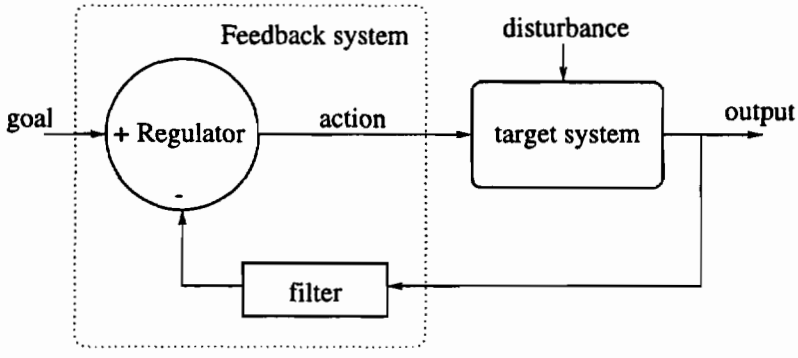


Figure 2.1: Overall structure of software feedback systems

applications not only need to work well in the current environment, but also should survive in the future. The software feedback mechanisms controlling these complex dynamic systems should work well across a wide range of system dynamics. Complexity and adaptability are two closely related properties. A feedback system is complex mainly because it needs to handle many different changing situations. Many software feedback systems are complex and adaptive, and call for methodologies in their development.

In this chapter we propose a methodology for the development of complex wide-range feedback systems. First, we model feedback components with a common interface, which makes it easy to compose different components. We discuss commonly used building blocks, and how simple components are composed hierarchically to form complex components. Next, by making guard-based meta-adaptation explicit and introducing the concept of guarded feedback component, we show how complex wide-range feedback systems can be composed based on a set of simple feedback policies. A wide-range feedback system is composed of multiple guarded component sub-systems. Triggering of a guard changes the parameters or state of the guarded component and its relations with the rest of the overall feedback system, thus making the structure of the overall feedback system adaptive to the dynamic environment. The development of a complex wide-range software feedback system follows the proposed methodology by decomposing the feedback into cooperative guarded components. Then the components are implemented on top of the building blocks provided by the toolkit, and composed to form the whole feedback system. Finally, we discuss how to make use of the results of existing control theories to build feedback systems

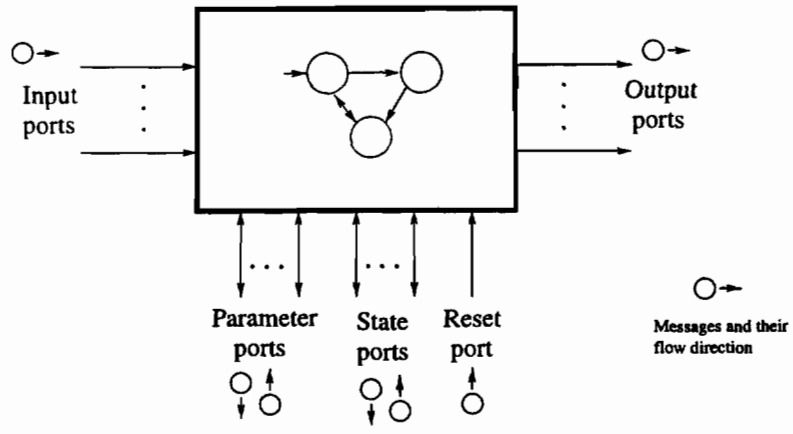


Figure 2.2: Model of software feedback components

with predictable theoretical properties.

2.2 Feedback Component

2.2.1 Feedback Component Model

A *feedback component* is modeled as a parameterized message-driven object with a set of input and output ports that interface to the outside world. All feedback components, no matter what their structures or functionalities are, have a common interface, so that they can be composed to form more complex components. The only way an external entity manipulates a feedback component is by exchanging messages through its various ports. Figure 2.2 shows the model of software feedback components. The elements of this model are listed as follows.

- **Message** A message is a signal with or without an associated set of values. Messages flow into, and out of feedback components through various types of input and output ports.
- **Port** A feedback component interfaces with the outside world through a set of message input-output ports. A port receives or sends out messages. As shown in Fig. 2.2, and to be further discussed below, there are input and output ports for feedback messages, parameter ports, state ports and a reset port.

- **Message-driven object** This object is the body of a feedback component. It receives and processes input messages and produces output messages. A feedback component object can have *internal states*, and may expose some of them to the outside world through its state ports. It may also have a set of *parameters*, and expose them to the outside through parameter ports. An important difference between a state and a parameter of a feedback component is that a state is updated by the component itself, while the value of a parameter remains constant unless updated by external entities. For example, in a lowpass filter for estimating the available bandwidth of a network connection, the recent average bandwidth measurement is the state, and the time constant of the filter is a parameter. Further structure of the component object is not specified in the model, and is up to the specific implementation of the model or component.
- **Input port** A feedback component can have zero or more input ports. Messages containing feedback data, also called feedback messages, are passed to a feedback component through its input ports. Input ports are asynchronous. They do not necessarily receive messages simultaneously, and a message from any input port may independently trigger state changes or message output. However, in individual components, the input ports can be related to each other or made synchronous through explicit programming.
- **Output port** A feedback component can have zero or more output ports. Feedback messages generated by a feedback component are sent to its output ports. Again, output ports are modeled as asynchronous, and can be related with each other and with the input ports through explicit programming.
- **Parameter port** A feedback component may expose its parameters to the outside for retrieval or update. Each exposed parameter is associated with a bidirectional parameter port. When a feedback component receives a message containing a parameter value from one of its parameter ports, it changes the corresponding parameter to the value contained in the message. When a component receives a message containing a request of parameter value from one of its parameter ports, it sends a message containing the current value of the corresponding parameter to the parameter port.

- **State port** A feedback component may also expose its internal states to the outside for retrieval or update. Each exposed state is associated with a bidirectional state port. An internal state is retrieved or updated through its corresponding state port in a manner similar to that in which a parameter is accessed.
- **Reset port** Each feedback component has one inbound reset port. Upon receiving a message from this port, the feedback component resets its internal states to the initial (default) values while keeping its parameters unchanged.

A feedback component has several stages in its life cycle. During initialization, its parameters and internal states are initialized to a default value. During normal operation, it receives feedback messages from input ports, processes them, manipulates its internal states, and generates output messages. Its parameters are updated upon receiving messages from its parameter ports. A message presented to the reset port of a feedback component resets its internal states to initial values. If all parameters and internal states are exposed, then a snapshot of its parameters and states can be saved and later restored, either to the same component instance, or to another instance of a similar type of component. This ability to save and restore parameters and states helps with dynamic replugging of feedback policies, which will be discussed later in this chapter.

2.2.2 Basic Feedback Component

Basic feedback components, also called *building blocks*, are those which have simple and well-defined functionalities and are unnecessary to further break down into even simpler components. Several criteria can be used to categorize feedback building blocks. Based on their roles in feedback systems and component composition, there are signal sensors, filters, regulators, connectors, and action generators. Depending on its complexity, a feedback system may consist of only some, or all of these types of components. A *signal sensor* collects measurements from a target system. A *filter* extracts information from noisy measurements about the parameters, performance or behavior of the target system. A *regulator* applies control laws on its input from filters to make decisions for adjusting the target system. A *connector* routes messages around to help connect feedback components.

Finally, an *action generator* adjusts the target system based on decisions made by the regulator. Among the above types of components, filters, regulators and connectors have a common interface, while signal sensors and action generators may be application specific, and may have ad-hoc interfaces to the target system. Similarly, depending on whether a feedback component or system conforms to linear systems theory [3, 5] or not, it can be classified as either linear or nonlinear. A *linear component* conforms to linear systems theory and is amenable to specification and analysis based on that theory. On the other hand, a *nonlinear component* has nonlinear behavior and cannot be characterized using linear theory for specification or analysis.

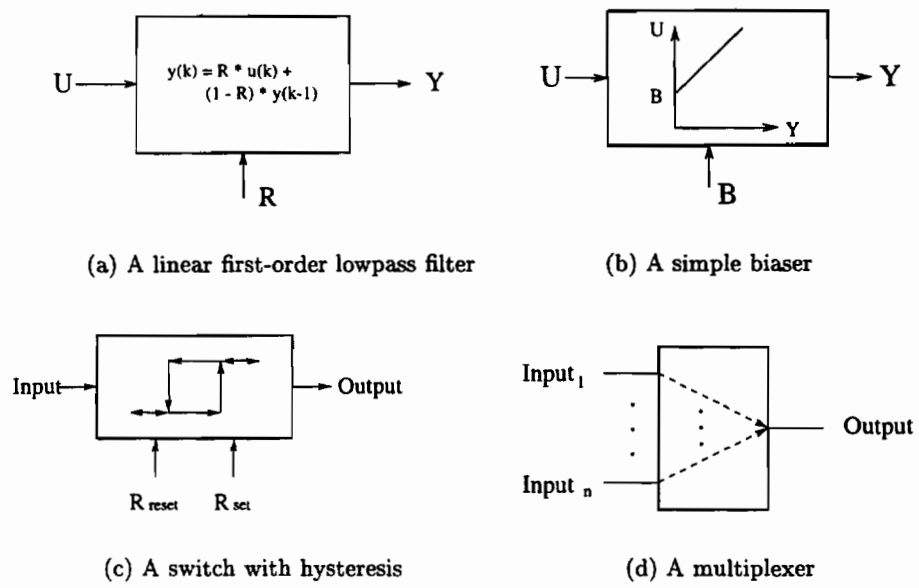
One example feedback building block is a simple first-order lowpass filter [5] shown in Fig. 2.3(a). This filter is a linear component with one input port, one output port, and a parameter port. Taking an input sequence of measurements $\{u(k)\}$ $(k \geq 0)$ and an output sequence $\{y(k)\}$ $(k \geq 0)$, the lowpass filter with a parameter R $(0 \leq R \leq 1.0)$ is defined by the following equation. Here we assume an initial condition of $y(-1) = 0$.

$$y(k) \quad = \quad R * u(k) + (1.0 - R) * y(k-1) \qquad \text{where } k \geq 0$$

The output of the lowpass filter is an estimator of the average of its recent inputs. The parameter R is an aging factor that specifies how much contribution old values have to the average. The filter has a internal state variable to hold to previous output $y(k-1)$, but does not export it.

Another example is a simple biaser shown in Fig. 2.3(b). For each input, it generates an output which is the sum of the input and its current parameter value. This biaser is a linear component usually used as a regulator. For example, in Chapter 4, we will present a packet rate control feedback for lossy networks, which uses a biaser to generate the rate at which the server sends packets in the future.

Figure 2.3(c) gives an example of a switch component with hysteresis. This component has one input port, one output port, and two parameters R_{reset} and R_{set}, where $R_{reset} < R_{set}$. When the input of the switch goes beyond R_{set}, the switch is turned on. When the input goes below R_{reset}, it is turned off. When the input is in the range of (R_{reset}, R_{set}), the state of the switch stays where it is, which can either be on or off. This component is



(a) A linear first-order lowpass filter

(b) A simple biaser

(c) A switch with hysteresis

(d) A multiplexer

Figure 2.3: Feedback building block examples

a nonlinear regulator commonly used in temperature control systems.

Finally, we see an example of a multiplexer in Fig. 2.3(d). This multiplexer has multiple input ports, one output port, and no parameter port nor state port. A message received from any input port is passed to the output port without processing or delay.

2.3 Hierarchical Feedback System Composition

Feedback systems are implemented by composing feedback components in a hierarchical manner, as shown in Fig. 2.4. Feedback building blocks live at the bottom of the hierarchy. A composite feedback component, which is a composition of simpler components, sits at one of the higher levels. A *feedback system* is a feedback component at the top of the feedback composition hierarchy.

A *composite feedback component* consists of a set of simpler components (called subcomponents) which are either building blocks, or composite components themselves. The general structure of a composite component can be seen in Fig. 2.5. The subcomponents are composed by connecting their message input and output ports together. An output port of a component can either be open, or be connected to one or several message input

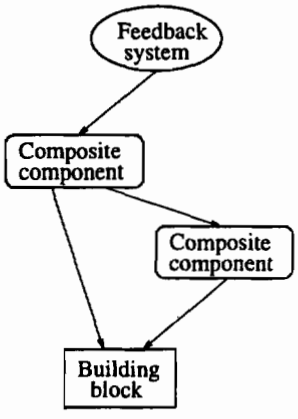


Figure 2.4: Feedback system composition hierarchy

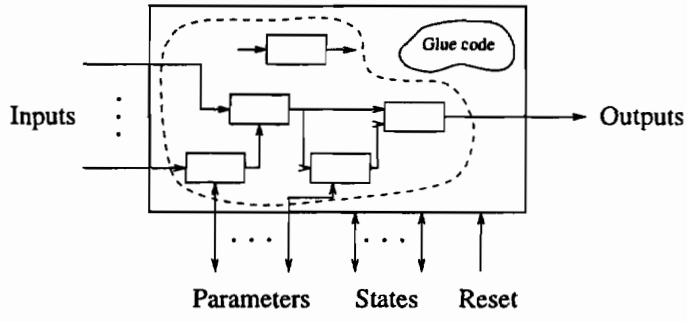


Figure 2.5: The structure of composite feedback components

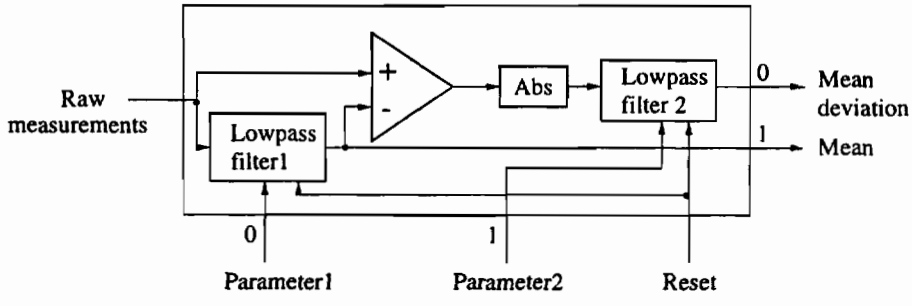


Figure 2.6: Composite feedback component for estimation of mean and mean deviation

ports, including input ports, parameter ports, state ports, or reset ports, of other components. When a message is presented at an output port, it is passed to all the connected message input ports. Some of the input, output, parameter or state ports are exported as ports of the composite component. Reset ports of all the subcomponents are connected and exposed as a single reset port of the complex component.

Glue code, which is specific to each individual component, is also an essential part of a composite component, as indicated by Fig. 2.5. In a composite component, there may be some parts that are hard or impossible to implement by composing subcomponents. One example is the dynamic replugging of subcomponents to be discussed in the next section. Implementation of these component-specific parts is an important role of the glue code. Further modeling of the various operations performed by the glue code will not be explored by this thesis, but is part of the future research.

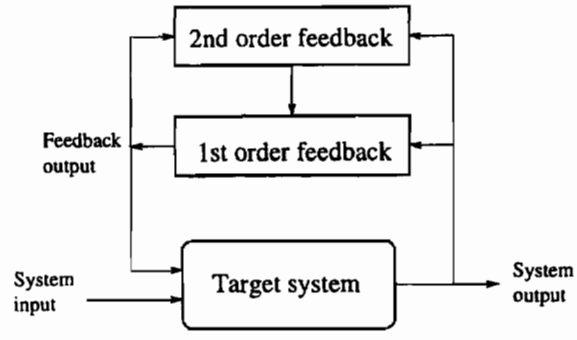


Figure 2.7: Higher-order feedback

One example of a composite feedback component is a filter for estimating the mean and mean deviation of a sequence of measurements. Suppose the raw measurement sequence is M_i ($i \geq 0$), then its mean A_i can be estimated by applying a lowpass filter shown in Fig. 2.3(a): $A_i = lowpass(M_i)$. The mean deviation is the average of $|M_i - A_i|$, which can be gained by using a component to compute the difference, and one for the absolute value, and finally a lowpass filter of the same type used for calculating A_i. Figure 2.6 shows the structure of the composite filter. This filter is composed of two lowpass filters and two other components. It has one input port for the raw measurements, two output ports, one for the mean and the other for the mean deviation. It also has two parameter ports, one for each lowpass filter.

There is an interesting case of feedback system composition when a feedback component sends messages to the parameter ports instead of the normal input ports of other components. In this case the former feedback component does not control the target system directly. Instead, it makes the latter feedback component more adaptive by changing its parameters. This case of one feedback component controlling another one is called *higher-order feedback*, as shown in Fig. 2.7. Higher-order feedback can be found in many complex software feedback systems. One example is the TCP flow control protocol, in which there is a time-out-and-retransmission mechanism to decide when to retransmit packets. The time-out time is set by a lowpass filter that continuously monitors the mean and variance of connection round-trip time.

Another interesting case is that a wide-range feedback system is likely to have a dynamic instead of static structure. Based on the current state of the target system and its changes, feedback components may be dynamically plugged into the working feedback system and activated, or deactivated and unplugged. This dynamic structural change is referred to as *dynamic replugging* of feedback components and will be further discussed in the next section.

2.4 Guard-Based Meta-Adaptation for Wide-range Feedback System Composition

As discussed in Chapter 1, computer systems are complex and highly dynamic. Their environment also evolves over time. The feedback systems in this environment need to work well across a wide range of dynamics, in a domain with discontinuity. On the other hand, an individual feedback policy usually has a limited domain. In particular, a control-theory-based feedback system usually assumes that the state transition in its target system is continuous. It is useful to compose a complex wide-range software feedback system based on a set of component feedback systems. A component may be active when the dynamics of the target system falls into its domain. When the system dynamics moves from the domain of one component to that of another, the overall feedback system takes actions, such as switching to the new component, in order to adapt appropriately.

In this section, we propose an approach to compose wide-range feedback systems with component feedback policies (sub-systems). We make guard-based meta-adaptation explicit and introduce the concept of guarded feedback component. The domain in which a component feedback system works well is identified. It is represented by a set of assumptions, each of which is guarded by placing corresponding guards around the component. The overall feedback system is composed of multiple, repluggable, guarded feedback components. Upon triggering of a guard, the overall feedback system changes the parameters or the structure of the guarded component, or deactivates and unplugs it, and plugs-in and activates a new component. Through dynamic replugging of the components, the overall feedback system works across the combination of the domains of the components.

Thus its domain is the union of those of its components. If there are gaps between the component domains, the over-all domain would have the property of discontinuity. We explore the possible ways of feedback system meta-adaptation, and discuss the types of events that need to be guarded. Much of the terminology used in describing the model of guard-based meta-adaptation are borrowed from the Synthetix project [44], which is summarized in Section 7.2.

2.4.1 Guarded Feedback Components and Their Composition

A feedback component implementing a single function (e.g., feedback policy or filtering algorithm) has a *domain* as a subset of its whole input space, in which it works well. This statement is especially true for filters implementing single algorithms and regulators enforcing simple feedback policies. If the dynamics of the target system goes beyond its domain, the feedback component may suffer performance degradation, have undesirable behavior, or simply break and become non-applicable. For example, the lowpass filter shown in Fig. 2.3(a) only makes sense if the variation is mostly caused by noise, and the true value is either constant, or changes gradually. So its domain can be stated as *high-frequency noise, actual input keeps constant or changes gradually.* If the actual input value jumps in big steps, then the tracking accuracy of the filter will be seriously compromised.

The domain of a feedback component can be represented by a set of *assumptions* concerning the applicability, input signals, performance, or properties such as stability and responsiveness. Some of the assumptions are true *invariants*, which are known to be valid all the time, while others are likely to be valid most of the time, but not always. This later type of assumption can be referred to as a *quasi-invariant.* For example, on a specific platform, word size, cache size, whether a processor has a FPU, etc., are invariants. On the other hand, in a personal computer invironment, it is likely, but not certain, that a file is not shared by more than one process. Thus "a file is not shared" is a *quasi-invariant.* To ensure that the quasi-invariants remain valid throughout the time when the feedback component is in effect, a set of *guards* are placed around the component. A guard monitors the events that may change the validity of the assumption it guards. Whenever an event validates or invalidates an assumption, a guard is triggered, and proper actions are taken

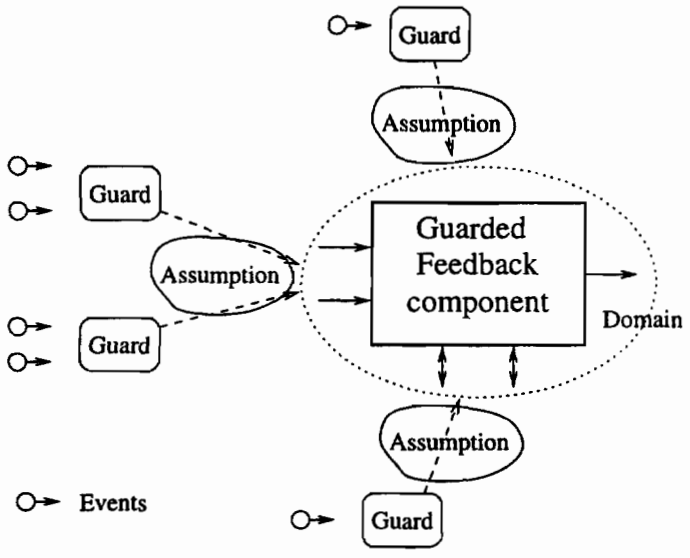


Figure 2.8: Guarded software feedback component

to change either the parameters or internal structure of the guarded components, or to activate or deactivate it. These actions are collectively referred to as *meta-adaptation* actions. A feedback component with a set of guards, which may be triggered by events and results in meta-adaptation on it is called a *guarded feedback component.* Figure 2.8 shows a scenario of this concept. The meta-adaptation of feedback systems based upon triggering of guards is referred as *guard-based meta-adaptation.* The process of composing multiple guarded feedback components, each of which has a limited domain, into a wide-range feedback system with a domain as a union of that of its components through guard-based meta-adaptation is then referred to as *multi-domain composition.* The resultant wide-range feedback system can be called an *adaptive-structure feedback system* if it has dynamically repluggable components as well as adjustable parameters. It can also be called a *sparse-domain feedback system* if there are gaps between the domains of its components,

2.4.2 Meta-Adaptation Actions

Upon triggering one of its guards, a feedback system may adapt in following ways: changing component parameters, resetting components, plugging or unplugging components, or signalling exceptions to the application.

Change of Component Parameter Value

A feedback component changes the values of its parameters upon receiving messages from its parameter ports. For example, in the low pass filter shown in Fig. 2.3(a), the value of the parameter R can be changed through its parameter port. Component parameter value change is a light-weight adaptation with very little overhead.

Component State Resetting

A feedback component resets its state to initial default values upon a signal sent to its reset port. Resetting is an effective way for a component to discard its no-longer-valid state. For example, in the video player to be discussed in Chapter 6, the client is capable of switching between PPP and Ethernet interfaces while a playback session is going. Upon a switch in network interface, the lowpass filter used for estimation of video display frame rate should be reset to discard the no-longer-valid frame-rate estimation.

Dynamic Component Replugging

Another way for a feedback system to adapt to new situations is to change its structure dynamically. As shown in Table 2.1, there are several ways in which a feedback system can change its structure: re-wiring the ports of feedback components, deactivating and unplugging existing guarded components from the working system, plugging-in and activating guarded components into the working system, or replacing existing components with new ones implementing a different policy. We call this process *dynamic component replugging*. In the adaptive packet rate control feedback in Chapter 4, there are two feedback policies, one to handle packet loss, the other for latency build-up. Each policy has a guard detecting if corresponding congestion conditions show up. Upon triggering of a guard, the set of components for the corresponding policy are plugged in. If the corresponding congestion condition no longer exists, the components of a policy are unplugged from the working feedback system.

An alternative to dynamic component replugging is to leave all components plugged in all the time, and use a mechanism, such as case-statement, to dynamically determine what components to invoke upon each input signal. For example, in the adaptive packet

Case	Description
Component re-wiring	Ports of components are re-connected
Component unplug	A component is deactivated and disconnected from the working feedback system
Component plug-in	A component is connected to the working feedback system and activated
Component replacement	An existing component is replaced by a new one with a different policy.

Table 2.1: Cases of dynamic component replugging

rate control feedback in Chapter 4, instead of having two repluggable feedback policies, we could have the two policies always plugged in. There would also be two flags indicating whether packet-loss or latency-increase has been detected. Each time when the packet rate control feedback is invoked, the flags are tested and appropriate policies are invoked accordingly. This alternative case-statement approach has several disadvantages. The first is that continuous testing of flags upon each invocation results in extra execution overhead. Dynamic component replugging eliminates this type of overhead, but introduces overhead for replugging of components. Fortunately, in most cases, dynamic component replugging only happens very infrequently, and is thus more efficient in execution. The Synthetix project [44], to be further discussed in Section 7.2, has demonstrated that in file `read()` system call, dynamic code generation (similar to dynamic component replugging) is more efficient than case-statements. Another disadvantage of the case-statement approach is that it is inflexible. A case-statement needs to know, during coding time, exactly what polices are to be invoked. As a result, dynamically extending a feedback mechanism for new situations is impossible. On the other hand, with dynamic component replugging, it is easy to extend an existing feedback mechanism, since the feedback mechanism does not need to know exactly which policy is being invoked, as long as the potential policies have the same interface.

In our feedback component model, a component exports input, output, parameter and state ports. The way external entities explicitly trigger adaptation of the component is to send messages to its parameter ports. in a complex component, some parameters may associate with its internal structure, changing these parameters actually changes the

structure, thus performing dynamic replugging of subcomponents. One example is the PLL simulator discussed in Section 3.5. In that example, there is a composite filter for the PLL policy, which has a parameter associated with a dynamically repluggable component to eliminate phase-error residue. If the parameter of the filter is set to zero, the the compensator is unplugged, otherwise it stays plugged in.

Dynamic component replugging is considered heavy-weight adaptation. In its simplest form, re-wiring between components is all that is needed. However, components may need to be allocated or reclaimed on the fly. In a more expensive case, the code of the new component may need to be synthesized at run-time. As discussed in the next section, synthesis of the code of linear feedback components is fairly reasonable, because of the availability of rigorous specification and predictability in properties of linear feedback components.

The feedback component model proposed in this chapter helps build feedback systems with dynamic structure in several ways. First, all feedback components have a common interface, so that composition of components is simple and straightforward. Second, each feedback component also has a set of ports for accessing its parameters and internal states, so that the parameters and state of to-be-unplugged components can be saved, and restored in newly-plugged-in components if necessary. For example, suppose a lowpass filter is to be replaced by a median filter[1]. The current state, i.e., the current estimator, can be retrieved from the lowpass filter, and restored to the median filter. Thus, the median filter can use this value instead of a default value as an initial estimation.

Exception

Because software systems are highly dynamic and evolve all the time, it may be the case that an originally unexpected event happens and triggers some guards, but the feedback system is not designed to handle the new situation yet. In this case, all the assumptions on which the whole feedback system is based become invalid, and an exception should be signaled to the application for further processing. The application can either take

[1] A median filter outputs the median of the recent set of input samples, instead of the average of them. See the toolkit user's manual [7] for more information.

measures outside of the feedback system, or even download a new feedback policy into the existing feedback system on demand. Here we envision an open feedback architecture: feedback systems are extensible, and have the interface to receive and plug-in new feedback components synthesized by external entities. Further investigation of the idea of open feedback architectures will not be discussed in the thesis, but is part of the future work.

We again can use the adaptive packet rate control feedback in Chapter 4 as an example. For both the feedback policies used, there is an assumption that the nominal network bandwidth is relatively stable, and changes slowly if there are changes, so that the feedback can catch up with the bandwidth change. In the current Internet environment, as far as our experiments go, it remains true that the available bandwidth averaged along a reasonable period of time does not change drastically all the time. But in the future, with the deployment of high-speed networks having a high throughput-latency product and more dynamic applications competing for the bandwidth, there is a chance that the assumption will break, and result in performance degradation in the form of higher-than-expected packet loss. To prevent this performance degradation from happening without detection, we can have a guard to monitor the packet-loss ratio, and when it becomes too high, to signal an exception to the application. The application may then react accordingly.

2.4.3 Events to be Guarded

Having defined a software feedback component, stated all its assumptions, and how the feedback component adapts upon validation or invalidation of the assumptions, the next step is to find out where to place the guards. Specifically, where the events are generated and how to guard against them are usually specific to individual applications and implementations. Nevertheless, in this section, we try to identify several areas where events may happen, and guards may need to be placed, as shown in Table 2.2.

Input Signal Properties

The most obvious place to put guards is on the input signals of a guarded component. The magnitude, derivative, continuity, and other properties of the input signals can be monitored and guarded when necessary.

Event	Explanation
Input signal property	Including magnitude and its derivative, continuity, etc.
Target system parameters	Examples are network bandwidth, latency and jitter
Feedback performance degradation	Examples are increase in network data drop ratio
Explicit event	Explicit action by the user or application, such as change in play speed or video picture size
Timer expiration	Used to detect if other events have happened recently

Table 2.2: Types of events which trigger feedback adaptation

Target System Parameter Changes

A software feedback system should be aware of the parameters of its target system and adapt when the parameters change. Some parameters are directly embedded in input signals to the feedback system. Others may require explicit probing. For example, a network connection has parameters including available bandwidth, latency, and jitter. In the receiver side, jitter can be directly measured based on the time-stamp sampled when packets are received. Latency needs to be probed if the sender and receiver clocks are not synchronized. Available bandwidth always needs to be measured explicitly.

Feedback Performance Degradation

When the model of a target system on which a software feedback is based is broken, one of the symptoms is that the performance of the feedback control degrades. Performance degradation may also result from changes in target system parameters, in which case the parameters can be measured indirectly through monitoring feedback performance. A widely used method to measure available network bandwidth is to inject packets at an increasing rate. Monitors at the receiver side then observe how the network reacts. In the MPEG player in Chapter 6, the frame rate feedback tries to manage the difference between video server send frame rate and client display frame rate, i.e., the frame-drop ratio by the video pipeline, within a specified value. However, if the available bandwidth of the video pipeline oscillates too quickly and widely, the performance of the frame rate feedback will degrade, with the frame-drop ratio exceeding specification.

Explicit Events

Operating systems can generate explicit events that feedback mechanisms should not ignore. For example, a mobile host may switch between different network interfaces (e.g., wireless PPP and Ethernet) while network applications are still running. In this case, the feedback mechanisms in these applications should take appropriate meta-adaptation actions to adapt to the sudden change in network capability. In the streaming video player to be discussed in Chapter 6, all the feedback mechanisms — for network flow and congestion control, presentation quality control and client-server synchronization — intercept network interface switching events, and react properly.

Applications may also explicitly generate events to which software feedback should adapt. One example is that in the MPEG video player in Chapter 6, the user can change video spatial resolution and play speed through the GUI. Changing the resolution changes the number of bits needed to encode the frames, thus effectively changing the available network bandwidth in terms of frame rate. The frame-rate feedback adapts to the new bandwidth quickly through meta-adaptation, instead of slowly through the steady-state frame-rate feedback policy.

Timer Expiration

Timers can also trigger adaptation of feedback systems. Timers are useful in detecting if other events have happened recently. Each event of a certain type resets a companion timer. If events happen frequently enough, the timer never expires, indicating that the condition associated with the events still holds. Otherwise, the timer expires, and triggers unplugging of the feedback components that reacts to the condition associated with the events. Later, when the event happens again, the unplugged components will be plugged back into the feedback system. For example, in the adaptive packet rate feedback in Chapter 4, each time the loss of a packet is detected (a packet-loss event happens), the feedback system resets a timer, as well as plugging in the packet-loss feedback component if it is not plugged-in yet. If the timer has not been reset in a specified period, it expires and triggers unplugging of the packet-loss component.

2.4.4 Discussion

With the introduction of guard-based meta-adaptation, it becomes easy to build feedback-based multi-level adaptation mechanisms for adaptive systems. A mechanism of this type consists of a set of guarded feedback components and a meta-adaptation layer, which glues the components together through guards and meta-adaptation actions. We will also be able to make multi-level performance or stability statements about a multi-level adaptation mechanism. At one (lower) level, we specify the performance or stability domain of each guarded feedback component. At another (higher) level, we make statements on how the overall adaptation mechanism transfers between the domains. With these multi-level performance statements, the policies of an adaptation mechanism can be tailored to the preference of individual users or the specific operating environment, and thus yield performance better than can be reached by a pure best-effort system.

A good example of multi-level adaptation is the QoS feedback for video resolution and frame-rate adaptation discussed in Chapter 6. This QoS feedback has two levels. At the lower level, a frame-rate feedback component maintains the optimal display frame-rate that the video player can sustain with the currently available resources. At the higher level is a spatial-resolution adaptation policy, which specifies when to stay with the current resolution, and when to switch to a higher or lower resolution. Whenever the frame rate yielded by the frame-rate feedback is too low, a guard is triggered to switch to a lower resolution. If the frame rate is too high, then another guard is triggered to scale up the resolution. Otherwise, the video player keeps playing video at the current resolution. With this multi-level QoS feedback, the users are able to specify their preference between video frame rate and resolution, even though they do not have control on the availability of the resources.

2.5 Composing Software Feedback with Predictable Theoretical Properties

In the previous sections, we proposed a methodology for building complex and adaptive software feedback systems. A complex software feedback system is decomposed into a

set of cooperative, guarded feedback components, each of which has a domain in which it works well. The components are implemented and composed. In many cases, the feedback components would be amenable to specification and analysis by control theories, especially linear systems theories [3, 5, 6, 14, 67]. Control theories can be applied to analyze or predict theoretical properties, to specify feedback systems at an abstract level or synthesize components from their formal specifications, or to identify guards. Control theories also provides a rich set of building blocks. In this section, we briefly describe the various control system theories and discuss about how they can be applied in the software feedback toolkit.

2.5.1 Theoretical Properties of Feedback Systems

The theoretical properties addressed by control theories [3, 5, 6, 14, 67] include formal specification, stability, time and frequency response, system composition, etc. Some control systems may be specified in mathematical formulas, so that their theoretical properties can be clearly seen. The stability of a feedback system concerns whether the output is bounded in response to input that is also bounded. The time response of a system answers the question of how quickly or sluggishly it responds to changes in its input, and its frequency response is about how it discriminates the energy of the input at some frequencies against others. Finally for some systems, their formal specifications can be derived from those of their components, thus their specification and analysis can be simplified. Some examples of specification derivation are shown in Figure 2.9. There has also been much effort and theoretical results in application of general control theories to various complex situations [19, 39].

In this section, we highlight aspects of various control theories, especially those closely related to the software feedback toolkit proposed. Those who are interested in understanding more details of control theories should refer to relevant texts [3, 5, 6, 14, 19, 39, 67].

Linear Systems

Linear systems theories [3, 5, 6] deal with linear control systems. There are two types of linear systems: continuous-time and discrete-time. Linear systems can be specified

formally in various forms. There are differential functions, s-transforms and state space equations for continuous-time systems [3], and difference functions, z-transforms and state space equations for discrete-time systems [5]. The formal methods make it possible to infer the theoretical properties, such as stability, time and frequency responses, right from their specifications. Linear systems have the global stability property [5, 6], i.e., a stable linear system converges in all of its input space. Linear systems also have properties such as composibility of components at the the abstract level, the principle of superposition on input signals [5], etc. These properties make linear feedback systems predictable. On the other hand, the property of linearity restricts the application of the results of linear systems theories in real-life situations, where many problems are inherently nonlinear. Discrete-time linear systems theory will be discussed in more detail in the next subsection.

Nonlinear Systems

While the stability property of linear systems is always global, the same is not true for nonlinear systems. A nonlinear system being stable in the neighborhood of an equilibrium point does not necessarily imply any global property. There may indeed be many equilibria, some stable and others not, in which case there is only a limited region of convergence (domain of attractions around any locally asymptotically stable equilibrium). There can be nonlinear behaviors such as persistent oscillations (also known as limit cycles), which are dynamic rather than static equilibria. Even quasi-stochastic situations may arise despite the deterministic nature of system equations. This phenomenon is known as chaos and is quite prevalent in discrete-time nonlinear systems. In all cases, the behavior of a nonlinear system may critically depend on the input applied to it.

No generally applicable rigorous specification or analysis methods are available for nonlinear systems. Nevertheless, there have been guidelines for various types of nonlinear systems. Linearization is a very popular means of specification and analysis [14]. A nonlinear system is linearized around some operating points, and linear control systems are designed. During operations, when moving from one operating point to another, appropriate linear control components are brought into operation, or their parameters are changed accordingly. Techniques based on linearization include gain scheduling, adaptive control,

and variable structure systems [14]. Another means is nonlinear analogy to linear theory, such as harmonics analysis [14] as compared to linear transfer functions. Finally, some (especially complex) nonlinear systems are analyzed in a totally empirical manner [19], in many cases through extensive simulations.

Fuzzy and Neural Control Systems

Observing the difficulties in design and analysis of nonlinear systems based on classical mathematical methods, there have been developed fuzzy [34, 67] and neural [16, 67] control theories for building nonlinear systems directly, based on intuition and experience instead of formal methods. Fuzzy control is based on the idea of fuzzy sets and fuzzy logic, the key to which is to develop a framework with imprecision. In a fuzzy set, each member is represented by a pair $< value, belonging >$, with not only the value, but also the degree of belonging. Fuzzy sets can be identified by linguistic variables. For example, the temperature in a room can be classified as either *low*, *normal*, or *high*. A specific temperature reading, say 75°F, may fall to all the above classifications with certain degrees of belonging. In fuzzy logic, operations such as *and*, *or*, *not* take both the value and degree of belonging into consideration. In a fuzzy control system, input signals are first fuzzified into linguistic variables, and then applied with fuzzy control laws. The output of the fuzzy logic are defuzzified back to values for control of target systems. Neural control [67] is based on neural networks [16], which are rough analogies of biological neural systems. Neural networks are self-learning. They can be trained with a large number of inputs, and then used as filters or control laws in control systems. Fuzzy and neural control seem more natural in handling complex and imprecise systems, and are more easily understood by the practitioners.

2.5.2 Theoretical Properties of Discrete-Time Linear Systems

In this section, we discuss the discrete-time linear systems theory in more detail. Then in the next two Sections 2.5.3 and 2.5.4, we will discuss how this linear systems theory can be used in building feedback systems with predictable properties, and present the phase-lock loop, an example linear system, respectively

A discrete-time linear system [5, 6] can be represented in various forms: difference function, summation-convolution, z-transfer function, or state-space equations. Suppose the input and output sequences of a discrete-time system are $\{u(k)\}$ and $\{y(k)\}$ respectively, where $-\infty < k < \infty$. Then its n-th order difference function is shown below, where b_i and a_j are coefficients (which may or may not vary over time),[2] and m and n are fixed non-negative integers. The current output is a linear combination of the recent history of its input and output sequences.

$$y(k) = \sum_{i=0}^{m} b_i u(k-i) + \sum_{j=1}^{n} a_j y(k-i)$$

A discrete-time linear system can also be specified by a state-space equation [5], where states are introduced, and the output and next step states are linear combinations of current input and state. The coefficients a_i and b_j determine all the properties of a linear feedback system: its stability, time and frequency response, etc.

Yet another form is a convolution-summation equation [5]. The current output of a linear system, $y(k)$ at step k is a polynomial of all its input history with a sequence of weighting coefficients $\{h(i)\}$ $(0 \leq i < \infty)$:

$$y(k) = \sum_{i=0}^{\infty} h(i) u(k-i)$$

A fourth form is called a z-transfer function [5] (or simply called transfer function hereafter when there is no confusion). Given a sequence $\{f(k)\}$, where $f(k) = 0$ for all $k < 0$, its z-transform is defined as below, where z is a complex variable.

$$Z[f(k)] = F(z) = \sum_{k=0}^{\infty} f(k) z^{-k}$$

The set of z in the complex z-plane for which $\mid F(z) \mid$ is finite is called the region of convergence, while the rest is called the region of divergence. As a matter of fact, virtually all sequences of any interest have z-transforms expressible as a ratio of polynomials in the variable z [5]. These z-transforms will have zeros (the roots of its numerator polynomial) and poles (the roots of its denominator polynomial), and one of the poles will pass through

[2] In this section, we only focus on constant coefficient discrete-time linear systems.

the boundary separating the regions of convergence and divergence. z-transforms have properties including linearity, left- and right-shifting, convolution-summation, etc. [5].

Suppose for a linear system, its initial input and output are all zero, i.e., $u(k) = 0$ and $y(k) = 0$ for all $k < 0$, then its convolution-summation equation can be converted to the following transfer function, where $U(z)$, $Y(z)$ and $H(z)$ are the z-transforms of input $\{u(k)\}$, output $\{y(k)\}$ and coefficient $\{h(k)\}$ sequences respectively.

$$Y(z) = H(z)U(z)$$

Discrete-time linear systems bear various theoretical properties, and are amenable to formal analysis. First, their stability is global and determined solely by their structure. A linear system is stable if its output remains bounded in response to any bounded input, and its transfer function tells this right away: all the poles have magnitudes less than one. Second, there is the principle of superposition, which states that different components of a single input can pass through a linear system independently without interfering with each other. If a linear system's responses to the input $u_1(k)$ and $u_2(k)$ are $y_1(k)$ and $y_2(k)$ respectively, then the system's response to the input:

$$u(k) = a_1u_1(k) + a_2u_2(k)$$

is

$$y(k) = a_1y_1(k) + a_2y_2(k)$$

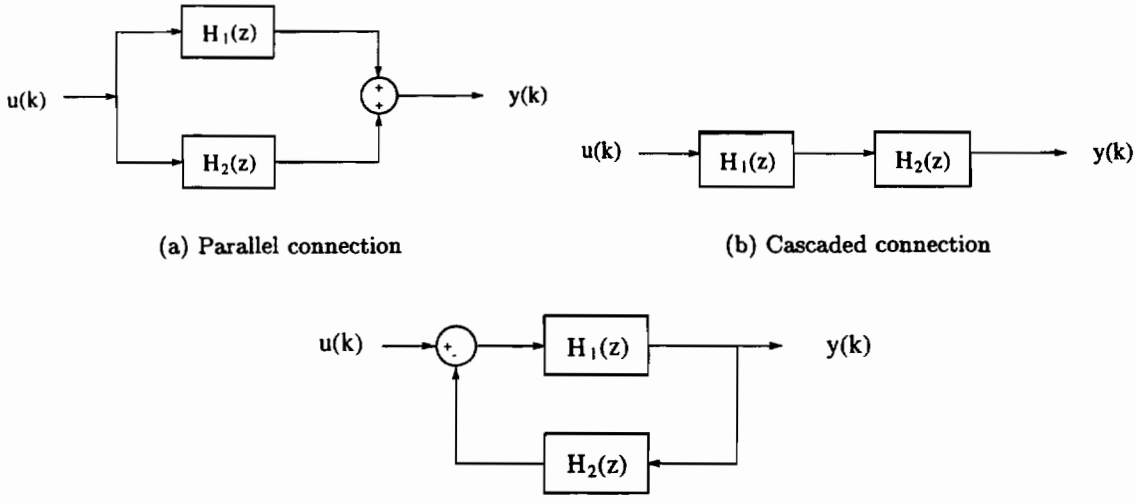
This principle of superposition is very helpful in the design and analysis of various linear filters, which need to discriminate energy of different frequencies in a single input.

Composition of linear components can be done at the transfer-function level. Suppose there are two components with coefficients in z-transform $H_1(z)$ and $H_2(z)$ respectively, and the input and output of the overall system are $U(z)$ and $Y(z)$ respectively, then in the case of parallel connection as shown in Fig. 2.9(a), we have

$$Y(z) = (H_1(z) + H_2(z))U(z)$$

If the components are connected in a cascaded manner (Figure 2.9(b)), then

$$Y(z) = (H_1(z)H_2(z))U(z)$$



(a) Parallel connection

(b) Cascaded connection

(c) Feedback connection

Figure 2.9: Interconnection of linear components

Finally, Fig. 2.9(c) shows the case where the two components (forward and backward) are connected into a feedback loop. We have the transfer function of the loop as

$$Y(z) = H(z)U(z)$$

where

$$H(z) = \frac{H_1(z)}{1 + H_1(z)H_2(z)}$$

To make a feedback loop feasible for implementation, it is required that the backward component not depend on its present input, i.e., $h_2(0) = 0$, where h_2 is the coefficient sequence of the backward component. The simplest backward component is a one-step time delay unit: $H_2(z) = z^{-1}$, when the output of the forward component is used for feedback directly.

The time and frequency response of a linear system can be easily seen from its transfer function. The time response concerns how fast a system response to changes in its input. A responsive system will have all the poles of its $H(z)$ sufficiently smaller than one in magnitude. Otherwise it would be sluggish if at least one of the poles of its $H(z)$ is close to one in magnitude. Given a sinusoidal input at a certain frequency, a linear

system would have a gain (the ratio between the magnitudes of the output and input signals) and angle (the phase difference between the output and input signals) that are determined solely by (the poles and zeros of) its z-transfer function and the frequency of the signal. Thus the gain and angle of a linear system at all frequencies consist of its frequency response. Because of the important role the poles play in determining the time and frequency response of linear systems, their placement in transfer functions is a very important part in design and analysis of linear systems, especially linear filters [5].

2.5.3 Building Feedback Systems With Predictable Properties in the Toolkit

The results from control theories can be applied in the software feedback toolkit to build feedback systems with predictable theoretical properties. Linear theories [5, 6] can be applied for formal specification of linear feedback components and systems, and automatic generation of linear components. They can also be applied for analysis and prediction of theoretical properties. Nonlinear [14], fuzzy and neural [67] control theories provide general guidelines for the development of complex software feedback systems, such as the methods to decompose a complex system into simpler sub-systems.

Linear systems theories also provide a set of well-understood and well-behaved building blocks, and formal methods of component composition. Example building blocks are digital filters such as the first-order lowpass filter as shown in Fig. 2.3(a), high-pass and band-pass filters, Notch filters [5], Butterworth filters [5] with sharp frequency attenuation, adaptive Kalman filters [19, 39], integrators, differentiators [5], etc. The composition rules shown in Fig. 2.9 help in composing building blocks into more complex components and systems at an abstract transfer-function level.

Since the properties of a linear system are solely determined by its structure and are predictable, another application of the linear control theory is to help make stability and performance statements about linear feedback components, and place guards around the feedback components against the statements made. Though stability is a global property for linear systems, a globally unstable system may still be stable for a specific subset of its input space. For example, the output of an integrator is bounded for all unbiased periodic

input signals. Within the domain in which a linear system is stable, it is still possible that its behavior and performance are desirable within a sub-domain, and undesirable otherwise. In next subsection, we will show that a PLL with only a lowpass filter and a VCO (voltage-controlled oscillator) in its feedback loop is stable, but if the speed of the reference clock is too large, then the maximum phase error could be undesirably large.

With the predictability of linear systems, given a linear feedback component, its current state and input, it is possible to predict if its output would be unbounded, its performance would degrade to an undesirable level, or if its maximum error or time to stabilize to a given error bound would exceed preset limits. With this prediction, we then can guard the stability or performance of the linear component by placing appropriate guards on its input signals, referred to as *input guards*. These input guards on performance or stability have the advantage that they are preventive instead of reactive. They are triggered if the performance is going to degrade, or the component is going to experience instability, and appropriate action can be taken to prevent these undesirable behaviors from happening. These actions include changing the parameters of the working components, or replacing one component with another one that works well in the new situation. On the other hand, for an unpredictable nonlinear software feedback system, a reasonable way to guard performance is to have guards to monitor the performance. The guards will be triggered when the performance has degraded to a specified limit, or the feedback has broken. Then action can be taken to recover from the already-incurred damages.

The advantages of input guards come with disadvantages such as complexity and inaccuracy. An input guard is built based on the knowledge of its feedback component. It effectively contains the model of the component, and thus is complex. An input guard predicts future output based on current state and input, effectively assuming that the input or is predictable. If the input does not happen as predicted, the prediction made by the input guard will be inaccurate, or simply invalid.

2.5.4 Example: a Phase-Lock Loop

In this subsection, we discuss a phase-lock loop (PLL), a linear feedback system example. Through this example, we show how linear feedback systems and components can be

specified by transfer functions, and how properties such as stability and time responses can be analyzed with the transfer functions instead of going through extensive simulation. We also demonstrate how discrete-time linear systems theory can be applied in placing input guards on performance of the PLL.

In a PLL, there is a reference clock and a local clock. The local clock has an adjustable speed. It tracks the phase of the reference clock and locks to it through a feedback loop. PLLs can be found in many applications. One example is FM radio receivers. Each receiver has a PLL in its circuitry. Whenever tuned to an FM radio channel, the PLL in the receiver locks on to the carrier frequency of the radio channel, and the modulated audio signals are then extracted and played out.

The overall structure of the PLL is shown in Fig 2.10. The PLL takes in the speed of the reference clock, and generates a sequence of phase values of the local clock. The reference clock is modeled as an integrator taking a positive input as its speed and produces a sequence of incremental phase values. The local clock is a VCO, with its speed controlled by an input voltage. Thus the VCO can also be modeled as an integrator with the filtered phase difference being its input voltage. The phase difference between the reference and local clocks is detected, and passed to a filter composed of a delay unit, a gain unit, and a first-order lowpass filter. The filtered phase difference is then fed to the VCO to drive the local clock. To make the PLL feasible for implementation, the local clock phase generated by the VCO is delayed for one unit of time before being sent to the phase difference detector. The delay unit is introduced to model the possible latency in the feedback loop.

To study the properties of the PLL, we want to see its output in response to a bounded input reference clock speed. Since the local clock phase value is monotonically incrementing and unbounded, we take the phase difference between the two clocks instead of the local clock phase as the output of the PLL. An unstable PLL may generate an unbounded phase difference, indicating the two clocks are out of synchronization. A phase difference of zero means the reference and local clocks are phase-locked without phase error.

The transfer function of the whole PLL can be inferred from that of its components. The difference functions and transfer functions of the building blocks used in the PLL are listed in Table 2.3, where a, g, and t are constants. The PLL is a cascaded connection

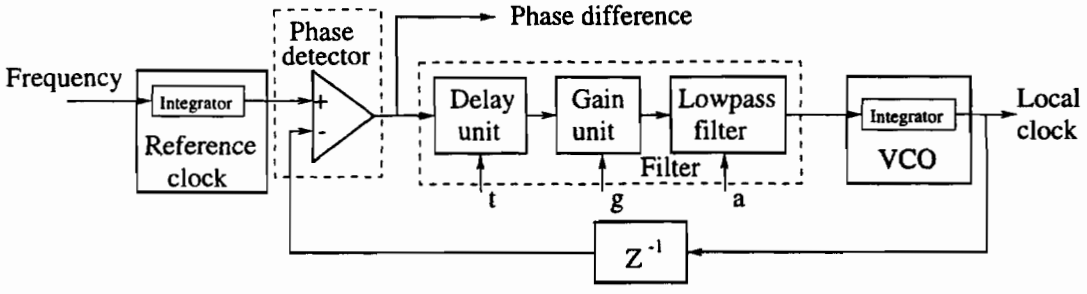


Figure 2.10: The structure of a phase-lock loop

Component	Difference function	Transfer function
Delay unit	$y(k) = u(k-t)$	$H_1(z) = z^{-t}$
Gain unit	$y(k) = gu(k)$	$H_2(z) = g$
Lowpass filter	$y(k) = au(k) + (1-a)y(k-1)$	$H_3(z) = \frac{az}{z+1-a}$
Integrator	$y(k) = y(k-1) + u(k)$	$H_4(z) = \frac{z}{z-1}$

Table 2.3: Transfer functions of the components in the PLL

of an integrator (the reference clock) and a feedback loop. While the feedback loop has a forward part with no component (equal to a unit gain component), and a backward part including all components in the loop except for the phase detector. The forward part of the feedback loop is equivalent to a unit gain, with a transfer function of $H_f(z) = 1$. The backward part of the feedback loop has a transfer function $H_b(z)$ as defined by following formula.

$$\begin{aligned} H_b(z) &= H_1(z) \times H_2(z) \times H_3(z) \times H_4(z) \times z^{-1} \\ &= z^{-t} \times g \times \frac{az}{z+1-a} \times \frac{z}{z-1} \times z^{-1} \\ &= \frac{gaz^{1-t}}{(z+1-a)(z-1)} \end{aligned}$$

The transfer function of the whole PLL, $H(z)$ is defined as:

$$\begin{aligned} H(z) &= H_4(z) \times \frac{H_f(z)}{1 + H_f(z)H_b(z)} \\ &= \frac{z(z+1-a)}{(z+1-a)(z-1) + gaz^{1-t}} \end{aligned}$$

A PLL with a basic configuration has only a VCO in its feedback loop, which is equivalent to having $t = 0$, $g = 1$ and $a = 1$ in its transfer function $H(z)$. Putting these constants into the equation above, we get the transfer function $H(z) = 1$. This $H(z)$

means a basic PLL is equivalent to a unit gain, with its output phase difference always equal to the speed of the reference clock. This PLL is stable, very responsive, and at the same time not smooth at all, since any noise in the reference clock would immediately be reflected in the local clock. There is always a residual phase difference between the reference and local clocks.

When the gain unit is added to the basic configuration ($g \neq 1$), the PLL has a transfer function $H(z) = \frac{z}{z-(1-g)}$ with a single pole $1-g$. This PLL is stable when $\mid 1-g \mid < 1$, i.e., $0 < g < 2$. It is also over-damped [5], with the output phase difference approaching its steady-state value monotonically. The closer g is to 0, to more sluggish the PLL is, meaning the that PLL will react to speed changes in the reference clock more slowly, while the speed changes in the local clock will be smoother.

When the lowpass filter is added to the basic configuration ($0 < a < 1$), the transfer function of the PLL becomes

$$H(z) = \frac{z(z+1-a)}{z^2-(1-a)}$$

This transfer function has two poles $\pm\sqrt{1-a}\,i$, indicating that the PLL is stable and under-damped, thus its output phase difference will approach its steady-state with decaying sinusoidal oscillation. The smaller the coefficient a is, the more sluggish the PLL would be.

However, when one unit delay is introduced to the basic configuration ($t = 1$), the PLL becomes unstable. The transfer function of the PLL is $\frac{z^2}{z^2-z+1}$. It has two poles $\frac{1}{2} \pm \frac{\sqrt{3}}{2}\,i$, the magnitude of both being equal to 1. This result indicates that this PLL with one unit latency in its feedback loop is unstable. The basic PLL with any number of units of latency in its feedback loop is unstable, and the instability problem can be corrected by setting appropriate loop gain (g) and lowpass filtering (a). Analysis of the more complex PLL configurations will not be further discussed in the thesis.

One problem with the PLL configuration above is that there is always a residual phase difference between the reference and local clocks. This residue can be compensated for by adding a stage of integration in the feedback loop. Unfortunately, simply using an integrator would make the PLL unstable. The instability problem could be avoided by the addition of a zero in the transfer function of the integrator used. Figure 2.11 shows

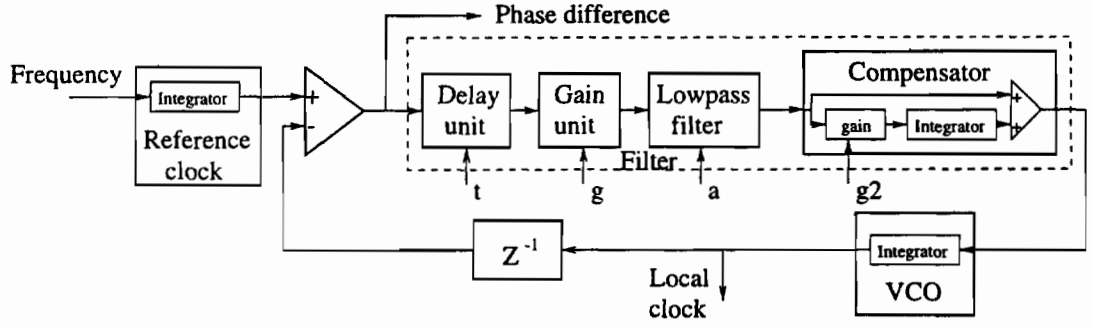


Figure 2.11: Phase-lock loop with compensation for residual phase error

the PLL with a phase-difference residue compensator. The gain unit (with a gain of $g_2 \in (0,1)$) in this compensator controls the effect of integration. The transfer function of the compensator is $H_c(z) = \frac{(1+g_2)z-1}{z-1}$. The PLL with the basic configuration plus this compensator (with $g_2 = 1$) has the following transfer function:

$$\begin{aligned} H(z) &= H_4(z) \times \frac{1}{1 + H_c(z)H_4(z)} \\ &= \frac{z}{z-1} \times \frac{(z-1)^2}{z^2} \\ &= \frac{z-1}{z} \end{aligned}$$

The transfer function above indicates that the PLL is stable. Furthermore, it is equivalent to a simple difference filter having a difference function: $y(k) = u(k) - u(k-1)$. As long as the reference clock keeps a constant speed, the phase difference will be zero, meaning that the local clock will be precisely phase-locked to the reference clock. It is possible to show that for each PLL configuration with the phase-residue compensator, if it is stable, the phase difference will always converge to zero.

Given a PLL with its current state, and a reference clock speed, it is possible to predict the maximum phase difference. With this prediction, we can place an input guard around the PLL guarding its performance. The input guard monitors the input reference clock speed. It is triggered when performance statement violation is predicted. Appropriate actions, such as changing the parameters of the components in the PLL, can then be taken to prevent the violation from actually happening. A general case analysis will not be given in the thesis. In this paragraph, we show a simple example of computing the

maximum phase generated by the PLL when the input reference clock speed is a fixed value S. In this example, the PLL has a lowpass filter with parameter $a = 0.01$, so its transfer function is

$$H(z) = \frac{z(z + (1 - a))}{z^2 - (1 - a)} = \frac{z(z + 0.99)}{z^2 - 0.99}$$

The z-transform of the input signal is $U(z) = S$. So the z-transform of the PLL output phaser error will be:

$$Y(z) = U(z)H(z) = S \times \frac{z(z + 0.99)}{z^2 - 0.99}$$

This indicates that the maximum phase error is proportional to input speed S. Further analysis shows that the maximum phase error is approximately $10.2S$. To guard the PLL against a phase error a statement such as "maximum phase error not exceed E_{max}" is required. The input guard is simply an assertion: $10.2S \leq E_{max}$.

2.6 Discussion

In this chapter, we have proposed a methodology for composing wide-range complex software feedback systems from simple components. In the feedback composition methodology, all software feedback components export a common interface. With this common interface, building blocks can be composed in a uniform way to form composite components. Feedback systems are built hierarchically based on existing building blocks and composite components. To help build wide-range feedback systems, the composition methodology makes guard-based meta-adaptation explicit, and introduces the concept of guarded feedback component. A feedback component has a domain in which it works well, and an associated set of assumptions. A set of guards is placed around a feedback component against its assumptions. A guarded feedback component is re-parameterized or dynamically replugged upon triggering of its guards. A wide-range feedback system can be composed of multiple guarded components, each of which is in effect only when it is applicable.

We also discussed control theories, in particular discrete-time linear systems theory and its application in composing feedback systems with predictable properties. Linear control theories can be applied in formal specification of linear feedback components and

systems, and automatic code-generation of linear components from their specifications. Linear control theories can be applied for analysis and prediction of theoretical properties of linear components and systems. They also provide a set of well-understood and well-behaved building blocks, and formal methods of component composition. Unfortunately, in many realistic cases, linearity is an overly-restrictive requirement. Most practical feedback systems are complex non-linear ones, many of which have multiple linear subsystems applicable under different conditions. Dynamic replugging of guarded feedback components in our feedback composition methodology helps composing simple linear components into complex wide-range feedback systems. Linear systems theories can then help in identifying and placing input guards around linear components. Input guards are preventive instead of reactive. They have the ability to prevent performance degradation or instability from happening, instead of detecting after the undesirable behaviors have already happened. For a wide-range complex nonlinear feedback system, though it may be composed of linear components, linear systems theories become inapplicable to the specification or analysis of the whole feedback system. The use of visualization, simulation and online instrumentation is appropriate and necessary in the analysis and performance tuning of non-linear feedback systems.

In the next Chapter (Chapter 3), we will present a software feedback toolkit prototype that implements the feedback composition methodology proposed in this Chapter. The toolkit prototype implements a component class library that defines the common component interface with base classes, provides facilities for dynamic component replugging, and includes a set of building blocks. It also has GUI-based tools for simulation and online instrumentation of feedback systems. These tools facilitate performance visualization and parameter tuning of feedback systems.

Chapter 3

Implementation of the Software Feedback Toolkit

3.1 Introduction

In this chapter, we present a prototype implementation of the software feedback toolkit. The prototype shows how the feedback composition methodology, including a common component interface and guard-based meta-adaptation operations, can be implemented. To make the feedback toolkit technology readily available to software developers, the toolkit prototype also provides a library of commonly used building blocks, and a set of tools for simulation, and instrumentation of software feedback systems.

The feedback model proposed in Section 2.2 is object oriented. A feedback component is modeled as a message-driven object with a set of input and output ports. As a result, it is natural to have an object-oriented implementation, and to implement feedback components as objects, referred to as feedback objects, instead of sub-routines. Using an object-oriented implementation, a feedback system is implemented as a network of feedback objects. An object implements the functionality of a feedback component. Objects communicate with each other by passing messages through their message ports. Due to the repluggable nature of feedback objects, the structures of feedback systems are dynamic.

To maximize the availability of the software feedback toolkit to software developers, the toolkit prototype is implemented in C++. Among many object-oriented programming languages, C++ [27] is the most pervasive, having a large developer base. In C++, feedback components can be naturally implemented as classes. Each instance of a feedback

component is then a C++ object. Message ports can be implemented by member functions. Passing a message from one component to another can be done by having the former invoke a corresponding member function of the latter with the message as a parameter. One straightforward way to implement common component interfaces in C++ is through derivation of all classes from the same base class. This approach is adopted in the toolkit prototype.

C++ feedback component classes provide two interfaces for composing feedback systems. The first is the "normal" operational interface, that passes feedback messages between components. The other interface, referred to as a meta-interface, is for dynamic component replugging. It supports dynamic component creation, connection, disconnection, and destruction. There are several possible ways to connect feedback components: input ports can be registered with output ports; output ports can be registered with input ports; or there can be a centralized message router that every component uses. In our prototype, feedback-component instances are C++ objects. Message ports are implemented as member functions. Passing a message to a feedback component is done by invoking the associated member function of the feedback object. Due to this message-passing mechanism, in our component base-class, connecting an output port of one component to an input port of another component is implemented by registering the input port with the output port. When a component wants to send a message to one of its output ports, it looks up the registration in the output port, and invokes the relevant member functions of all registered input ports.

The C++ feedback components are passive objects, in the sense that an object is only a data structure with entry points to its member functions. This execution model of passive-objects does not directly support active execution threads. In an application, any thread can invoke a object, and invocations from different threads can be in parallel. In order for feedback objects to work correctly in multi-threaded environments, it is necessary for member functions to explicitly synchronize between accesses from different threads. However, thread synchronization is an implementation detail, and does not contribute very much to the feedback composition methodology. Fortunately, in all the examples and applications given in the thesis, the feedback systems do not need multiple threads.

Support of multi-threaded execution of feedback systems is not implemented in the current version of the toolkit prototype, but is left as part of future work.

In the rest of this chapter, we describe the C++ prototype of the software feedback toolkit. We present the feedback component class library with a component base class, a set of building blocks, and a composite base class for composition of components. The library provides facilities for guarding and dynamic replugging of feedback components. It also implements a set of tools for simulation and instrumentation of feedback systems. The use of the software feedback toolkit in implementing and simulating software feedback systems is demonstrated through two examples: a linear phase-lock loop and a nonlinear flow control feedback system. At the end of this chapter, some issues in the implementation of the software feedback systems and the feedback toolkit are discussed further.

3.2 Software Feedback Component Class Library

3.2.1 Software Feedback Component Base Class

Software feedback components are implemented as C++ classes, all of which are derived from a common base class. Figure 3.1 shows a simplified version of the base class declaration.

The feedback component base class defines a set of public virtual member functions implementing input and output ports as defined in Fig. 2.2, and member functions to initialize or reclaim component instances (objects). In the current version of the prototype, each message flowing through the ports contains a double-precision floating point value. In the implementation of the feedback component library, it turns out that most building blocks need to access most recent input and output messages and current parameters and state values. To simplify the implementation of these building blocks, the base class maintains a set of arrays holding the most recent input or output message values and current values of exported parameters and states. The virtual member functions defined in the base class are as follows:

- **`Feedback()`**

```
class Feedback {
  typedef double FBDataType;
  typedef struct {...} OutputLinkType;
  enum FBPortType {InputPort, OutputPort,
                   ParameterPort, StatePort, ResetPort};
public:
  virtual InputId(int inputPortId, FBDataType message);
  virtual ConnectOutputPortId(int outputPortId,
                              Feedback * feedback_component,
                              FBPortType portType,
                              int portId);
  virtual DisconnectOutputPortId(int outputPortId,
                                 Feedback * feedback_component,
                                 FBPortType portType,
                                 int portId);
  virtual SetParameterId(int parameterPortId, FBDataType message);
  virtual GetParameterId(int parameterPortId, FBDataType &value);
  virtual SetStateId(int statePortId, FBDataType value);
  virtual GetStateId(int statePortId, FBDataType &value);
  virtual Reset();
  virtual ~Feedback();
protected:
  virtual Feedback(int numberOfInputPorts, int numberOfOutputPorts,
                   int numberOfParameterPorts,
                   int numberOfStatePorts);
  FBDataType input [numberOfInputPorts];
  FBDataType output [numberOfOutputPorts];
  FBDataType parameter [numberOfParameterPorts];
  FBDataType state [numberOfStatePorts];
  void Output(int outputPortId);
};
```

Figure 3.1: Definition of software feedback component base class

This function is the constructor of the base class. It initializes instances of the component class. It specifies the number of input, output, parameter and state ports, and allocates memory for the internal arrays holding values for the ports.

- `InputId()`

  This member function implements the input ports of a feedback component. A message is sent to an input port `inputPortId` ($\in$ `[0..numberOfInputPorts-1]`) of a feedback component by invoking this member function. The message will be saved in `input[inputPortId]` and processed. Processing of an input message may or may not generate output messages.

- `SetParameterId()` and `GetParameterId()`

  These member functions implement the parameter ports of a feedback component. An invocation of `SetParameterId()` sends a message to the specified parameter port `parameterPortId` ($\in$ `[0.. numberOfParameterPorts-1]`) to set the corresponding parameter, while `GetParameterId()` reads the current parameter values. A parameter may be associated with a component's internal structure, in which case setting it may cause changes in the structure of the component.

- `SetStateId()` and `GetStateId()`

  These member functions implement the state ports of a feedback component. They allow the exported internal states of feedback components to be accessed. Upon each message sent to a state port `statePortId` ($\in$ `[0..numberOfStatePorts-1]`) by invoking `SetStateId()`, the corresponding internal state is updated with the value contained in the message. `GetStateId()` is invoked to retrieve the current state values.

- `Reset()`

  This member function implements the reset port of a feedback component. It resets the internal state of the feedback component back to its initial state by resetting all entries in array `input[]`, `output[]` and `state[]` to zero, while keeping the current set of parameters unchanged.

- `ConnectOutputPortId()` and `DisconnectOutputPortId()`

  These member functions implement the output ports of a feedback component. They

also facilitate dynamic component composition. `ConnectOutputPortId()` connects the given message input port of the specified feedback component, `<feedbackComponent, portType, portId>` to the identified output port `outputPortId` ($\in$ [0..`numberOfOutputPorts-1`]) of the current component. If the message input port is of of type `ResetPort`, then the `portId` field is ignored. `DisconnectOutputPortId()` disconnects the specified connection. A connection from an output port to a message input port is established by registering the identifier of the message input port in the output port.

- `~Feedback()`

  This is the destructor of the base class. It deallocates all the memory allocated for the internal arrays.

Besides the virtual member functions above, the base class implements a `protected` member function `Output(int outputPortId)`, which sends out the message stored in the output message array entry `output[outputPortId]` to all input ports connected to the output port number `outputPortId`. `Output(int outputPortId)` can be called in the implementation of all feedback components.

The C++ class implementation of a software feedback component inherits the properties and restrictions of the C++ class model. A C++ class is an abstract data type. A C++ object is a data structure with entry points to functions for manipulation of the data itself. The processing of each message by a member function is inherently synchronous. If an output message is generated, it will be sent out by the member function to all the connected message input ports by calling the member functions of the objects corresponding to those input ports. A member function processing an input message returns only after all the output messages have been propagated to whatever components they could reach. So processing of a message may incur a whole chain of calls to member functions of various objects. This implementation of the feedback toolkit also assumes single thread execution. Since most of the time, and for all the examples and applications discussed in the thesis, single threaded implementation is all that is needed. On the other hand, making objects multi-thread safe, and handling all the unlikely synchronization problems

are non-trivial, and are topics of active research efforts.

The class inheritance mechanism in C++ can be applied to build the feedback component class hierarchy. Further base classes for different types of software feedback components are derived from the top level base class shown in Fig. 3.1 and other already defined base classes. At the leaf level are the classes for actual feedback components. It is required that all the feedback component classes have the same public interface as defined by the top level base class, except for class constructors which are specific to individual feedback components. With this common interface, no matter what different behaviors feedback classes have, they can interact with each other in a uniform way.

Derivation of a feedback component class from the top level base class shown in Fig. 3.1 involves redefining some or all the virtual member functions, or adding private data structures or member functions or both. The class constructor is always extended. The member function `Input()` is refined to implement the functionalities of the component. Member functions for accessing parameters and states, and resetting the component may also be refined. Private structures are introduced as needed to hold hidden internal states, and private member functions are defined to facilitate implementation of the functionalities of the component. Fig. 3.2 gives the implementation of the first-order lowpass filter shown in Fig. 2.3(a), as a class derived from the top level base class. The class constructor `LowPassFilter()` specifies that the lowpass filter has one input port, one output port, one parameter port and one state port. It also initializes the filter parameter. For each raw value input to the filter, the member function `InputId()` updates the smoothed value and, by calling an protected member function `Output()`, sends the result to all message input ports connected to the output port of the filter. Member function `SetParameterId()` is redefined to check that the parameter received is within the expected range of $[0, 1.0]$.

3.2.2 Feedback Building Blocks

A set of basic feedback components have been implemented in the software feedback toolkit prototype. These components include filters such as the lowpass filter discussed earlier, median, minimum, maximum, average, integrator and difference filters, as well as a sifter, which only passes values within specified ranges. There are regulators such as

```
class FOLowPassFilter : public Feedback {
private:
public:
  FOLowPassFilter(double p = 1.0);
  int InputId(int id, double value);
  int SetParameterId(int id, double value);
};

FOLowPassFilter::FOLowPassFilter(double para) : Feedback(1, 1, 1, 1)
{
  parameter[0] = para;
  state[0] = 0;
}

int FOLowPassFilter::InputId(int id, double value)
{
  if (id != 0) return -1;
  input[0] = value;
  state[0] = (1 - parameter[0]) * state[0] + parameter[0] * value;
  output[0] = state[0];
  Output();
  return 0;
}

int FOLowPassFilter::SetParameterId(int id, double value)
{
  if (id != 0) return -1;
  if (value < 0) value = 0;
  else if (value > 1.0) value = 1.0;
  parameter[0] = value;
  return 0;
}
```

Figure 3.2: Implementation of a first-order lowpass filter as a C++ class

gain and delay units, biasers, quantizers, limiters, etc. There are also various components such as mergers and triggers, that are for routing messages through the networks of filters and regulators. In the following paragraphs, we explain some of components provided by the toolkit library, including a lowpass filter and a gain unit. A list of feedback building blocks, especially those used in the applications in the thesis, can be found in Appendix B. A complete list of components, including GUI-based components for simulation and instrumentation, are discussed in the toolkit user's manual [7].

Figure 3.2 shows the implementation, as a derivation from the top-level base-class in Fig. 3.1, of the first-order lowpass filter `FOLowPassFilter`. The first-order lowpass filter has a control function shown in Fig. 2.3(a). The constructor, `FOLowPassFilter(double para)`, defines that the lowpass filter has one input port, one output port, one parameter port, and one state port. It also takes a parameter as the time-constant R of the lowpass filter. The parameter-setting function, `SetParameterId(int id, double value)`, checks that the new time-constant parameter is within the range of $[0, 1.0]$. Finally, the member function for the input port, `InputId(int id, double value)`, implements the control function of the lowpass filter. It stores the input value to the input port variable `input[0]`; calculates a new smoothed value based on the input value and the previous smoothed value (`state[0]`) and saves it in its state variable `state[0]`; and sends the new smoothed value out to all down-stream components by storing the value in `output[0]` and invoking `Output()`.

Figure 3.3 gives the implementation of a gain unit as a derivation from the top-level base-class. Suppose a gain unit has a gain of g, its input sequence is $\{u(k)\}$ $(k \geq 0)$, and its output sequence is $\{y(k)\}$ $(k \geq 0)$, then the gain unit can be described by following difference function:

$$y(k) = gu(k)$$

Since the gain unit is very simple, its implementation contains only inline redefinitions of two virtual member functions. The constructor, `Gain(double para = 1.0)`, defines that the gain unit has one input port, one output and one parameter port. It take one input value, *para*, as its gain coefficient. The default gain is a unit of 1. The input port function, `int InputId(int id, double value)`, specifies that each input value is saved

```
class Gain : public Feedback {
public:
  Gain(double para = 1.0) : Feedback(1, 1, 1, 0) {
    parameter[0] = para;
  }
  int InputId(int id, double value) {
    if (id) return -1;
    input[0] = value;
    output[0] = parameter[0] * value;
    Output();
    return 0;
  }
};
```

Figure 3.3: Implementation of a gain unit as a derivation from the base class

in `input[0]` and multiplied by the gain. The result is then stored in `output[0]`, and distributed to all down-stream components.

3.2.3 Composite Feedback Components

A composite feedback component is mainly a composition of simpler components, some of which are dynamically repluggable. It may also need glue code to implement component-specific behaviors such as dynamic component replugging. To implement the common part of composite components, a composite base class, the base class for all composite feedback components, is defined in the feedback toolkit. Fig. 3.4 shows a simplified version of the definition. The composite base class is derived from the top level base class in Fig. 3.1. The structural essence of this composite base class is captured in Fig. 3.5.

The main idea in the composite base class is the introduction of port blocks for all message ports. A port block has one input port and one output port. It simply passes messages through as well as storing the values to an address that is specified during its initialization. In a composite feedback component, each message input or output port is associated with a port block. These port blocks couple the internal structure of a composite component to the outside world, and facilitate dynamic replugging of feedback components. An internal subcomponent can be dynamically connected to a message port

```
class CompositeFeedback : public Feedback {
  typedef double FBDataType;
  class Port : public Feedback {
  public:
    Port(double *store = NULL): Feedback(1,1,0,0) {...}
    int InputId(int inputPortId, FBDataType message) {...}
  };
public:
  InputId(int inputPortId, FBDataType message);
  ConnectOutputPortId(int outputPortId,
                      Feedback * feedback_component,
                      PortType portType,
                      int portId);
  DisconnectOutputPortId(int outputPortId,
                         Feedback * feedback_component,
                         PortType portType,
                         int portId);
  SetParameterId(int parameterPortId, FBDataType message);
  SetStateId(int statePortId, FBDataType message);
  Reset();
  ~CompositeFeedback();
protected:
  CompositeFeedback (int numberOfInputPorts, int numberOfOutputPorts,
                     int numberOfParameterPorts, int numberOfStatePorts);
  Feedback * inputPortBlock [numberOfInputPorts],
           * outputPortBlock [tnumberOfOutputPorts],
           * parameterPortBlock [numberOfParameterPorts],
           * resetPortBlock;
};
```

Figure 3.4: Definition of the composite software feedback component base class

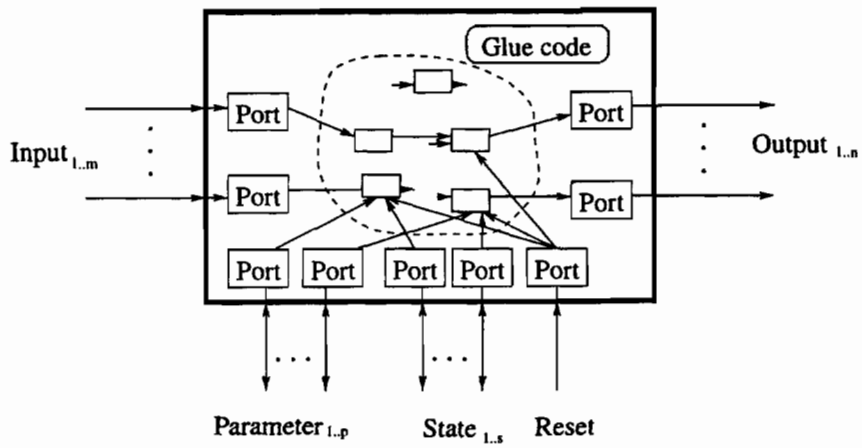


Figure 3.5: Composite software feedback component structure

of the composite component by linking to its corresponding port block, or unplugged by disconnecting the links to the port block. The dynamic replugging of the subcomponents is transparent to the external world. An input port receives input messages from the corresponding input port of the composite component, records the values, and distributes them to connected internal subcomponents. An output block holds all the connections from the corresponding output ports of the composite component to other components, and passes output messages from internal subcomponents to all established connections. A parameter port block passes a parameter message from its corresponding parameter port to all connected subcomponents. Finally, the reset port block distributes reset signals from the reset port of the composite component to all internal subcomponents.

Besides the introduction of port blocks, several public virtual functions in the top level base class are redefined by the composite base class. The object constructor is extended to allocate and initialize all the port blocks, and the destructor is extended to undo the work. `InputId()` directs input messages to the specified input port blocks. `ConnectOutputPortId()` and `DisconnectOutputPortId()` connect or disconnect a link from an output port block to a message input port. `SetParameterId()` directs parameter messages to parameter port blocks. `SetStateId()` directs state messages to state port blocks. Finally, and `Reset()` directs reset signals to the reset port block.

If a composite feedback component has all its functionalities implemented by putting

a set of subcomponents together, its class definition can be a simple derivation from the composite base class by merely extending the class constructor and destructor. The constructor allocates and initializes all the subcomponents, and connects them with each other and with port blocks properly. The destructor then reclaims the internal subcomponents.

Glue code is needed for the class definition of a composite feedback component if it involves operations such as component-specific functionalities or dynamic component replugging, which could not be implemented by mere composition of subcomponents, or retrieving parameters and accessing states distributed among the subcomponents. Currently, glue code exists as extensions to the virtual member functions.

3.2.4 Example: Implementation of a Mean Deviation Filter

As an example of composite feedback component implementation, Fig. 3.6 presents an implementation of the mean deviation filter shown in Fig. 2.6 of Chapter 2. The C++ class definition of this filter is a simple derivation of the composite base class. It uses four existing building blocks provided by the toolkit: two lowpass filters, a merger with two input ports, and a component for computing absolute value. The class definition for the composite filter only extends the constructor and destructor of the composite base class. The extended constructor allocates the four internal subcomponents and links them together and to the appropriate port blocks. The destructor deallocates all the four internal subcomponents. No links need to be removed during destruction, since all the port blocks and the composite components will be deallocated by the destructor. However, this deallocating-without-disconnecting is not the case for dynamic replugging. If a component is dynamically unplugged, all inbound connections from the outside should be removed. If the unplugged component will be reused in the future, all outbound connections should also be removed. Otherwise accesses to already-reclaimed memory segments can occur, resulting in undefined program behavior such as a memory fault.

```
class MeanDeviationFilter : public CompositeFeedback {
public:
  MeanDeviationFilter();
  ~MeanDeviationFilter();
private:
  Feedback *lpfilter1, *lpfilter2, *merger, *abs;
};
MeanDeviationFilter::MeanDeviationFilter()
  : CompositeFeedback(1, 2, 2, 0) {
  // Initialize internal subcomponents
  lpfilter1 = (Feedback *) new FOLowPassFilter;
  lpfilter2 = (Feedback *) new FOLowPassFilter;
  merger = (Feedback *) new Merger(2, "+-");
  abs = (Feedback *) new Abs;
  // Connect subcomponents to each other and to port blocks
  inputPortBlock[0]->ConnectOutputPortId(0, lpfilter1, InputPort, 0);
  inputPortBlock[0]->ConnectOutputPortId(0, merger, InputPort, 0);
  lpfilter1->ConnectOutputPortId(0, outputPortBlock[1], InputPort, 0);
  lpfilter1->ConnectOutputPortId(0, merger, InputPort, 0);
  merger->ConnectOutputPortId(0, abs, InputPort, 0);
  abs->ConnectOutputPortId(0, lpfilter2, InputPort, 0);
  lpfilter2->ConnectOutputPortId(0, outputPortBlock[0], InputPort, 0);
  parameterPortBlock[0]->ConnectOutputPortId(0, lpfilter1,
     ParameterPort, 0);
  parameterPortBlock[1]->ConnectOutputPortId(0, lpfilter2,
     ParameterPort, 0);
  resetPortBlock->ConnectOutputPortId(0, lpfilter1, ResetPort, 0);
  resetPortBlock->ConnectOutputPortId(0, lpfilter2, ResetPort, 0);
}
MeanDeviationFilter::MeanDeviationFilter() {
  delete abs;
  delete lpfilter1;
  delete lpfilter2;
  delete merger;
}
```

Figure 3.6: Implementation of a mean deviation filter as a composite feedback component

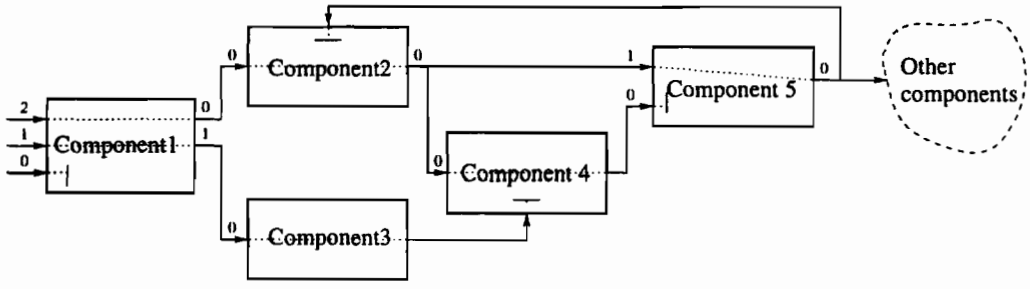


Figure 3.7: An example of directed graph of feedback components

3.2.5 Issues in Composition of Feedback Components

Control and Data Flow in Feedback Graphs

When a set of software feedback components are composed, we effectively get a directed graph of feedback components, referred to as a directed feedback graph. The nodes of the graph are the feedback components. A link from one node to another is established by connecting an output port of the former to an input port of the latter. The directed feedback graph has a dynamic structure, since nodes and links can be created or destroyed dynamically. Figure 3.7 shows an example feedback graph with five components. In this graph, component 1 has three input ports and two output ports. A message sent to its input port 0 does not generate output messages, while a message sent to its input port 1 or 2 generates a message to its output port 0 or 1, respectively. Each of components 2, 3 and 4 has one input port and one output port, and each input message generates one output message. Component 5 has two input ports and one output port. A message received on its input port 0 is absorbed within the component, while a message from input port 1 generates one output message. The parameters of components 2 and 4 are set by the output from components 5 and 3 respectively.

As discussed in Section 3.2.1, the C++ implementation of feedback components is single-threaded and synchronous. With these implementation properties, the control flow in a feedback graph represents a set of feedback component invocation trees, each of which is rooted from a message input port. Suppose in the feedback graph shown in Fig. 3.7, the links from each output port are established in bottom-up order, then there are three component invocation trees associated with the three input ports of component 1. These

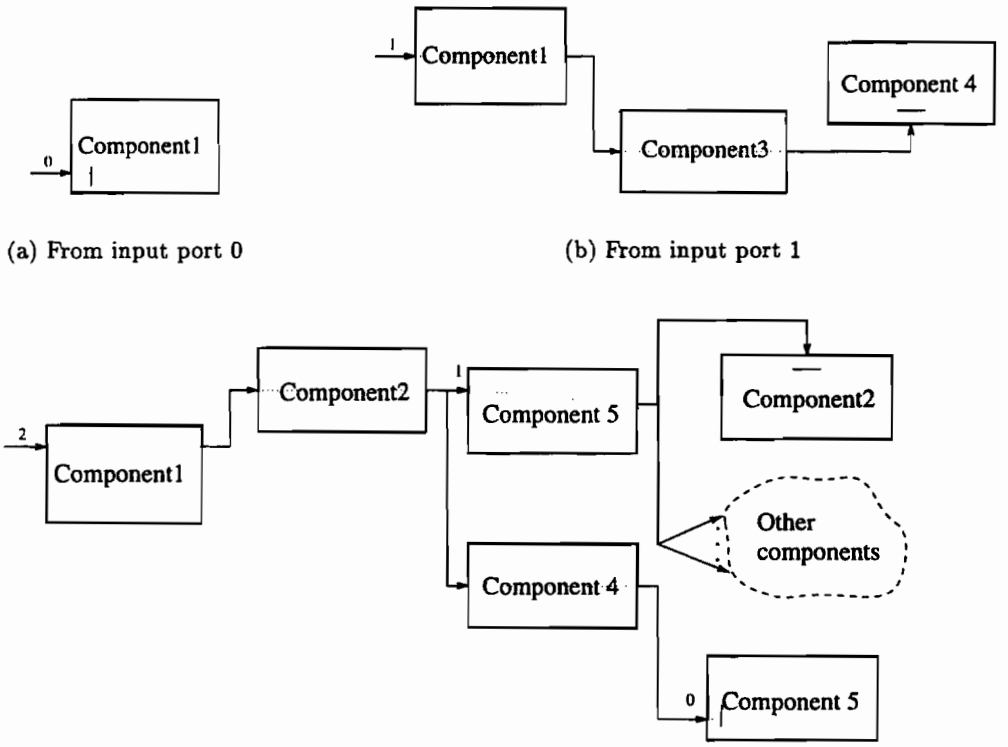


Figure 3.8: Feedback component invocation trees rooted from the input ports of component 1

trees are shown in Fig. 3.8 (a), (b) and (c). In a feedback component invocation tree, each link represents a call to a member function of the destination component. A left-right, bottom-up search of a tree gives the order in which the feedback components are invoked.

Feedback Component Invocation Loop

Directed loops of feedback component invocation may be formed in a feedback graph. Upon receiving a message from the previous node, each node in a loop may generate at least one message to the next one. Loops result in component invocation trees of infinite depth. In the C++ implementation model, a loop is equivalent to a (direct or indirect) recursive function call, and the recursion is formed dynamically by linking the components together. If feedback messages flow in a loop infinitely, what we get is an infinitely long chain of function calls, which causes the application to crash due to limited memory space for function-call stack. Attention should be payed to avoid accidental loops. To reduce the chance of error, though the model does not impose any restriction, it is recommended that a feedback component, upon reset or update of its parameters or states, does not generate messages.

However, all feedback systems by nature work in loops. They bring target systems into stable states through iterations. In simulation, it is usually necessary to build a closed loop of feedback components, and have feedback messages flow inside infinitely, to see if or how the feedback loop converges. In this case, feedback component invocation loops can be prevented by introducing a decoupling component, or decoupler, in the feedback loop. A simple decoupler is the *trigger* described in Appendix B and in the toolkit users manual [7]. It has two input ports, one for data input and the other for a triggering signal input, and one output port. The most recent input message is latched in *trigger*. Whenever a signal is received from its signal input, *trigger* passes the latched input message to its output port. As shown in Fig. 3.9, *trigger* can be incorporated as a node in a feedback loop. This loop is driven by some external triggering signals sent to *trigger*'s signal input. Each triggering signal pushes the feedback loop through a single iteration step, and the result is held in *trigger* until another signal arrives for next step. For example, in both the PLL and flow control feedback simulators to be discussed later

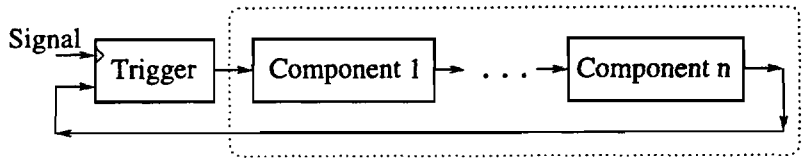


Figure 3.9: Implementation of feedback loops with the toolkit-based feedback components

in this chapter, decouplers, which are used to break the loops, are introduced. In the PLL loop (Section 3.5), the phase difference detector works as a decoupler: it computes and outputs a phase difference whenever the reference clock inputs a phase value. The phase difference is taken by the filter to control the VCO, which computes the next-step phase of the local clock. The next-step phase value is then held until next-step phase values comes from the reference clock. In the flow control feedback loop (Section 3.6), a *trigger* serves as a decoupler.

Limitations Imposed by the Implementation

There are some limitations the single-threaded and synchronous C++ implementation imposes on the kinds of software feedback systems that can be easily implemented. In order to make feedback components asynchronous, explicit programming for (re-)ordering the feedback messages is necessary. If feedback components need to be implemented as separate threads, or need to reside on different hosts, external mechanisms are necessary for implementing inter-process communication and synchronization. To introduce delays in components, we will need to rely on external timers or periodic invocation (polling) of the components to identify time. Fortunately, as demonstrated by the applications in the later chapters in this thesis, even for distributed feedback systems, it is usually the case that the toolkit-based feedback components can be placed within a single thread, and the whole feedback system is implemented with these components plus some simple application-specific mechanisms.

3.3 Implementation of Guard-Based Meta-Adaptation

Guard-based meta-adaptation involves elements such as guarded feedback components, guards, and dynamic component replugging operations. A guarded feedback component needs to be repluggable. All feedback components derived from the base classes in Section 3.2 can readily be used as guarded components, since all of them are already repluggable, and can be dynamically created, plugged in, unplugged, or destroyed. Guards are used to detect events and trigger meta-adaptation. They can either be implemented as feedback components, or simply as glue code in composite components. For dynamic component replugging, feedback components need to be dynamically created and plugged into the active system, or unplugged and destroyed. Since all feedback components are repluggable, dynamic replugging operations are simple, and can be implemented as glue code in composite components that manage repluggable components. In a multi-threaded environment, when replugging operations can happen in parallel with "normal" feedback operations, synchronization needs to be implemented explicitly. In future versions of the feedback toolkit in which feedback components can be synthesized dynamically at run-time, a centralized dynamic component synthesizer-and-replugger may be needed. In a feedback system with guard-based meta-adaptation, during feedback operation, events from the target system are intercepted in an application-specific manner, and are passed to the feedback system through its input or parameter ports, where they are checked by guards and may initiate meta-adaptation in the form of either parameter change or dynamic replugging of feedback components. Events may also trigger exceptions that signal to the application.

3.3.1 Implementation of Guards

In the simplest form of guards, aspects of the internal structure of a composite component are exposed through its parameter ports. The outside world manipulates its internal structure directly by sending messages to its parameter ports. One example is in the PLL simulation to be discussed later in this chapter. The composite filter in the PLL in Section 3.5 has a dynamically repluggable compensator and a corresponding parameter

port. The compensator is unplugged whenever the parameter is set to zero, otherwise it is plugged in.

Guards cal also be implemented by feedback components such as filters. For example, in the receiver side of a UDP network connection without packet reordering problems, the assumption of "no packet loss" can be guarded by a difference filter that takes in the sequence numbers of packets received. Whenever a packet is lost, the output of the difference filter is greater than one, signalling the invalidation of the assumption. The guard for packet-loss feedback in the adaptive packet flow control feedback to be discussed in Chapter 4 is based on this technique.

3.3.2 Dynamic Replugging of Feedback Components

The C++ class implementation of software feedback components provides facilities for dynamic component replugging. Instances of a component class are dynamically initialized and reclaimed by its class constructors and destructors respectively. Links are established or removed through component class member functions `ConnectOutputPortId()` and `DisconnectOutputPortId()`. The parameters and states of a feedback component can be set or saved, by accessing by its parameter and state ports.

Dynamic replugging of guarded feedback components can be performed either eagerly or lazily [15]. A component can be plugged in or unplugged eagerly by its guards each time the guards are triggered. The PLL simulation in Fig. 3.10 shows an example of eager replugging of the compensator component. Eager component replugging involves potential overhead if a component is replugged frequently without being actually invoked. On the other hand, a component may also be replugged lazily. Its guards, when triggered, only set appropriate flags. Each time the composite feedback component containing the repluggable component receives an input message that may invoke it, the flags are checked, and the component is replugged when indicated. The adaptive packet flow control feedback in Chapter 4 contains two feedback policies, the dynamic replugging of both of them is performed lazily. Lazy component replugging involves overhead in processing input messages in order to check relevant flags, and to possibly replug the components. How to perform dynamic replugging of feedback components, either eagerly or lazily, involves

trade-offs between implementation complexity and performance. In the case of multi-threaded implementation, dynamic replugging also involves trade-offs in complexity is synchronization between different operations such as guarding, component replugging and normal operations [15].

One operation that should be avoided is the unplugging of an active feedback component. As discussed previous Section 3.2.5, a feedback component graph may have several component invocation trees, each of which may have one or more component invocation chains. In a feedback component invocation chain, a component in a later stage should not directly unplug another one in an earlier stage. Instead, the component in the later stage should set a flag for lazy replugging later, either at the end of the current invocation, or during the next invocation. For example, Fig. 3.8(b) shows a invocation tree with a chain of three components. In this chain, component 4 should not unplug component 3 by itself, otherwise the application may invoke a feedback component which has been reclaimed, resulting in undefined behavior.

3.3.3 Signalling Exceptions to the Application

One way for applications to receive events (including exceptions) is through callback functions. An application registers callback functions with underlying system for the events it is interested in. Whenever an event happens, all callback functions registered for the event are invoked. Thus the applications are notified, and any necessary reactions can be performed.

The feedback toolkit prototype implements a simple *exception* component for exception signalling through callback functions. As described in the toolkit user's manual [7], an *exception* component takes the address of a callback function during initialization. It has a single input port, and no output or parameter ports. Whenever a message is received by the exception object, the callback function is invoked with the value of the input message as its argument. Other components can be connected to this *exception* components. Whenever an component detects an exception, it sends out a message containing an exception code to the *exception* component.

3.4 Simulation and Instrumentation Tools

The feedback toolkit prototype provides a set of special feedback components for simulation of feedback systems as well as on-line instrumentation in real-life applications. These components include Motif-based graphical user interface (GUI) components such as the parameter setting panel, timer panel, control panel, and oscilloscope, as well as various signal generators and components for file input and output. Though these components have special functionalities, they still bear the same interface as normal feedback components. So interaction between these components and the rest of feedback systems are simple and clean. These components can be readily linked to the feedback systems to be simulated or instrumented directly. This section briefly describes some of the components. A more detailed description can be found in the toolkit user's manual [7].

3.4.1 GUI Components

The parameter setting panel sets parameters of other feedback components and controls the display of other GUI panels. There are two parts in the parameter panel GUI: a set of scales for setting numerical parameters, and a number of normal or toggle buttons for sending signals to or setting on-off parameters of other components. Each parameter corresponds to a pair of input and output ports. A parameter can be set by user actions on the corresponding scale or button as well as upon receiving input messages from its corresponding input port. Each time a parameter is set, an output message is generated to the corresponding output port, and the current parameter value is reflected in the panel (in the case of the scales or the toggle buttons). The parameter panel has a single parameter port, receiving a message from which toggles the display state (shown or hidden) of the panel.

The timer panel generates timing signals, and is useful in simulation. Through its GUI, the user can start, stop or step the timer, as well as set the timing interval.

The control panel is used to set parameter values of other feedback components, to toggle the display state of other GUI components, and to control simulation processes. Figure 3.11 shows the GUI of control panel for the PLL simulator. The control panel is

basically a combination of a parameter setting panel and a timer panel, plus a *reset* button issuing signals to reset other feedback components, and an *exit* button for terminating the whole application. The control panel is the module that initializes the Motif and X run-time environment, and drives the processing of Motif and X events received by all GUI panels. Thus the control panel must be used whenever other Motif-based components are needed.

The oscilloscope panel (scope) displays one or more channels of data in real-time. Figure 3.12 shows the scope panel for the PLL simulator. The user has the choice of either external timing from one of the data channels, or internal timing. The scope has a timer panel for adjusting the internal timing, and a parameter panel for setting the bias and range of all the data channels. The display state of the scope can be toggled by sending messages to its parameter port.

3.4.2 Components for File Input and Output

There are two file I/O components: *InputFromFile* for reading from files, and *OutputToFile* for writing to files. An *InputFromFile* component is initialized with a file name from which double precision floating point numbers are read. It has one input, one output, and one reset port. Upon receiving an input message, the next data item, if available, is read from the file and output to all connected input ports. A message to its reset port rewinds the file pointer to the beginning. An *OutputToFile* is also initialized with a file name. All data received from its input port is written to the file, and a reset signal discards all the data already in the file. The associated data files are closed upon destruction of the *InputFromFile* and *OutputToFile* components.

3.4.3 Components for Signal Generation

The feedback toolkit prototype implements components to generate sequences of signals for simulations. There are components generating sinusoid waves, square waves, random numbers, etc. A signal generation component has one input, one output and a number of parameter ports. Signal generators are triggered by signals from sources such as timer panels. The signal range, and the period in the case of periodic signals, can be set

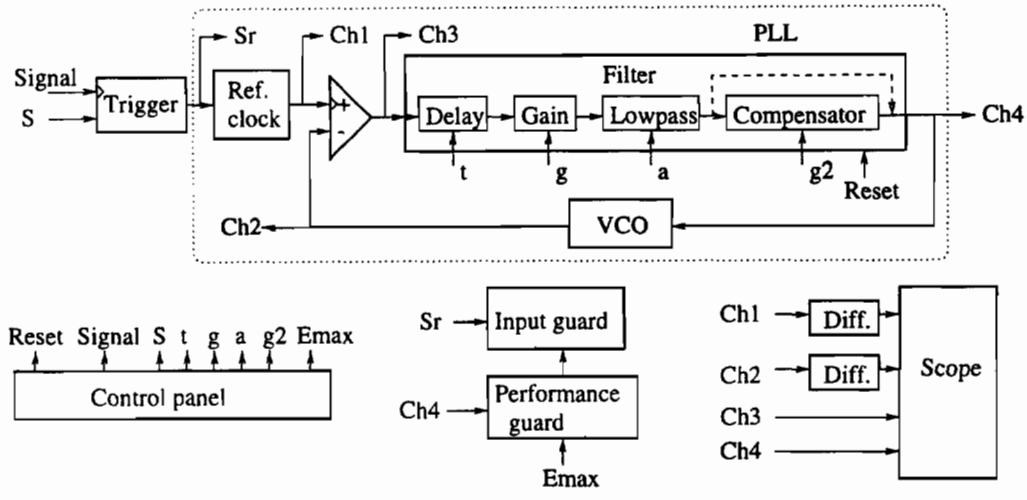


Figure 3.10: Structure of the PLL simulator

dynamically through its parameter ports.

3.5 Example: Phase-Lock Loop Simulation

To demonstrate the application of the software feedback toolkit in composition and simulation of software feedback systems, we have built a simulator of the phase-lock loop discussed in Section 2.5.4. The simulator is implemented using the building blocks provided by the feedback toolkit. Through simulation, the theoretical results reached in Section 2.5.4 are verified, and the dynamics of the PLL with various configurations is visualized.

3.5.1 Simulator Structure

The structure of the PLL simulator is shown in Fig. 3.10. The PLL implemented has the same structure as that shown in Fig. 2.10 of Section 2.5.4. The reference clock is driven by a trigger component, which generates a message containing the clock speed upon each triggering signal. The phase difference detector and unit delay are implemented as a single merger component, with the "+" input as the trigger. All other components in the feedback loop except for the VCO are packed in a single composite filter component.

The phase residual compensator in the composite filter component is dynamically

repluggable. When parameter g_2 of the composite filter is set to zero, this compensator is unplugged. On the other hand, whenever g_2 becomes non-zero, it is plugged into the composite filter.

The input guard on phase error discussed at the end of Section 2.5.4 is implemented in the simulator. This input guard takes the speed of the reference clock S_r as its input, and phase error limit E_{max} as its parameter. When the assertion $10.21S_r \leq E_{max}$ (Section 2.5.4) fails, this input guard is triggered and prints out a warning message. This input guard assumes no delay ($t = 0$), unit gain, $g = 1$, no phase residual compensation ($g_2 = 0$) and strong lowpass filtering ($a = 0.01$). Its triggering is accurate only when the parameters are set as mentioned. For simplicity however, these assumptions are not guarded. For comparison, a similar simple performance guard is also implemented. These guards are disabled when the error limit E_{max} is set to zero, so that they are not triggered during experiments for other purposes.

The scope displays four channels of simulation data in real-time. In Fig. 3.10, from top to bottom, these channels are reference clock speed (calculated by applying a difference filter on the reference clock phase sequence), local clock speed (a difference filter is also used), phase difference between reference and local clocks, and composite filter component output.

The control panel for this PLL simulator is shown in Fig. 3.11. Through the control panel, the user can view and adjust all the parameters of the PLL including reference clock speed, feedback loop delay, gain, phase residual compensator gain, and phase error limit for the guards. The user can also reset the PLL, and control the process of simulation through the control panel.

3.5.2 Simulation Results

The stability results reached in Section 2.5.4 are verified by simulation. The basic configuration, in which the feedback loop effectively consists of only the VCO, has a unit gain. The output phase difference is always the same as the reference clock speed. The addition of the phase-error residual compensator changes the basic configuration into a difference filter, with the output phase difference always being the derivative of the input

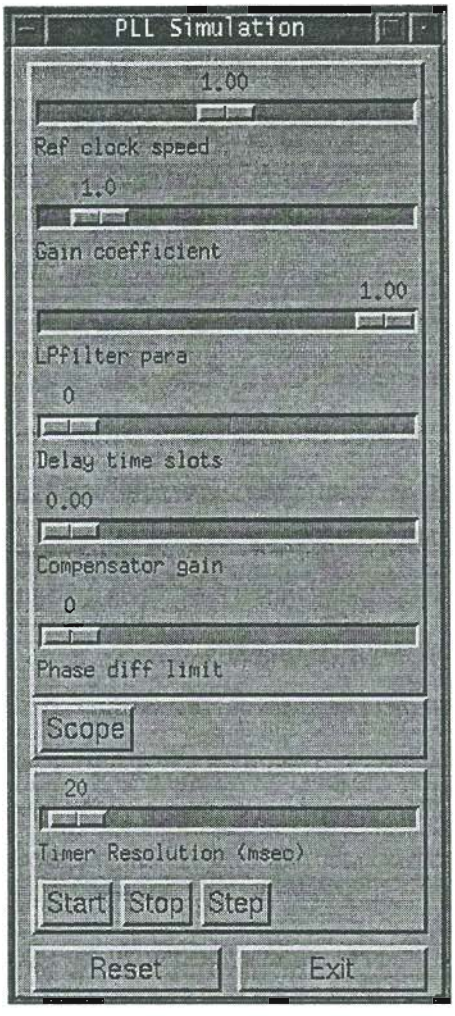
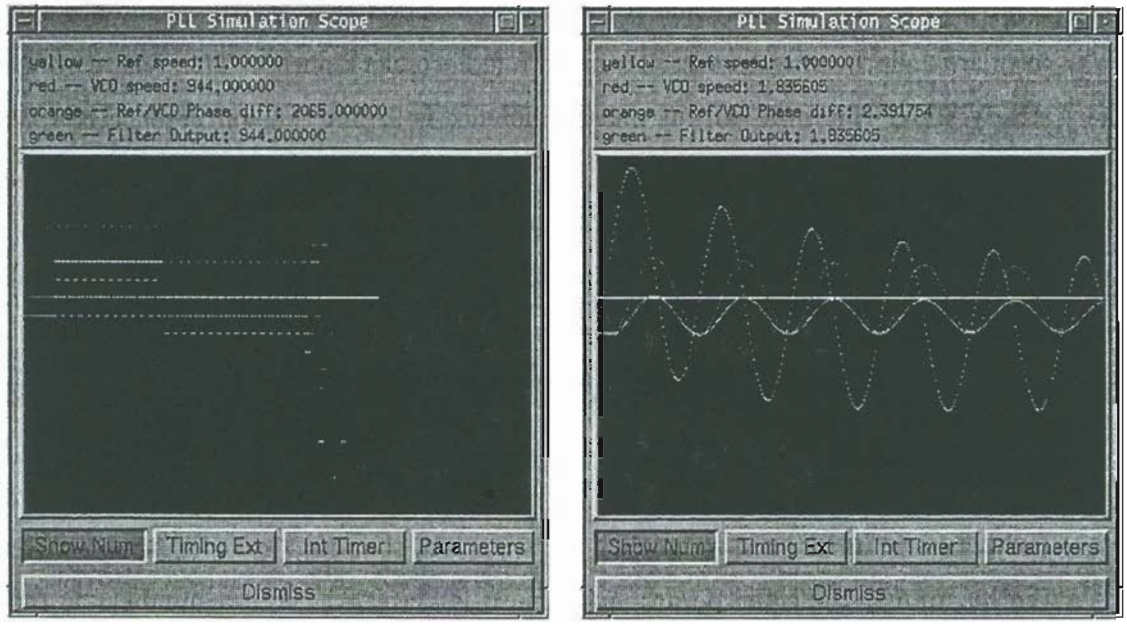


Figure 3.11: Screen dump of the control panel used in the PLL simulator

reference clock speed. Lowpass filtering makes the VCO speed change smooth, with the convergence of the PLL always being accompanied by decaying oscillations. Smaller-than-1 gain smoothes out the VCO's tracking of reference clock changes, and the convergence is monotonic. However, in the steady state, the phase-error residue is proportional to the inverse of the gain, the smaller the gain is, the larger the residue becomes. Adding a non-zero delay in the feedback loop of the basic configuration makes the PLL unstable, with its output phase difference unable to converge.

The simulation also clearly illustrates the dynamics of the PLL with different configurations. Figure 3.12(a) shows a snapshot of the scope for an experiment with an basic



(a) Instability behavior with delay of 1 and 2 and non-constant reference clock speed

(b) Stable PLL with long latency and low gain, lowpass filtering and phase difference compensation

Figure 3.12: Oscilloscope snapshots showing the dynamics of PLLs

PLL except that the feedback loop delay is non-zero. This experiment starts with feedback loop delay $t = 1$ and reference clock speed $S = 1$. The PLL is unstable, but the output phase difference is still a constant equal to S. Later, when S changes to 2, the instability of the PLL shows up as an oscillation in its output. The oscillation is persistent even when S changes back to 1. Finally, when feedback-loop delay t changes to 2, the PLL output quickly goes unbounded. This experiment shows that the phenomena of the instability of linear systems is manifold. An unstable system may generate unbounded output, or persistent oscillation (limit cycles), or it may also generate output that appears to converge.

PLLs with long feedback loop delay can still be made stable by setting other parameters appropriately, though we will not prove this claim formally. Figure 3.12(b) shows the scope snapshot for an experiment, in which the PLL has a long feedback loop delay of $t = 13$. It also has parameters gain $g = 0.1$, lowpass filtering $a = 0.5$, and phase error residual

compensation $g_2 = 0.01$. The scope snapshot shows that there is a latency before the feedback loop begins to react the input reference clock speed. The PLL converges to a steady state where the output phase difference is 0, but the convergence is a very slow process. Simulation shows that when long latency is involved, the gain of the PLL should be kept to a low level, otherwise the PLL becomes unstable. This property may limit the application of this simple PLL in situations where long round trip latency cannot be avoided, e.g., in long-haul networks with many hops or with satellite links.

To demonstrate the effect of the input and performance guards, we set the parameters of the PLL as appropriate ($t = 0$, $g = 1$, $g_2 = 0$ and $a = 0.01$), and enable both guards. The experiment shows that these two guards are either both triggered or neither triggered. Furthermore, in the case when they are triggered, the input guard is always triggered at the beginning, while the performance guard is alway triggered a number of steps later, when the phase error excesses the limit. For example, in one experiment, the reference clock speed is set as $S = 1$ and the phase error limit as $E_{max} = 10$. The input guard is triggered at step 1, while the performance guard is triggered at step 11. The preventive nature of the input guard leaves a chance for the PLL to take measures to avoid the performance degradation problem from occurring.

3.6 Example: Flow-Control Feedback Simulation

Simulation becomes an even more important means in understanding nonlinear software feedback system properties such as stability, performance, and responsiveness, as well as for tuning-up performance. In this section, we demonstrate the simulation of a simple nonlinear feedback system that controls the flow of multimedia data through an idealized pipeline.

3.6.1 The Flow Control Feedback

In client-server style applications for adaptive real-time multimedia streaming, such as the MPEG video player to be discussed in Chapter 6, a typical configuration consists of a set of media pipelines. Each of the pipelines is composed of stages including a server, network

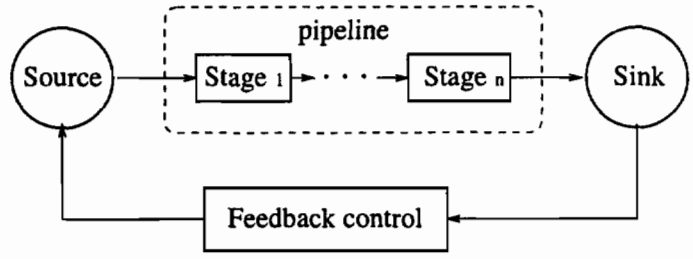


Figure 3.13: Multimedia data pipeline with feedback-based flow control

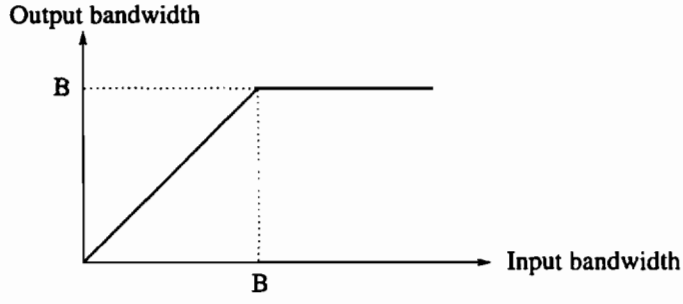


Figure 3.14: A data packet rate model for multimedia data pipelines

links, and a client. Media data flow from the server through stages to the client. The pipeline stages can drop data independently in the case of overload, and no effort is made to retransmit lost data. Feedback-based mechanisms are used to dynamically control the flow, in order to limit packet-loss ratio while sufficiently exploiting the available bandwidth.

Figure 3.13 shows a simple and idealized model of the scenario above. The data pipeline is modeled as a piece-wise linear function, as shown in Fig. 3.14, with two parameters: nominal bandwidth B and latency D. If at time (step) k, the rate of data input to the pipeline is $u(k)$ (≥ 0), and the output data rate is $y(k)$, then the model function is defined as below, where P is the name of the function. For an input data rate up to the nominal bandwidth, the pipeline drops no data. Otherwise the pipeline only outputs data at rate of B. We assume that all excessive data are dropped randomly and evenly.

$$y(k) = P(u(k-D)) = \begin{cases} u(k-D) & \text{if } u(k-D) \leq B \\ B & \text{otherwise} \end{cases}$$

The feedback mechanism controls the data input rate of the pipeline to limit data loss.

It simply tries to keep the input data rate higher than the output rate, with the difference close to a rate-incremental constant Δ. The current pipeline output rate $y(k)$ is sampled, optionally processed by a lowpass filter, then used to compute the next-step pipeline input rate. The feedback control can be modeled with the following equation:

$$u(k+1) = Lowpass_a(y(k)) + \Delta$$

3.6.2 Simulator Structure

The simulator for the feedback-based flow control loop is shown in Fig. 3.15. The feedback loop is composed of two composite components, one for the pipeline and the other for the feedback policy, as well as a *trigger*. The pipeline is in turn a composition of a delay unit and a component computing $y(k) = P(u(k))$. It receives and processes next-step input data rates. After D units of delay, both the input and output rates are generated by the pipeline. The composite component implementing the feedback policy contains a lowpass filter and biaser. The trigger holds next-step input rate values, and sends them to the pipeline upon each signal from the timer of the control panel. It is reset at the beginning of each simulation to an initial state where its data output prior to the first input is always zero. Thus all simulations start with an initial input data rate of zero. All four parameters, pipeline nominal bandwidth B, delay time D, lowpass filter parameter α and rate constant Δ, can be adjusted through a control panel. The scope panel displays three channels of data, the current pipeline output rate, the input rate that generates the current output rate, and next-step input rate.

3.6.3 Simulation Results

It is intuitive that the flow control feedback without pipeline latency would be stable, with a steady state in which the input and output data rates are close to the nominal bandwidth of the data pipeline. Simulations suggest that this intuition is actually valid. In one experiment, we configure the feedback loop so that no pipeline latency nor lowpass filtering exists (basic configuration). We also use the rate incremental $\Delta = 1.0$ and start with a pipeline nominal bandwidth $B = 30$. Figure 3.16(a) shows a snapshot of the scope for this experiment. The snapshot indicates that the pipeline output linearly increases

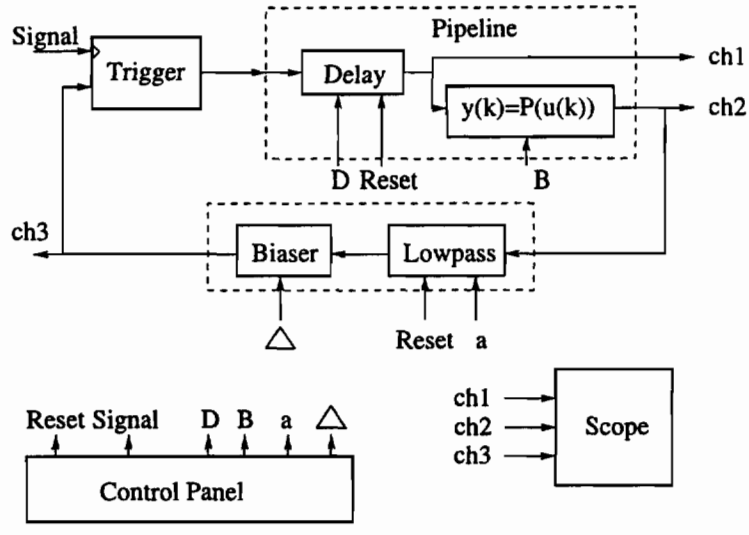
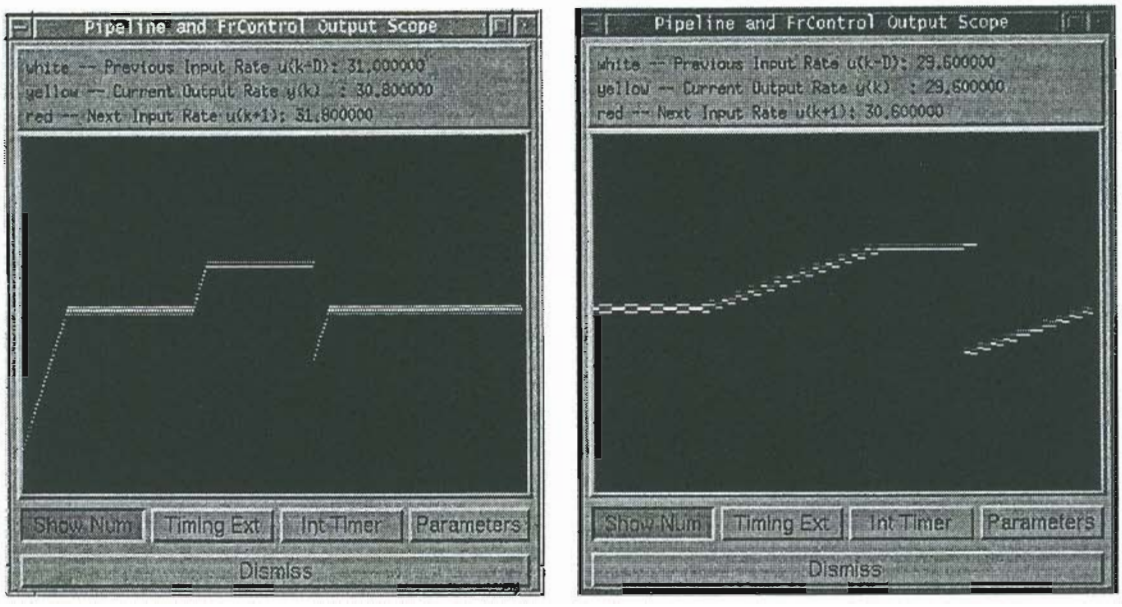


Figure 3.15: Structure of the flow control feedback simulator

to 30 and then oscillates. Then we set the nominal bandwidth B to a higher level, and the feedback linearly traces the change, this time the limit cycle is missing. Later B is set back to around 30 (30.8 in the experiment), and packet drop occurs, but the feedback quickly reacts to yield the new bandwidth again. Generally, the flow control feedback with the basic configuration (no pipeline latency nor lowpass filtering) is stable, most of the time with limit cycles oscillating around the nominal bandwidth of the pipeline. The magnitude of the limit cycles is determined by the data rate incremental Δ. The feedback is responsive in tracking changes in nominal pipeline bandwidth.

The simulation also reveals several effects of the lowpass filter to the flow control feedback, and shows that the longer the lowpass time constant (by setting a smaller parameter value a), the stronger the effects are. Firstly, no matter what time constant it has, lowpass filtering effectively curbs the limit cycles. Secondly, lowpass filtering tends to make the feedback less responsive, taking longer to track changes in nominal (available) bandwidth. The result is that when available bandwidth increases, more time is needed to detect and make use of it. On the other hand, if the available bandwidth decreases, more time is also needed for the pipeline input rate to follow, causing increased data loss. In practice, the same sluggishness makes the feedback more robust in the case when the variation in measurement of pipeline available bandwidth is caused by measurement noise



(a) Basic configuration: no pipeline latency nor lowpass filtering

(b) Basic configuration except for long pipeline latency

Figure 3.16: Oscilloscope snapshots showing the dynamics of the flow control feedback

instead of actual changes in nominal bandwidth.

One concern about the latency in the pipeline is that it may cause oscillation or instability, especially if not enough damping is applied. The simulation demonstrates that the instability problem does not exist. The latency increases the likelihood of limit cycles, and makes the feedback more sluggish. But with all the parameter combinations simulated, the feedback always eventually converges to the neighborhood of the pipeline nominal bandwidth B, sometimes with limit cycles. Figure 3.16(b) shows the result of an experiment in which the flow control has the same basic configuration as that for the experiment shown in Fig. 3.16(a), except that the latency is set to 8. The first segment is the steady state for $B = 30$. Then B is set to around 40, and the feedback reacts to the change more slowly. Then B is set back to around 30 again (29.6 in the experiment), and it can be seen that it takes several steps before the feedback reacts to the change, causing packet drops for a longer period (8 steps in this experiment).

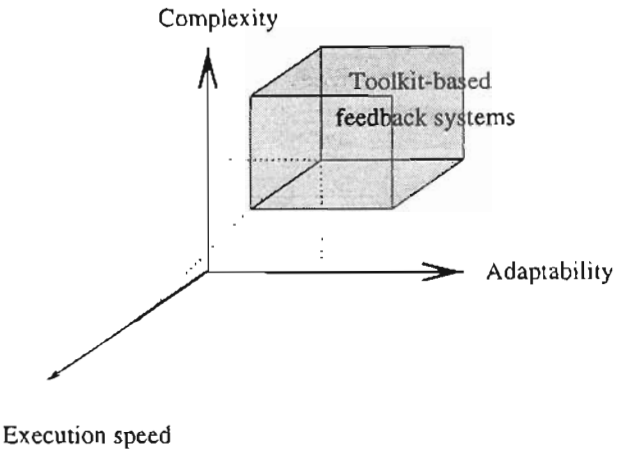


Figure 3.17: Feedback system design space and where the toolkit fits

3.7 Discussion

3.7.1 Feedback System Implementation with the Toolkit

We envision a design space of software feedback systems as shown in Fig. 3.17. The design space has several dimensions, such as complexity, adaptability, and execution speed. Different feedback systems have different requirements and stay at different positions in the design space. Feedbacks such as those used in Synthesis for adaptive scheduling [32] are invoked very frequently, and need to be highly optimized for execution speed. Some feedbacks, such as TCP flow control [23], are complex. They need complex and fine-tuned filters and control laws to ensure system stability and satisfactory performance. There are also feedback systems that need to be highly adaptive. They need to work well in multiple situations, some of which may not even be known before their development. The adaptive packet rate control feedback in Chapter 4 falls into this category. In many cases, complexity and adaptability are closely related. A feedback system is complex mainly because it needs to handle multiple cases, and the need for a feedback to work in multiple situations makes it complex.

Like other software systems, feedback systems can be implemented in one of several ways as shown in Fig. 3.18. Each way has pros and cons in criteria such as modularity, software reuse, extensibility, execution speed, etc. First, a feedback system could be

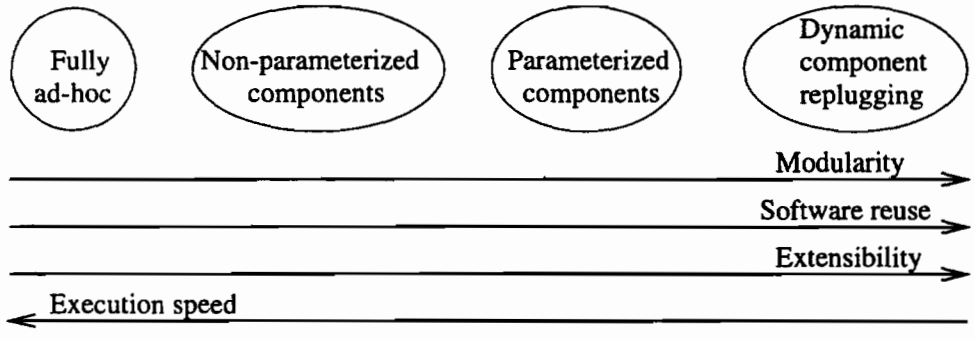


Figure 3.18: Software feedback implementation approaches and their properties

implemented from scratch as a single ad-hoc monolithic module. For complex feedback systems, this approach has known deficiencies: it is neither modular nor extensible, provides no software reuse, and is hard to implement and maintain. But if implemented right, their execution performance can be highly optimized. Second, existing non-parameterized components could be used. Modularity and software reuse are enhanced. The code may not be optimized for execution performance, but powerful compile-time optimization techniques are readily applicable. Third, existing parameterized components could be used in building complex feedback systems. Modularity, software reuse and extensibility are pushed further. But the potential difficulty in compile-time optimization increases, since the adjustable parameters are not amenable to constant folding, upon which many optimization techniques are based. Finally, dynamic replugging of feedback components can be incorporated. With this last approach, modularity and software reuse are maintained, and extensibility is maximized. But optimization for execution speed is even harder, because the interaction between feedback components can be dynamic, and is stored in data structures. Compile-time optimization cannot help in reducing the overhead of crossing the boundaries of dynamically-linked modules.

The proposed feedback toolkit approach helps the design and implementation of complex wide-range software feedback systems, as indicated in Fig. 3.17 and Fig. 3.18. It provides methodologies and tools to decompose a complex feedback system into guarded subsystems, and then implement the feedback system by composition of existing building blocks. It also provides tools for simulation and instrumentation. In the case when high execution performance is critical, a feedback system can still be designed, implemented

PLL filter version	Original	Manually optimized
Execution speed (μs/call)	16.57	0.91

Table 3.1: Execution speed of the original and manually optimized PLL component on an 100MHz Pentium Notebook

and tuned with the toolkit, and then re-implemented for actual application for higher performance.

3.7.2 Execution Performance of Toolkit-Based Feedback Systems

The feedback toolkit helps the development of complex wide-range feedback systems, possibly in exchange of some overhead in execution speed of the feedback systems implemented. The toolkit adopts a general feedback component model. Feedback components may have adjustable parameters. The interactions between components are dynamic, and stored in data structures (C++ objects). Modern compiler technologies are powerful enough to fold constants in expressions and program control flows, and to remove cross-module interpretation overhead when constants are involved. But not enough run-time optimization efforts have been made, such that programs with dynamic values and interactions can be effectively incrementally optimize incrementally at run-time. Thus, even though the feedback components are implemented as C++ classes, there is still much room to improve their execution performance through compile-time and run-time optimizations.

As an example to show the execution overhead of the toolkit approach, we re-implemented the PLL filter component in the PLL simulator (shown in Fig. 3.10) as a single class, derived from the `Feedback` base class. The re-implemented PLL filter component has a zero-delay, unit gain, lowpass filter parameter 0.1, and a unit gain in its compensator. Efforts are paid to optimize the execution speed of the re-implemented PLL component. Table 3.1 shows a comparison of the execution times of the original (the one used in the PLL simulator in Section 3.5) and re-implemented PLL components on a 100MHz Pentium notebook running LINUX. This table shows that the manual optimization improves the execution speed more than 20 times. Different experiments on different platforms may yield different numbers, but we expect that the speed difference will be of similar magnitude.

On the other hand, the toolkit-based implementation of feedback systems unveils new opportunities of dynamic run-time optimization. When a variable keeps the same value for a long time, optimization similar to compile-time constant folding can be applied. If the dynamic interaction between modules is stable, execution speed can be increased through cross-module optimization (similar to compile-time function inlining) to eliminate interpretation overhead involved in crossing of module boundaries, or even to merge modules. Run-time optimization is much more complicated than the compile-time version, since dynamic code generation and replugging are involved. Especially in the case when the variable or interaction may change, the optimized code needs to be guarded, and unplugged when the assumptions on which the optimized code is based are no longer valid. One important part of the Synthetix project [44] is to explore run-time optimization opportunities with a technique called optimistic incremental specialization. There are also other projects, such as the Scout project [36], on dynamic code generation and run-time optimization. But these research efforts are mainly focused on run-time constant folding within individual modules. Run-time cross-module optimization has not been addressed except in special contexts such as network protocol stacks [36]. Aggressive run-time optimization of the toolkit-based software feedback systems is part of our future research work [43].

3.7.3 Alternative Toolkit Implementation Approaches

The software feedback components in the toolkit prototype are implemented as single-threaded C++ objects. An alternative is a multi-threaded implementation, which can be achieved by making all the feedback component C++ classes and objects multi-thread safe. In a multi-threaded environment, a feedback component can be accessed by more than one thread. Its ports become inherently asynchronous, and it becomes necessary to explicitly ensure the synchronization among the access by different threads, and between dynamic component replugging and normal feedback message processing. Synchronization and concurrency control is a difficult issue, especially when dynamic replugging of components is involved. In Synthetix, a method consisting of a set of primitives has been proposed to link operating system code dynamically [15].

The software feedback components implemented as C++ objects are passive objects. A component is a data structure associated with some member functions for manipulation, and the various ports are represented by the member functions. Alternatively, feedback components could be implemented as active objects. Each feedback component is a thread of execution, or a set of threads (especially for composite components). Components are composed through inter-process communication channels. This implementation model is also called communicating sequential process (CSP) [25]. An interesting parallel programming language OCCAM [25] was designed based on CSP, and has been implemented on Transputer networks [64]. This concept of active threads maps more directly than the passive object to our feedback component model. But unfortunately, for various reasons, the CSP programming paradigm has not become as popular as C/C++. As a result an CSP-based implementation of the software feedback toolkit would be less pervasive.

Chapter 4

Adaptive Packet-Rate Control Base on the Feedback Toolkit

In this chapter, we demonstrate how the toolkit can be applied in the development of wide-range feedback systems with a non-trivial example: a feedback system for adaptive real-time packet-rate control. Following the guidelines from the toolkit, the packet-rate control feedback is composed of two guarded component policies through guard-based meta-adaptation. The whole feedback system is then implemented as a hierarchical composition of feedback components, most of which come from the toolkit class library. Finally, the simulation and instrumentation tools are used to visualize the effects of the feedback and to tune its parameters. We also briefly analyze the feedback policies for the interaction between multiple packet transmission sessions sharing the same network link.

4.1 Introduction

A typical scenario for adaptive real-time distributed multimedia applications, as shown in Fig. 4.1, has a media server and a media client connected through a data packet network such as the Internet. The media server either fetches compressed and stored media data, or captures and compresses a live media source. It packs the compressed media data into packets, and paces them out to the network. The client receives data packets from the network, and decompresses and renders them in real time. Due to the real-time requirements, packets lost in the network or other stages of the media pipeline from the server to the client are not retransmitted. The resources in the server, client, and especially in the network, are shared by all types of applications, and their availability

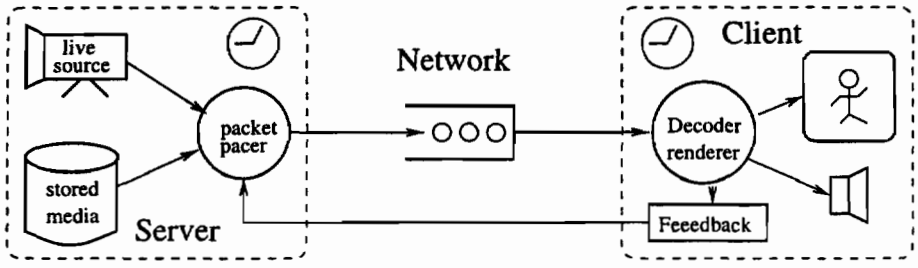


Figure 4.1: A scenario of real-time distributed multimedia applications

changes dynamically. To ensure that the application adapts to the changes, feedback is used to continuously monitor the presentation quality observed at the client, and adjust the sending packet rate at the server accordingly.

In this chapter, we focus on a client-side rate-based feedback that adapts the rate at which the server sends packets based on changes in network conditions. The client continuously monitors metrics such as received packet rate and packet transmission latency (from the server to the client), and determines the future packe-rate sent from the sender. This next-step rate is then sent to the server as a limit on the rate at which future packets are sent. The goal of the feedback is to maximize the utilization of network bandwidth while avoiding congestion.

A packet network connection consists of a number of network links connected through routers and switches, each of which receives and stores packets from the previous link, and forwards them to the next one. Such a network connection can roughly be modeled as a pipe with parameters such as bandwidth, transmission latency and (aggregate) internal buffer size. When network congestion occurs, the internal buffer fills up, causing increased latency, and eventual packet-loss.

Based on the ratio of bandwidth and buffer size, network connections may show different symptoms upon congestion. A connection composed of fast links such as Ethernet or ATM has a large bandwidth-to-buffer-size ratio that congestion will cause the internal buffer to overflow before an increase in transmission latency becomes significant and easily observed. Such a network connection has the congestion symptom of packet-loss, and can be referred to as a *lightly-buffered network connection.* On the other hand, a network

connection such as a PPP link has a much smaller bandwidth-to-buffer-size ratio. For example, the PPP server at OGI's CSE department provides more than $50K$ bytes of buffer for each $28.8Kbps$ PPP link. For a network connection with the above PPP link, in the presence of congestion, the latency would increase from millisecond level up to 15 seconds before packets are dropped by the PPP server. This significant increase in latency can be easily detected and packet-loss can be avoided with appropriated flow control. This later type of network connection is referred to as a *heavily-buffered network connection.* Some other network connections may have a mid-ranged bandwidth-to-buffer-size ratio, and show both symptoms simultaneously of congestion.

In order for the multimedia applications shown in Fig. 4.1 to work well with all types of network connections, the packet-rate feedback needs to react to both network congestion symptoms. Following the methodologies in the software feedback toolkit, we propose the feedback as a composition of two repluggable guarded component policies: packet-loss feedback and latency feedback. The packet-loss feedback policy responds to packet loss, and adjusts the server packet sending rate so that the packet-loss rate is maintained at a preset level. The latency feedback policy reacts upon detection of a significant increase in packet transmission latency, and controls the packet sending rate to keep the latency at a specified level. When the network is congested, if packet loss is detected, then the packet-loss feedback policy is activated. If significant increase in transmission latency is detected, then the latency feedback policy is activated. In the case where both feedback policies are active, the lower of the packet rates generated by the two component policies is used by the media server. When no feedback policy is active, no limit is imposed on the rate at which the server sends packets.

In the rest of this chapter, we first present the architecture of the packet-rate feedback control, including the two feedback policies for lightly-buffered and heavily-buffered network connections, and meta-adaptation of the feedback upon various events. Then we describe the implementation of the feedback system, in which the two feedback policies are implemented as repluggable guarded feedback components. Upon detection of the respective congestion symptoms, the policies are dynamically plugged into the feedback system through meta-adaptation actions. Finally we show some experimental results, and

briefly analyze of the interaction between pakcet transmission sessions using either of the feedback policies.

4.2 Packet-Rate-Control Feedback Architecture

4.2.1 Feedback Policy for Lightly-Buffered Network Connections

We assume that a lightly-buffered network connection can be modeled as shown in Fig. 3.14, with a nominal (available) bandwidth B. When the server sends packets at a rate no more than B, no packet is dropped. Otherwise, the receive packet rate remains at B and all excessive packets are dropped.

The packet-loss feedback policy keeps the server sending packet at a rate close to the nominal bandwidth B by maintaining a preset rate of packet-loss. The policy is simple and similar to the flow-control feedback discussed in Section 3.6. It traces the network available bandwidth B through iterations. At each step, the client measures the current client receive packet rate, and generates a next-step server send rate as the measured received rate plus a constant rate-incremental coefficient.

Suppose at step k, the estimation of the current client receive rate is $\hat{\mu}_k$, and the packet-rate incremental coefficient is Δ, then the next-step server sending rate λ_{k+1} is:

$$\lambda_{k+1} = \hat{\mu}_k + \Delta \tag{4.1}$$

4.2.2 Feedback Policy for Heavily-Buffered Network Connections

The goal of the latency feedback policy is to maintain the overall network buffer-fill level (or buffering latency)[1] close to a preset target value. The buffering latency of a packet can be measured as the transmission latency of the packet less the minimum transmission latency of the connection.[2] The feedback also maintains the target buffer-fill level through

[1]The buffer-fill level is the number of packets buffered inside the network. It can be measured by buffering latency — the time a packet spends in the network buffers. So buffer-fill level and buffering latency are used interchangeably.

[2]The latency of a packet transmitted when the network connection is otherwise quiet. The minimum latency of a packet consists of the processing time in the hosts, switches and routers, plus the propagation latency on the network links.

iterations. At each step, the current buffer-fill level as well as client-received packet rate are measured, and the next-step server sending rate is calculated and sent to the server.

Suppose the preset target buffer-fill level is F, and the sampling interval (the length of each feedback iteration step) is T.[3] Also suppose that at step k, the server sending rate is λ_k (enforced since the beginning of step k), the client received rate estimation is $\hat{\mu}_k$, and the buffer-fill level estimation is $\hat{\gamma}_k$ (both estimated at the end of step k), then the goal of the latency feedback policy is to come up with a next-step server sending rate λ_{k+1}, with the hope that at the end of step $k+1$, the buffer-fill level reaches the target F:

$$\begin{aligned} \hat{\gamma}_{k+1} &= \hat{\gamma}_k + (\lambda_{k+1} - \hat{\mu}_{k+1})T \\ &= F \end{aligned}$$

From the equation above, the following formula for the next-step server sending rate λ_{k+1} can be deduced:

$$\lambda_{k+1} = \hat{\mu}_{k+1} + \frac{F - \hat{\gamma}_k}{T}$$

This formula is not feasible, since the next-step server sending rate λ_{k+1} depends on the next-step client received rate $\hat{\mu}_{k+1}$, which is not available yet at the beginning of step $k+1$. However, the changes in available available bandwidth are usually gradual. It is reasonable to assume that $\hat{\mu}_k$ is a good approximation of $\hat{\mu}_{k+1}$. With this assumption, we get the following control law for the latency feedback policy:

$$\lambda_{k+1} = \hat{\mu}_k + \frac{F - \hat{\gamma}_k}{T}$$

In practice, a constant K $(0 < K \leq 1)$ may introduced, and a limit R may be set on how much λ_{k+1} can deviate from μ_k. Also it does not make sense to have a negative server sending rate. Application of all these modifications results in following practical control law:

$$\lambda_{k+1} = max(\hat{\mu}_k + min(max(K\frac{F - \hat{\gamma}_k}{T}, -R), R), 0) \tag{4.2}$$

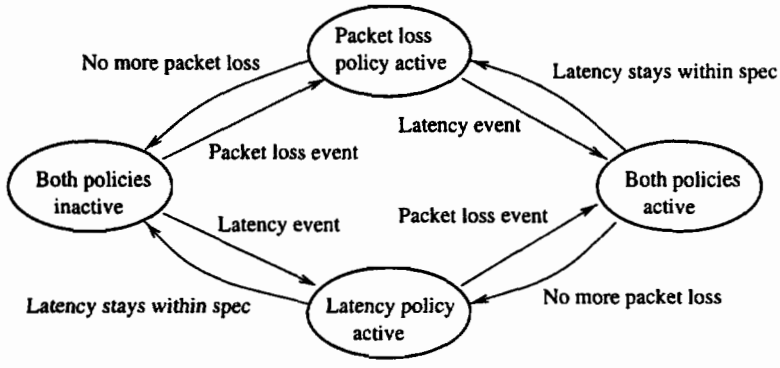


Figure 4.2: Meta-adaptation of the packet rate feedback: the states and their transition

4.2.3 Events and Meta-Adaptation

Several events associated with the network connection congestion conditions are guarded by the packet-rate control feedback. When a guarded event happens and triggers a guard, the feedback performs meta-adaptation actions to activate or deactivate the two feedback policies. The following are the four events to be guarded:

1. Packet loss — A packet-loss event happens whenever a gap between the packet currently received and the previous one is detected.
2. No more packet loss — This event is the expiration of a packet-loss timer. The timer is initialized with a given duration, and is reset upon each packet-loss event. If it is not reset within the specified duration, it expires, indicating that packets are no longer being dropped.
3. Latency higher than specification (latency event) — This latency event is raised upon receiving a packet if the current buffering latency is greater than the specified target value.
4. Latency stays within specification — This event is the expiration of a latency timer. Similarly to the packet-loss timer, the latency timer is initialized with a duration, reset upon each latency event, and expires if not reset within the given duration.

[3]To simplify the problem, we assume that T is significantly larger than the packet transmission latency, thus the latter can be ignored in the design and analysis of the latency feedback.

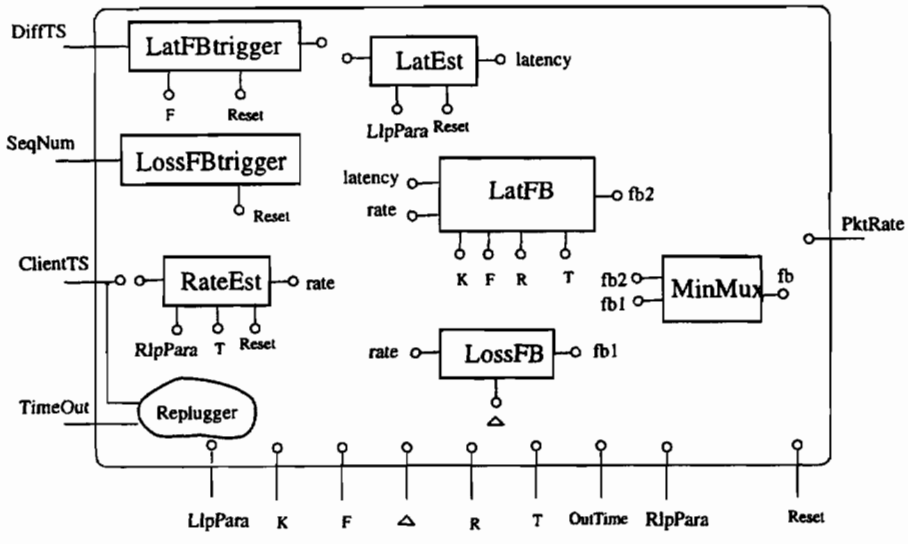


Figure 4.3: Packet rate control feedback overall structure, where floating components are dynamically-repluggable

Depending on whether the network connection is congested and the symptoms detected, the packet rate feedback is in one of the following four states: (1) both policies inactive; (2) packet-loss feedback policy active; (3) latency feedback policy active; and (4) both policies active. Transitions between these states are triggered by the four events above, as shown in Fig. 4.2.

4.3 Implementation of the Feedback System with the Feedback Toolkit

4.3.1 Overall Structure

With an overall structure shown in Fig. 4.3, the packet-rate control feedback is implemented as a composite feedback component. It has a number of sub-components, including feedback policies, packet rate and latency estimators, and policy triggers (guards). The un-connected components, including the components of the two feedback policies and the two estimators are repluggable. Replugging of these components depends on the congestion symptoms as detected by the triggers (guards), and is performed by the replugger.

The following list of components are contained in the packet-rate feedback. Most of

them will be discussed further in the following subsections.

1. *LatFBtrigger* – Component for detection of increase in packet-transmission latency and triggering of latency-feedback policy (guard).
2. *LossFBtrigger* – Component for detection of packet-loss and triggering of packet-loss-feedback policy (guard).
3. *LatEst* – Buffer-fill level (packet-buffering latency) estimator. It estimates latency caused by buffering in the network connection.
4. *RateEst* – Client-received packet-rate estimator.
5. *LatFB* – Packet-latency feedback-policy component.
6. *LossFB* – Packet-loss feedback-policy component.
7. *MinMux* – A multiplexer that outputs the smaller of the two input values. This component is used to select the next-step server-sending packet-rate when both the latency and packet-loss feedback policies are active.
8. *Replugger* – Glue code for dynamic replugging of the feedback-policy-related components.

The packet-rate feedback system has following four input ports.

1. $DiffTS$ – The difference between the client and server timestamps associated with packets. To detect packet-loss or transmission latency, every data packet carries an incremental sequence number and a server timestamp (the server time when the packet was sent). It is assumed that the clocks of the server and the client run at the same speed, so that the latency estimation can be based on the difference between server and client timestamps. Upon arrival of each packet at the client, a client timestamp is recorded, and the difference between it and the server timestamp carried in the packet is sent to this input. The feedback monitors this input to determine the replugging of the components related to the latency feedback policy, and uses the input sequence to estimate the buffering latency.
2. *SeqNum* – Sequence number. Upon receipt of each packet, the sequence number the packet carries is sent to this input. The feedback monitors this input to detect packet-loss, and triggers replugging of the packet-loss feedback policy related

components.

3. *ClientTS* – Client timestamp. For each packet received by the client, its arrival time is fed to this input port. The client timestamp sequence is used to estimate the client-received packet-rate, as well as for invoking the replugger for synchronous component replugging.
4. *TimeOut* – Time out signal. The client generates a time-out signal if it does not receive a packet in a sufficiently long period. This signal is used by the replugger for replugging of the various components.

The packet-rate feedback has a single output port *PktRate* for the next-step server-sending packet-rate.

In order to provide the applications the flexibility in tuning the packet-rate feedback to suit their specific environments, the feedback system exports the following parameters. For each component plugged into the feedback system, its parameter ports are connected to that of the packet-rate feedback system with the same names.

1. *LlpPara* – Parameter of the lowpass filter in the latency estimator component; $0 < LlpPara \leq 1$.
2. K – Constant coefficient of the latency feedback policy; $0 < K \leq 1$.
3. F – Target buffering latency (buffer-fill level in terms of seconds) of the latency feedback policy.
4. R – Limit on packet-rate adjustment range for the latency-feedback policy.
5. Δ – Packet rate-incremental coefficient for the packet-loss-feedback policy.
6. *OutTime* – Timeout time for feedback policy unplugging. The components of a feedback policy are unplugged if the associated congestion symptom is not detected in a period of time given through this parameter port. In our implementation of the packet-rate feedback, the same parameter *OutTime* is used for both policies. Alternatively, in a different implementation, the two policies to have separate and different parameters.
7. T – Period length of a feedback iteration step.

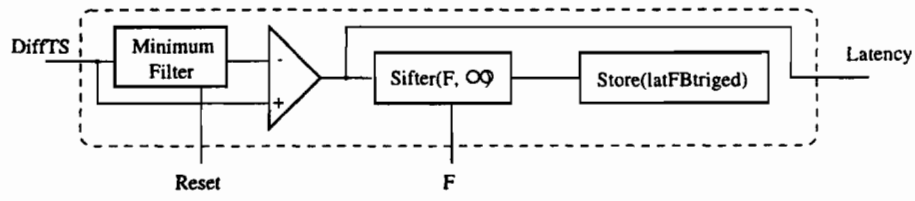


Figure 4.4: Latency-increase-detection component of the packet-rate feedback

8. *RlpPara* – Parameter of the lowpass filter in the client-received packet-rate estimator, $0 < RlpPara \leq 1$.

4.3.2 Detection of the Increase in Network Buffering Latency

The latency-increase-detection and latency-feedback-policy trigger component *LatFBtrigger* is a composite feedback component composed of several building blocks from the feedback toolkit component library. It has a structure shown in Fig. 4.4. *LatFBtrigger* inputs a sequence of timestamp difference values, and calculates the minimum latency. Then it subtracts the minimum latency from the raw latency (input value) to get an estimation of the buffering latency. If this buffering latency is greater than the target buffering latency F, then a flag $latFBtriged$ is set, signalling the replugger for possible plugging of the latency-feedback-policy-related components. The resulting buffering latency is also output for latency estimation by the latency estimator *LatEst*.

Assuming that the server and client clocks run at the same speed, even though the two clocks may have a (constant) phase difference, the estimation of the buffering latency, as output by *LatFBtrigger*, is still valid. For each packet received, its timestamp difference has the following four components: (1) server-client clock phase difference, (2) network-connection minimum transmission-latency, (3) network buffering latency, and (4) timestamp sampling noise. Assuming that the client and server are highly responsive, then factor (4) can be ignored. For a typical network connection, the first two factors are constant. Since we assume that the phase of the clocks in the client and server might not be synchronized, it is not easy to measure each of the first two factors individually. Fortunately, the timestamp difference of a packet transmitted through an otherwise quiet network connection is the sum of the first two factors. Furthermore, this measurement can

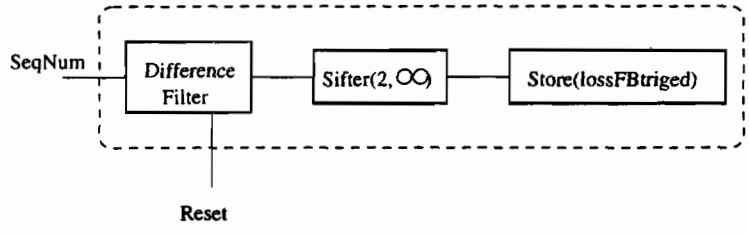


Figure 4.5: Packet-loss-detection component of the packet-rate feedback

be approximated by the minimum of the whole sequence of the timestamp difference measurements. Thus *LatFBtrigger* is able to produce an estimation of the buffering latency by subtracting the minimum latency from the raw timestamp difference.

4.3.3 Detection of Packet Loss

The component for detection of packet-loss and triggering of packet-loss-feedback-policy is also a composite feedback component. Its structure is shown in Fig. 4.5. It is composed of three building blocks. The *difference filter* outputs the gap between the current and the previous packets. The *sifter* passes the gap only when it is greater than or equal to 2, which indicates that one or more packets have been lost. Upon receiving this packet-loss signal, the *store* component sets a flag *lossFBtriged*, informing the *Replugger* of the congestion condition.

4.3.4 Estimation of Client Receive Packet Rate

The client-received packet-rate estimator *RateEst* is shown in Fig. 4.6. For each packet received, it computes the time elapsed between the previous and the current packets, inverts it to get a raw packet-rate, and applies a lowpass filter to get an average packet-rate. Finally, a building block *TimeGate* is used to control when to enable the output of the estimations.

The period of iteration steps of the packet-rate feedback is controlled by *TimeGate*. At the beginning of each step, *TimeGate* is disabled, while the packet-rate and buffer-fill level are being estimated. These estimation processes last until the *TimeGate* is enabled after a period of time specified through its parameter T. At this moment, the packet-rate and

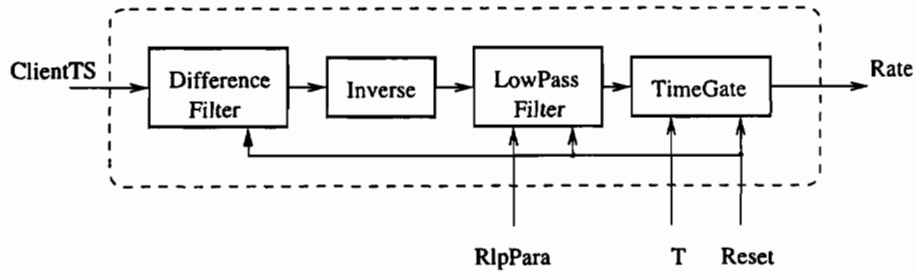


Figure 4.6: Packet-rate estimator of the packet-rate feedback

buffer-fill-latency estimations are passed to the two feedback policy components, which generate a next-step server-sending packet-rate.

4.3.5 Estimation of Buffering Latency

The buffering-latency estimator *LatEst* is simply a lowpass filter from the feedback toolkit component library. It inputs a sequence of raw buffering-latency estimations, filters out the transient noise, and generates a sequence of smoothed estimations.

It should be noted that the output of the buffering latency estimator does not actually reflect the current buffer-fill level. Instead, it reflects the buffer-fill level when the recently received packet was buffered by the network. Depending on the recent server sending packet rate, the current buffer-fill level may be higher or lower than the estimate. Fortunately, the latency-feedback policy should still be stable if the feedback step interval is much larger than the buffering latency, though a rigorous proof will not be given in the thesis.

4.3.6 Packet-Loss-Feedback Policy Component

The packet-loss-feedback policy component *LossFB* inputs a packet-rate estimation, and applies the packet-loss-feedback policy control law, as represented by Equation 4.1, to generate a next-step server-sending packet-rate.

4.3.7 Latency-Feedback Policy Component

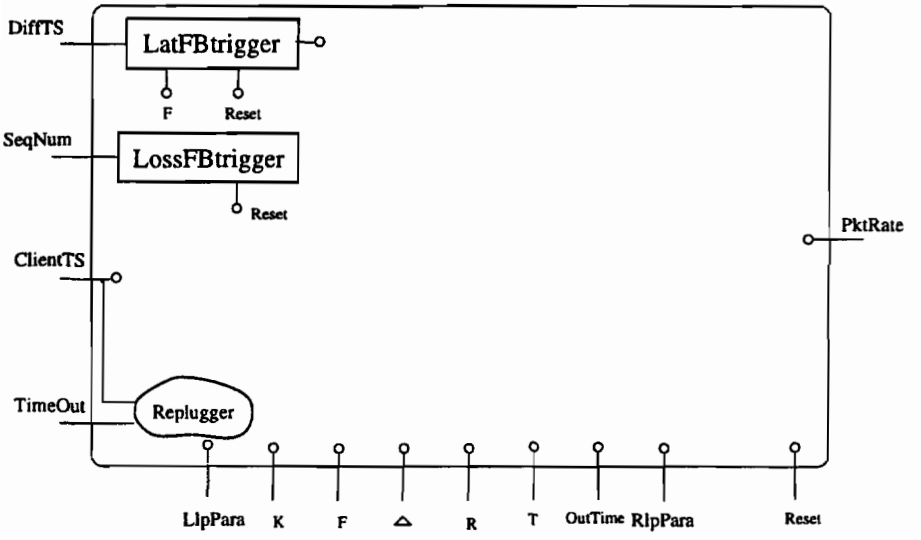
The latency-feedback policy component *LatFB* takes two inputs: the buffering-latency estimation and packet-rate estimation. When a rate estimation is available, *LatFB* converts the buffering latency (*seconds*) into buffer-fill level (*packets*) by multiplying it by the packet-rate estimation. It also converts the target buffering latency F into a target buffer-fill level in terms of number-of-packets. *LatFB* then applies the latency-feedback-policy control law, as represented by Equation 4.2, to calculate a next-step server-sending packet-rate.

4.3.8 Packet-Rate-Feedback States and Dynamic Policy Replugging

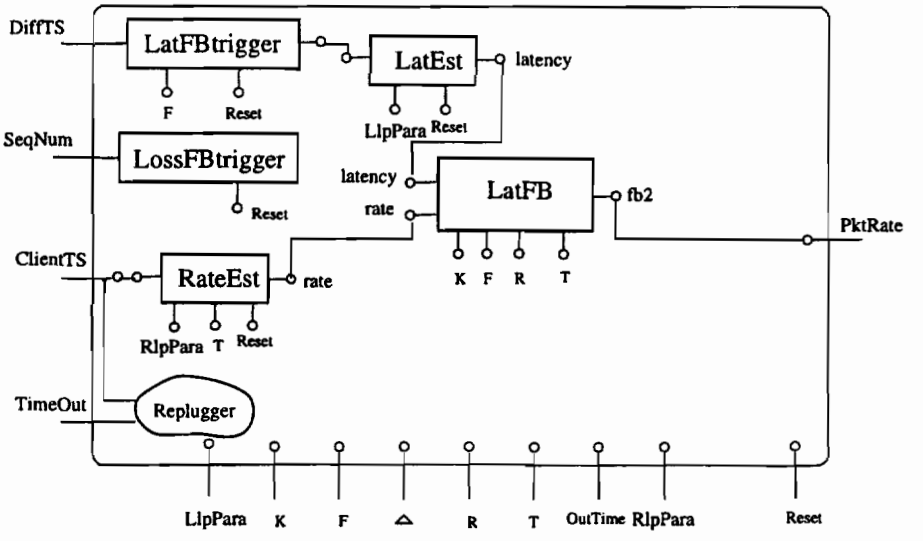
Depending on whether the feedback policies are plugged in or not, the packet-rate feedback system is in one of the following four states, as indicated by two flags $LatFBon$ and $lossFBon$. These states correspond to the ones shown in Fig. 4.2.

- $LatFBon = 0$, $lossFBon = 0$ – Neither of the two feedback policies is plugged in. Figure 4.7(a) shows the components active in this state.
- $LatFBon = 1$, $lossFBon = 0$ – The latency feedback policy is plugged in. Figure 4.7(b) shows the components active in this state.
- $LatFBon = 0$, $lossFBon = 1$ – The packet-loss feedback policy is plugged in. Figure 4.8(a) shows the components active in this state.
- $LatFBon = 1$, $lossFBon = 1$ – Both the packet-loss and latency feedback policies are plugged. As shown in Fig. 4.8(b), in this state, all component are plugged in and active.

The *replugger* performs dynamic feedback-policy replugging. It maintains four flags $latFBtriged$, $lossFBtriged$, $latFBon$ and $lossFBon$. Taking the latency feedback policy as an example, $latFBtriged$ indicates whether it is triggered, and $latFBon$ indicates if it is currently plugged in. When the *replugger* is invoked upon receiving a signal from the input ports $ClientTS$ or $TimeOut$, if a feedback policy has been triggered, then the *replugger* tries to plug in the components of the policy (if not plugged yet) and associates the current timestamp with the triggered policy. On the other hand, if a plugged-in policy has not

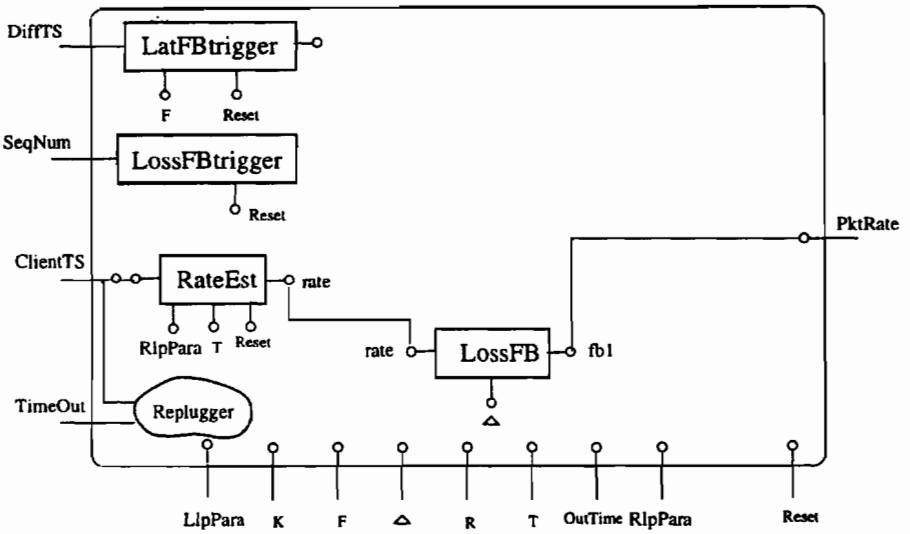


(a) No feedback policy plugged in

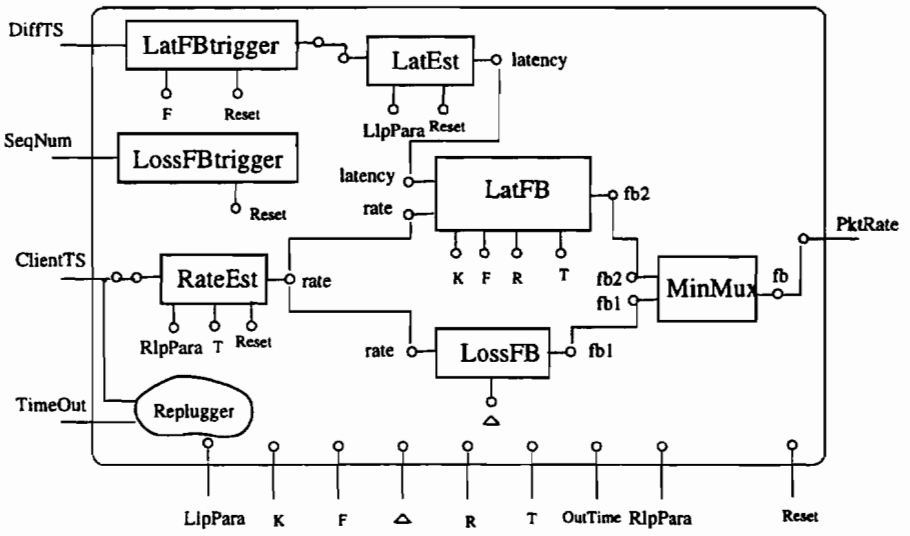


(b) Latency feedback policy plugged in

Figure 4.7: The states of the packet-rate feedback system (part 1)



(a) Packet-loss feedback policy plugged in



(b) Both feedback policies plugged in

Figure 4.8: The states of the packet-rate feedback system (part 2)

been triggered for a period of time specified by the parameter *OutTime*, the timer expires, and the policy is unplugged. At the end of each invocation of the *replugger*, *latFBtriged* and *lossFBtriged* are cleared, so further triggering of the two feedback policies can be detected.

4.4 Experimental Results

To test the packet-rate feedback system developed in this chapter, a test program has been built, and experiments carried out. The test program has two components: a server (sender) and a client (receiver). Upon request from the user at the client, a session between the server and the client, with a TCP control channel and UDP data channel, is established, and dummy UDP packets of a specified size are transmitted from the server to the client in real time. The packet-rate feedback at the client monitors the packets received, and triggers the feedback policies to adjust the rate at which the server sends packets. Through a client-side GUI based on the control panel from the feedback toolkit class library, the user can view and set the packet size, the maximum packet-rate, as well as all the parameters used by the rate-control feedback. The user can also enable or disable the feedback. Statistics on the packets received by the client, such as raw and minimum latency, packet intervals, and gaps can be viewed on a scope panel also from the toolkit component library.

Experiments have been performed with two machines at the CSE Department of OGI: *lemond* (an HP RA-RISC 9000 running HPUX) as the server and *anquetil* (a Pentium notebook running LINUX) as the client. The configuration is shown in Fig. 4.9. Two types of network links are tested: $28.8Kpbs$ PPP and $2Mbps$ WaveLAN. The PPP is a typical heavily-buffered network, introducing more than 15 seconds of latency before any packets are dropped, while the WaveLAN is a typical lightly-buffered network. In the PPP configuration, two Ethernet hops are also involved, but due to their $10Mbps$ bandwidth and millisecond level latency, their effect on the experiments can be ignored.

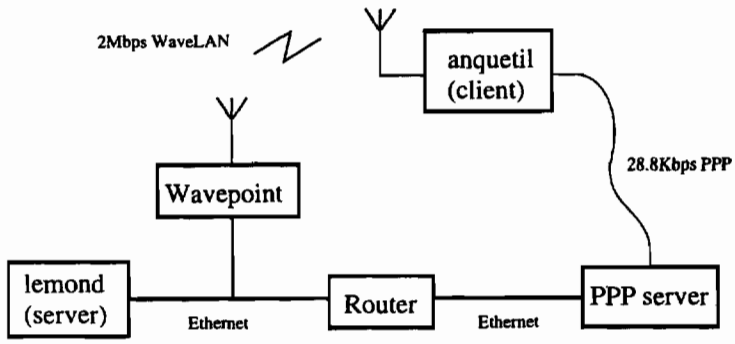


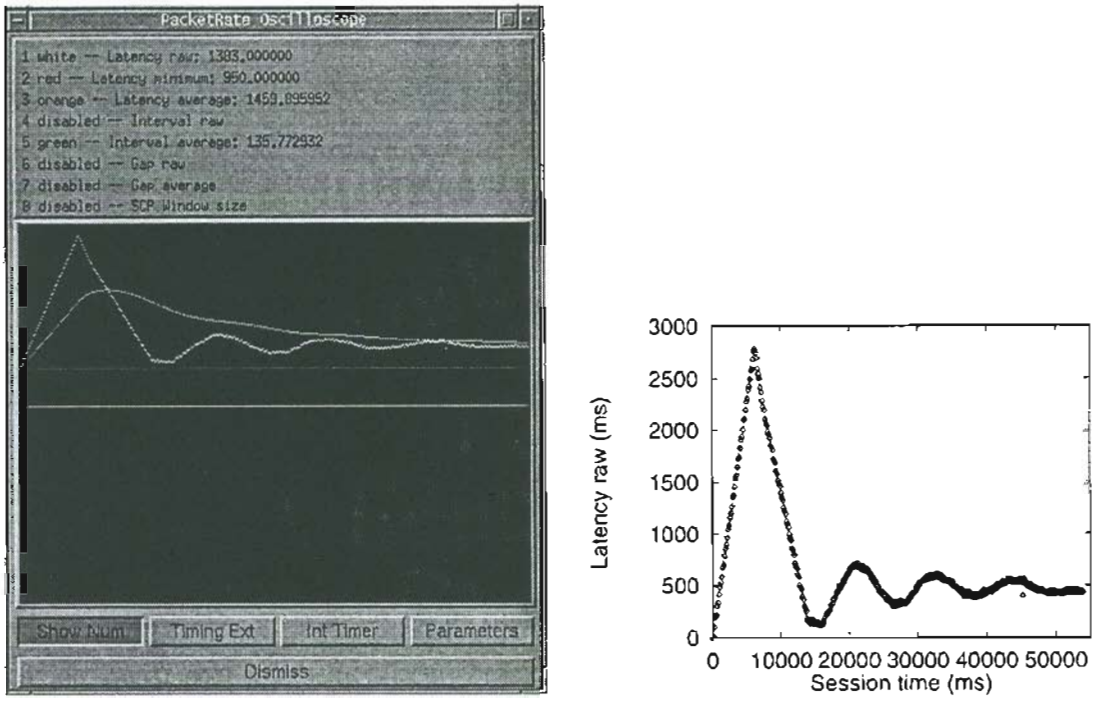
Figure 4.9: Configuration for packet-rate feedback experiments

4.4.1 Experiments Over PPP

A set of experiments have been performed over the $28.8Kbps$ PPP network link. In these experiments, packet size is set to 400 bytes, and the minimum interval at 75 milliseconds (equivalent to a maximum rate of 13.3 packets/second, or $42.7Kbps$). The following feedback parameter settings are used: $LlpPara = 0.2$, $K = 0.94$, $F = 0.4$ second, $\Delta = 1.0$ packets/second, $OutTime = 60$ seconds, $T = 2.0$ seconds, and $RlpPara = 0.01$. Values of most of the parameters are set through cycles of trial-and-tune.

Figure 4.10(a) shows a snapshot of the scope panel of an experiment with a single session through the PPP link. This snapshot contains the statistics of the initial phase of the session. All the four data lines shown in the scope are in units of milliseconds. Among the two straight lines, the lower one is for the smoothed interval between packets received, and the upper one for the minimum latency. The rough curve is for the raw packet latency (the difference between client and server timestamps), and the smoother one for the smoothed packet latency. The buffering-latency measurements over time are also plotted in Fig. 4.10(b). The experiment demonstrates that when congestion in the heavily-buffered PPP network link causes an excessive increase in buffering latency, the latency feedback policy is triggered, and, through iterations, brings the buffering latency to the neighborhood of the target value. As shown in Fig. 4.10(a), at the end, the raw latency is $1383ms$, and the minimum latency is $950ms$, so the buffering latency is $1383 - 950 = 433ms$, close to the target $F = 0.4$second.

To observe how multiple sessions with the latency feedback policy share the same PPP



(a) Oscilloscope snapshot

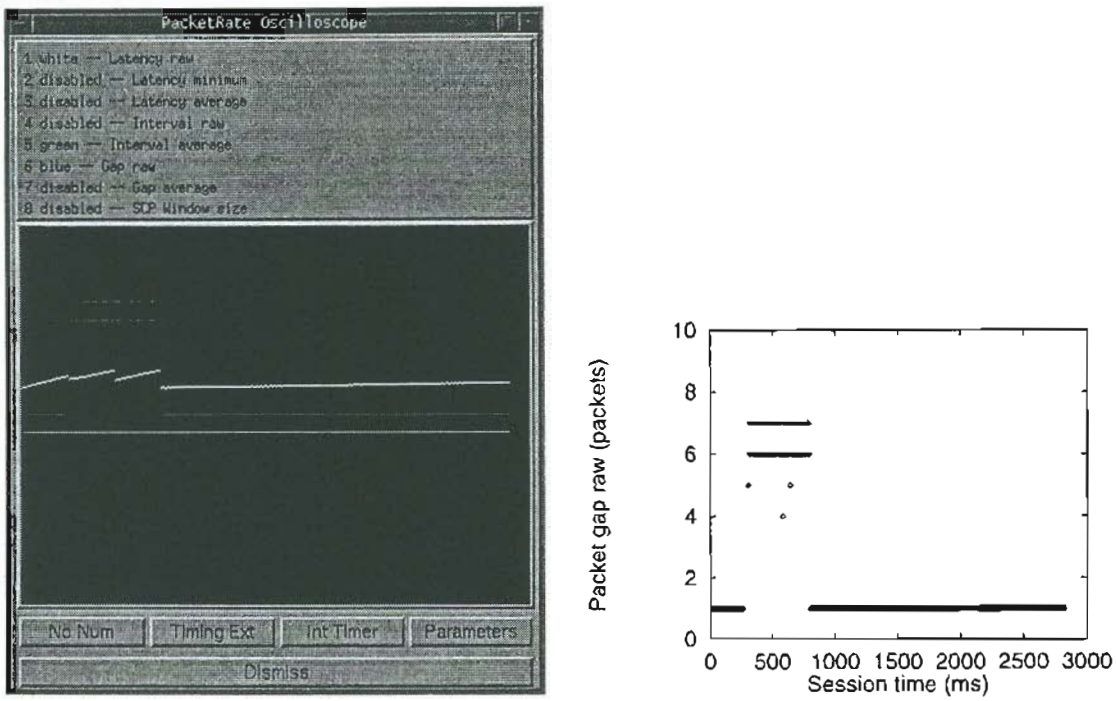
(b) Plot of buffering latency over time

Figure 4.10: Results of a single-session experiment across the 28.8Kbps PPP link

link, experiments with two sessions have also been performed. Both sessions have servers on *lemond*, and clients on *anquetil*. They also have the same parameter settings as above. In all these experiments, the session started earlier always eventually backs off completely, while the later session takes all the available bandwidth. An analysis, to be described later in Section 4.5.1, reveals that the latency feedback policy is inherently unfair in sharing network bandwidth. This issue will be addressed further in Chapter 5.

4.4.2 Experiments Over WaveLAN

Experiments have also been carried out to test the transmission of packets over the WaveLAN with the proposed packet-rate feedback. To make the feedback fit better with the characteristics (higher bandwidth, lower latency) of WaveLAN, in these experiments,



(a) Oscilloscope snapshot

(b) Plot of packet gap over time

Figure 4.11: Results of a single-session experiments across the WaveLAN link

packet size is set to 1469 bytes,[4] no limit is imposed on maximum packet-rate, and the feedback step interval $T = 0.2$ second. All other parameters have the same values as in the PPP experiments.

The first experiment is with a single packet-transmission session from *lemond* to *anquetil*. Figure 4.11(a) shows a snapshot of the scope containing the statistics of the initial phase of the session. To accommodate all the samples in the scope window, scales different than that in the PPP experiment are used. Thus, the curves in the two snapshots should not be compared for absolute values. Also, the values of the latest samples of the channels are not shown on the scope, since otherwise the performance of the session will be adversely affected. This snapshot shows three curves, with the brightest for the raw

[4] IP packet size $= 20 + 8 + 1469 = 1497B$ is the MTU (maximum tranport unit) of Ethernet interface of the server *lemond*.

latency, the dimmest for the raw gaps between packets received, and the other for the average client-received packet-rate. The packet-gap measurements over time are also plotted in Fig. 4.11(b). In this experiment, the buffering-latency is not even close to the target value, so the latency-feedback policy is always dormant. Since the server initially sends packets at its maximum capacity which is larger than what the Wavepoint can sustain, according to the packet-gap curve, soon after the session starts, the buffer in the Wavepoint overflows, and packet-loss feedback is activated. The feedback tracks the client-received packet-rate, and at the end of first feedback step, sends the first feedback packet to the server to slow down its packet rate. From then on, the packet-loss feedback policy keeps exploiting the available bandwidth by maintaining a specified level of packet drop.

Unlike the latency feedback policy, the packet-loss feedback policy is stable and fair in sharing network bandwidth between multiple session. In all the experiments with two packet-loss feedback sessions sharing the same WaveLAN link, after they stabilize, the two sessions are able to split the WaveLAN bandwidth roughly evenly. These experiments confirm the result of a theoretical analysis to be explained in Section 4.5.2.

4.5 Analysis of the Interaction Between Multiple Sessions

The experiments in the previous section indicate that multiple packet-transmission sessions with latency-feedback do not share the same network link in a stable and fair manner, while multiple packet-loss feedback sessions do. In this section, analytical explanations are given to explain why this is the case.

4.5.1 Interaction Between Sessions with Latency Feedback

Suppose there are two latency feedback sessions A and B sharing the same heavily-buffered network link, and they have target buffer fill levels F_a and F_b respectively. Since A and B share the same buffers inside the network, they should have the same buffering latency (buffer-fill level) γ (ignoring the all estimation errors). But in reality, any estimation will inevitably include some errors, so we assume that the buffering latency estimations for A and B are $\hat{\gamma}_a = \gamma - e_a$, $\hat{\gamma}_b = \gamma - e_b$ respectively, where e_a and e_b are estimation errors.

We study the following two cases for the interaction between the two sessions.

Case 1: Suppose the two sessions have different target buffer fill levels (suppose $F_a > F_b$), but their buffer-fill level estimation errors can be ignored ($e_a = e_b = 0$). There are the following three states associated with bandwidth sharing in this case:

1. $\gamma \leq F_b < F_a$ — The buffering latency is lower than the targets of both sessions A and B. So both sessions increase their packet rates, and cause γ to increase.
2. $F_b < F_a \leq \gamma$ — The buffering latency is higher than the targets of both sessions A and B. So both sessions reduce their packet rates, and cause γ to decrease.
3. $F_b < \gamma < F_a$ — The buffering latency is lower than A's target F_a, but higher than B's target F_b. So A increases its packet rate, but B decreases its one.

If the current state is 1, then γ will keep increasing until state 3 or 2 is entered. Similarly, if the current state is 2, then γ will keep decreasing until state 3 or 1 is entered. In theory, there are two possibilities: either the sessions eventually converge to state 3, or they never converge, and keep jumping among the three states. If sessions A and B converge to state 3, in this state, session A would keep increasing its packet rate, and B would keep reducing its rate until it backs off completely. If they do not converge, then the bandwidth partitioning between these two sessions would be unstable and oscillating all the time. As demonstrated by all the experiments performed, we believe that multiple competing sessions will always converge to state 3, and eventually one session will take all the bandwidth, and the others will back off completely.

Case 2: Suppose sessions A and B have a common target buffer-fill level ($F_a = F_b$), but they have different buffer-fill-level estimation errors (suppose $e_a < e_b$ so $\hat{\gamma}_a > \hat{\gamma}_b$). In this case, there are the following three states.

1. $\hat{\gamma}_b < \hat{\gamma}_a \leq F_b = F_a$ — The buffer-fill level estimations of both sessions A and B are lower than their common target. So the two sessions increase their packet rates, and cause γ to increase.
2. $F_b = F_a \leq \hat{\gamma}_b < \hat{\gamma}_a$ — The buffer-fill level estimations of both sessions A and B are higher than their target. So the two sessions reduce their packet rates, and cause γ to decrease.

3. $\hat{\gamma}_b < F_b = F_a < \hat{\gamma}_a$ — Sessions A's estimation $\hat{\gamma}_a$ is higher than the common target, but session B's estimation $\hat{\gamma}_b$ is lower. In this state, session B increases its packet rate, while session A decreases its one.

This case is very similar to the previous case. The two sessions A and B will either converge to state 3, in which session A eventually backs off completely, leaving session B to take all the available bandwidth, or the session will jump among the states, and get an unstable and oscillating bandwidth partitioning. Our experiments also showed the competing sessions tend to converge to state 3.

The two sessions A and B might be able to share network bandwidth only if $e_a - e_b = F_b - F_a$. In this case, the two sessions either both increase, or both decrease their packet rates. Unfortunately, even in this case, the sharing is not well-defined

It is also infeasible to reach a strict relationship ($e_a - e_b = F_b - F_a$) between two measurement errors (including sampling noise), which by definition are somewhat random.

The experiments in Section 4.4.1 for multiple packet transmission session with latency feedback policy fall into case 2. The two sessions have the same feedback parameters. But the session started first (A) has a lower buffer-fill-level estimation, thus backs off completely. To explain the situation, we notice that A starts with an otherwise quiet network link, thus it has a minimum estimation error e_a. When session B starts, the network buffer is already filled with packets from session A, and the network buffer may never become empty again. This residue in buffering causes have some residual effect in the estimation of the minimum transmission latency, and leads to a larger estimation error e_b. The relation, $e_a < e_b$, means that session A, the one which starts first, will always back off.

4.5.2 Interaction Between Sessions with Packet-Loss Feedback

Suppose that the two sessions A and B sharing the same lightly-buffered network link have packet-rate incremental coefficients Δ_a and Δ_b respectively for their packet loss feedback policies, and that the total available bandwidth of the shared network link is μ.

Also suppose that there is a steady state in which sessions A and B have client-received packet-rates μ_a, μ_b and server-sending packet-rates λ_a, λ_b, respectively. We have following observations:

1. The aggregate link bandwidth is the sum of the two sessions:

$$\mu_a + \mu_b = \mu$$

2. The control policy for session A is:

$$\lambda_a = \mu_a + \Delta_a$$

3. The control policy for session B is:

$$\lambda_b = \mu_b + \Delta_b$$

Thus we have an aggregate packet-rate λ input to the shared network, and aggregate packet-loss rate ρ as, defined by the following formulas:

$$\begin{aligned} \lambda &= \lambda_a + \lambda_b = \mu + (\Delta_a + \Delta_b) \\ \rho &= \frac{\Delta_a + \Delta_b}{\lambda_a + \lambda_b} = \frac{\Delta_a + \Delta_b}{\mu + (\Delta_a + \Delta_b)} \end{aligned}$$

The above formula for ρ shows that it is wholly defined by the parameters, and is independent of any transient state. Assuming that the shared network drops packets from different connections evenly (which is the case for most existing networks), then both sessions A and B should also have the same packet drop rate ρ. So we have following formulas for the server-sending packet-rate for each session:

$$\begin{aligned} \frac{\Delta_a}{\lambda_a} = \rho \quad &\Rightarrow \quad \lambda_a = \frac{\Delta_a}{\rho} \\ \frac{\Delta_b}{\lambda_b} = \rho \quad &\Rightarrow \quad \lambda_b = \frac{\Delta_b}{\rho} \end{aligned}$$

The formulas above indicate that in the steady state, the two sessions share the network stably, and the bandwidth a session takes from the shared network is proportional to its packet-rate incremental coefficient. Though not further investigated in the thesis, we believe that the steady state actually exists, and the sessions converge to it.

4.6 Discussion

This chapter demonstrates how the methodologies and components in the software feedback toolkit are used in the development of an adaptive real-time packet-rate-control feedback system. The proposed feedback has multiple repluggable policies, which are activated or deactivated when triggered by associated events. The implementation of the feedback system makes extensive use of the components in the toolkit library. The experimental results indicate that the packet-rate feedback adapts to different types of networks well.

Though the proposed packet-rate feedback is highly adaptive, some shortcomings prevent it from being readily used in serious applications such as streaming video and audio players. As discussed in Section 4.5.1, the latency feedback policy has problems sharing network bandwidth between multiple sessions. Both feedback policies are rate-based. Their lowpass filtering in packet rate estimation needs time to detect change in packet-rate, so they are not responsive enough to network congestion. Also both policies rely on negative acknowledgement, with which the server only reduces its packet rate when requested by the client. These types of policies may fail in the presence of severe network congestion, when the request to reduce the packet rate cannot get through to the server. In Chapter 5, we propose a rate- and congestion-window-based streaming control scheme, which overcomes all of the problems mentioned above as well as being adaptable to different network conditions.

Chapter 5

SCP: Flow and Congestion Control For Internet Media Streaming

5.1 Introduction

The real-time distribution of continuous audio and video data via streaming multimedia applications accounts for a significant, and expanding, portion of the Internet traffic. Many research prototype media players have been produced, including the Berkeley MPEG player [52], the OGI (Oregon Graduate Institute) distributed video player [10], the Vosaic player [12], and the Mbone tools [33]. Over the past year, many industrial streaming media players have also been released, such as the Netscape streaming video plug-ins [37], RealAudio [42] and Vxtreme [63]. It is expected that real-time media streaming traffic will increase rapidly, and will soon make up a significant portion of the total Internet bandwidth.

The key characteristics of such real-time streaming applications are the potential for high data rates, and the need for low and predictable latency and latency variance. Unfortunately, the Internet is characterized by a great diversity in host processing speed and network bandwidth, wide-spread resource sharing, and highly dynamic workload. The Internet is also currently a best-effort network, without any facility for resource reservation or Quality-of-Service (QoS) guarantees. Consequently, Internet-based applications experience large variations in available bandwidth, latency and latency variance. For a streaming application to survive in this highly dynamic Internet environment, feedback-based adaptation and robustness in the presence of data loss are necessary. For example,

streaming media players can preserve the real-time play-out of their data by adaptively sacrificing other presentation quality dimensions such as the total reliability, video frame rate, spatial resolution and signal-to-noise ratio. Through feedback-based adaptation, streaming applications can dynamically discover the currently available bandwidth, and scale the media in one or more of these quality dimensions to fully utilize that bandwidth. To mask short-term variations in the available bandwidth or end-to-end latency, players typically buffer data at the sender or receiver or both. Reliability through indefinite data retransmission is not desirable, since streaming applications can often tolerate some degree of data loss, but can not usually tolerate the delay introduced by the retransmission of lost data.

Successful adaptation relies on accurate and reliable discovery of the currently available network bandwidth. Bandwidth discovery can be achieved through flow and congestion control mechanisms. It is important for such mechanisms to be "good network citizens". That is, they should not allow an undue amount of traffic to be generated, congesting the network, and causing other network traffic to back-off unfairly. Similarly, they should be sensitive to increases in network congestion, and should respond to them by backing-off. Without this behavior, their potential to generate very high data rates could cause serious congestion in the Internet, and perhaps another Internet congestion collapse [18, 23]. Consequently, such mechanisms must operate in harmony with TCP [23], which is the base for the currently dominant FTP [41] and Web/HTTP [1] traffic. They should ensure that multiple streaming sessions share the network among themselves and with other non-streaming traffic in a fair manner. Finally, they should attempt to minimize latency and maximize the smoothness of the streaming data.

There have been several approaches proposed in the literature, such as receiver-initiated rate-based feedback [12, 52], RTP [54] with rate-based feedback [4], sender-initiated rate-based congestion control [30], TCP [23], TCP minus retransmission [22], etc. Unfortunately, they fail to have one or more of the properties described above. Rate-based feedback [4, 12, 52] is inherently sluggish in reacting to network congestion (due to its time- or state-based rate estimation), and has the danger of failure in the presence of severe congestion (due to its negative acknowledgement, where the sender reduces the packet

rate only when told by the receiver). Sender-initiated rate-based schemes [30] avoid the danger of failure, but still have a sluggish rate-estimation process. TCP [23] has been known to be a good citizen, and enables bandwidth sharing between multiple sessions, but its infinite retransmission results in wasted network bandwidth (by retransmitting late data) and highly unpredictable latency and jitter. TCP's throughput is inherently jerky due to its repeated process of window-size increase until packet loss, followed by exponential back-off. While removing the retransmission from TCP [22] eliminates the associated bandwidth waste and latency unpredictability problems, the burstiness in data throughput still remains.

We have proposed SCP (Streaming Control Protocol) [11], a unicast[1] streaming flow and congestion control scheme that has the properties described above. Similar to the congestion control in TCP, SCP employs sender-initiated congestion detection through positive acknowledgement, and uses a congestion-window-based policy to back-off exponentially. During the start-up phase, SCP uses a TCP-style slow-start policy[2] to quickly discover the available network bandwidth. The similarities of SCP congestion control to that of TCP make SCP as robust and as good a network citizen as TCP, and enable the two of them to share the Internet fairly. But unlike TCP, when the network is not congested, SCP invokes a combined rate- and window-based flow control policy that maintains smooth streaming with maximum throughput and low latency. While TCP repeatedly increases its congestion window size, causes packet loss, and backs off, SCP tries to maintain an appropriate amount of buffering in the network for sufficient utilization of available bandwidth, but no more than that. SCP also ensures fair and stable partitioning of network bandwidth between multiple streams. SCP does not retransmit data lost in the network, thus it avoids the associated unpredictability in latency and wasted bandwidth. Streaming sessions also have unique properties not identified in non-real-time sessions, such as limited source data rate (e.g., a typical MPEG1 video stream has a maximum

[1]There are two types of streaming: unicast and multicast. A unicast stream is sent from a single sender to a single receiver, while a multicast stream can be simultaneously received by multiple receivers. Because of this fundamental difference, unicast and multicast streaming need to be treated differently in flow and congestion control as well as in many other aspects.

[2]The slow-start policy in TCP starts with a congestion-window size of 1 and increases the window size exponentially until congestion is detected

data rate of around $1.5Mbps$) and pauses in the middle due to user interaction (e.g., when the user hits a "pause" button). SCP is designed to handle these characteristics properly. SCP also has mobility awareness [20]. A mobile host may dynamically switch between network interfaces connecting it to different networks with different properties, such as link speed, latency and workload. For example, while in an active video conferencing session, a notebook may be un-docked, thus switching from Ethernet to wireless PPP, or docked and switched back to Ethernet. Upon mobility events such as switching between different network interfaces, SCP's internal states and parameter estimators can be reset. Then the slow-start policy is invoked to quickly discover the capacity of the new network connection.

SCP is an excellent application of the software feedback toolkit. It is a sophisticated wide-range feedback system with several repluggable feedback-based policies for different network conditions. Upon events signalling changes in network and session conditions, meta-adaptation actions are performed to switch between the policies. SCP requires the estimation of packet round-trip time (RTT) and data rate, which can be easily implemented with the building blocks provided by the toolkit. The tools in the toolkit can help in online instrumentation of the performance of SCP, and in parameter tuning.

This chapter focuses on the design and implementation of SCP following the software feedback toolkit approach, as well as an evaluation of SCP's performance through Internet experiments. In Section 5.2, we present the design of SCP as a set of guarded feedback policies, which are plugged together through events and guard-based meta-adaptation. An analysis of the bandwidth sharing between multiple SCP sessions is given in Section 5.3. Next, in Section 5.4, we describe the implementation of SCP, which makes use of the building blocks in the feedback toolkit component library. Then in Section 5.5, we discuss experimental results, which demonstrate the performance of SCP, and its ability to share network bandwidth between multiple SCP and TCP sessions. Finally, in Section 5.6, we summarize the work presented in this chapter.

5.2 SCP Streaming Control Architecture

In this section, the SCP streaming control architecture is designed following the methodologies in the software feedback toolkit. SCP is composed of a set of feedback-based congestion-window-size adjustment policies, each of which has a domain (network condition) in which it is applicable. When triggered by events indicating changes in network condition, these policies are activated or de-activated by SCP through guard-based meta-adaptation actions. Firstly, the scenario of an SCP streaming session and the internal states and parameter estimators used are described. Then the overall streaming control architecture is presented. Finally more detailed descriptions of the individual policies and the transition between them upon the events are given.

5.2.1 Real-Time Media Streaming with SCP

A unicast streaming scenario with SCP consists of a media sender and a media receiver, linked by a network connection, as shown in Fig. 4.1 in Section 4.1. SCP resides at the sender side. Media packets are streamed in real-time from the sender to the receiver. Each packet carries, among other fields, an incremental sequence number. For each packet, SCP records the time when it is sent, and starts a separate timer. The receiver acknowledges each packet received with an ACK carrying the sequence number of the data packet. In this section, we assume that all the packets are transmitted by the network in-order. The network may drop or duplicate packets, but will not reorder them. Thus a gap in sequence number between two ACKs received back-to-back indicates packet loss. This assumption will be relaxed in the actual implementation, discussed in Section 5.4. Based on the reception of ACKs and timer expiration events, the sender adjusts the size of its congestion window to control the flow and hence avoids network congestion. SCP maintains the following internal state variables and parameter estimators.

- *state* — The current state: *paused*, *slowStart*, *steady*, or *congested*. Each state corresponds to a specific network and session condition and a flow and congestion control policy.

- *next* — The sequence number of the next packet to send. It is incremented by 1 after sending each packet.
- *acked* — The sequence number of the latest packet whose ACK was received or whose timer has expired.
- W_l — The size of the congestion window (number of packets).
- $W = next - (acked + 1)$ — The number of outstanding packets (sent but not acknowledged). When $W < W_l$, the congestion window is open, and more packets can be sent, otherwise it is closed.
- W_{ss} — The threshold of W_l for switching from the slow-start policy to the steady-state policy.
- $\hat{T}_{brtt}$ — Estimator of the base RTT (round-trip time), the transmission RTT of a packet sent when the network is otherwise quiet.
- $\hat{T}_{rtt}$ — Estimator of the recent average RTT.
- $\hat{D}_{rtt}$ — Estimator of the standard deviation of the recent RTT.
- $\widehat{rto}$ — Estimator of the timer duration, the time a timer lasts from its initialization until it expires.
- $\hat{r}$ — Estimator of the packet rate – the rate at which ACKs are received.

As long as the congestion window is open, the sender streams packets at a rate no more than $\frac{W_l}{\hat{T}_{brtt}}$, instead of bursting them out in chunks, so as to improve smoothness in streaming. Whenever an ACK is received, or a timer expires, the congestion window size W_l is adjusted using a policy determined by the current state.

- ACK_i is received. If $i > acked$ ($i > acked + 1$ indicates a gap in ACK), then $acked \leftarrow i$, estimators $\hat{T}_{brtt}$, $\hat{T}_{rtt}$, $\hat{D}_{rtt}$, $\widehat{rto}$ are updated, all un-expired timers for packets up to and including i are reset, and W_l is adjusted. If $i <= acked$, ACK_i is a duplicate and is ignored.
- The timer for packet i ($i > acked$) expires The un-expired timers for all packets up to i are reset,[3] and W_l and $\widehat{rto}$ are adjusted.

[3]It is possible for a timer started earlier to remain running after another one started later has expired, because the timer-expiration-time estimation $\widehat{rto}$ changes over time.

5.2.2 Overall Architecture

The observations on which SCP is based are similar to those discussed in Section 4.1. Excessive packets in the round-trip network connection (from the sender to the receiver, then back to the sender) fill up the buffers inside the routers and switches, and increase in RTT. When too many packets are held inside the network, network buffers can overflow, and packets are dropped. The amount of buffering is affected by all the real-time streaming and non-real-time data transfer sessions sharing the network.

To adapt the streaming to the changing network conditions, SCP tries to quickly find out how much buffering is appropriate for maximum throughput while avoiding over-buffering, or buffer overflow and resultant packet-loss, which is the optimal network condition for media streaming. As SCP runs, it keeps pushing the appropriate number of additional packets into the network connection to maintain the optimal condition, and traces the changes in available bandwidth closely. SCP reacts immediately when network congestion is detected. To ensure smooth streaming, SCP paces out packets, instead of bursting them out as long as the congestion window is open. Based on the current condition of the network and the streaming session under control, SCP is in one of four states: *slowStart*, *steady*, *congested* or *paused*. Each state is associated with a specific network and streaming session condition (the domain) and a policy for congestion window size adjustment, as listed in Table 5.1.

The domain of each state (policy) is guarded against events indicating when the domain is entered or left. Upon these events, meta-adaptation actions are taken by SCP to update its internal states and to switch to a new state (and associated policy). There are events indicating whether an SCP session is paused or active, and whether the available network bandwidth has been discovered, whether the network is congested. SCP also manages explicit events such as network interface switches. Table 5.2 lists the events, their categories, the states (domains) guarded against the events, and the meta-adaptation actions taken. Figure 5.1 then shows how SCP transfers between its states upon the events. If multiple events happen simultaneously, the event listed first in Table 5.2 takes priority. After initialization, SCP stays in the *paused* state until the sender requests sending a

State	Network and session condition (domain)	Congestion window adjustment policy
slowStart	Available bandwidth not discovered yet.	SCP opens the congestion window exponentially (relative to elapsed time) by increasing the window size by one upon the receipt of each ACK.
steady	Available bandwidth being fully utilized.	SCP maintains appropriate amount of buffering inside the network to gain maximum throughput, avoid excessive buffering or buffer overflow, and trace the changes in available bandwidth.
congested	The network is congested.	SCP backs off multiplicatively by halving the window size. Persistent congestion results in exponential back-off.
paused	No outstanding packet in the network.	When a new packet is sent, SCP shrinks the window size and invokes slow-start policy.

Table 5.1: SCP states, their associated network conditions (domains), and feedback-based congestion window size adjustment policies

Category	Events	Guarded states	SCP Meta-adaptation actions
Mobility indication	Network interface switch	all states	Resets and enters the *paused* state.
Network becomes congested	timeout; gap in ACKs	*slowStart*, *steady*, *congested*	Backs off and and enters the *congested* state.
Bandwidth becomes fully utilized	RTT significantly long ($\hat{T}_{rtt} > K\hat{T}_{brtt}$ where $K > 1$); $W_l \geq W_{ss}$	*slowStart*	Enters the *steady* state.
Session becomes paused	No more outstanding packet ($W = 0$)	*slowStart*, *steady*, *congested*	Enters the *paused* state.
Session becomes active	A new packet is sent	*paused*	Shrinks the window size and enters the *slowStart* state.

Table 5.2: Events and SCP meta-adaptation actions

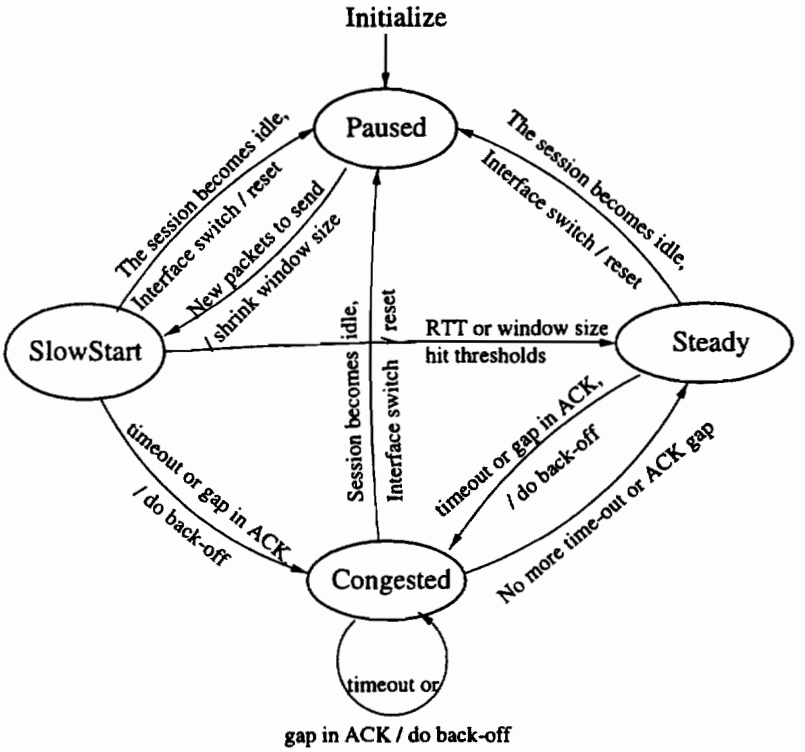


Figure 5.1: SCP state transition diagram

packet. Upon this request, SCP enters the *slowStart* state, and invokes the slow-start policy to quickly open up the congestion window and detect available bandwidth. This process does not end until either the network becomes congested, or the congestion window is sufficiently big. i.e., RTT is sufficiently long, or W_l hits threshold W_{ss}. In the latter case, SCP enters the *steady* state, in which it streams out packets smoothly, and traces changes in bandwidth availability closely. At any time if network congestion is detected, SCP backs off by shrinking the window size in half. After each back-off, SCP stays in the *congested* state until the effects of the new, halved window size can be observed. Persistent congestion triggers exponential back-off. Whenever all already-sent packets have been acknowledged and no more packets are pending for streaming, SCP becomes idle and enters the *paused* state. Later when a new packet is to be sent, SCP becomes active again. It shrinks its congestion window size based on the length of the time it stayed idle, and enters the *slowStart* state. At any time, whenever an event indicating a network interface switch is received, SCP resets itself, discards all the existing estimations and internal states, and re-starts afresh with a slow-start policy to quickly adapt to the new and possibly totally different network environment.

5.2.3 Initialization

Upon initialization, SCP sets $acked = 0$, $next = 1$, $W = 0$, $W_l = 1$ and $W_{ss} = L_w$, where L_w defines an absolute limit on how big the congestion window size can grow. $\hat{T}_{brtt}$, $\hat{T}_{rtt}$ and $\hat{D}_{rtt}$ are all set to ∞. The estimator $\widehat{rto}$ is set to an initial default value. After initialization, SCP enters the *paused* state. Later, sending the first packet brings SCP to the *slowStart* state.

5.2.4 Slow Start

The slow-start policy is invoked after initialization or when SCP resumes from a pause. Its goal is to quickly open up the congestion window and detect the available network bandwidth. The congestion-window size W_l is incremented by 1 upon each ACK received in-order. As a result, W_l increases exponentially since it is doubled in each each RTT

period of time. SCP leaves the *slowStart* state when any one of following events happen, indicating either the bandwidth has been fully utilized, or the network has become congested:

- Network congestion — A timer for a pending packet expires, or a gap in ACK is detected. At this moment, SCP performs back-off, and enters the *congested* state.
- Available bandwidth utilized — The congestion window threshold is hit ($W_l \geq W_{ss}$), or the RTT becomes significantly longer than the base RTT ($\hat{T}_{rtt} >= K\hat{T}_{brtt}$, where $K > 1$ is an RTT threshold coefficient). Upon either of these two events, SCP enters the *steady* state.
- The session becomes idle — All packets are acknowledged and no new packet is pending for streaming, SCP enters the *paused* state.

5.2.5 Steady-State Smooth Streaming

By pushing an appropriate number of extra packets to keep the right amount of pressure, the steady-state policy maintains an optimal amount of buffering inside the network connection. This pressure enables sufficient utilization of the available bandwidth while avoiding excessive-buffering and resultant sluggishness in response to user actions. In the *steady* state, estimation of ACK rate $\hat{r}$ is enabled. Whenever a new $\hat{r}$ estimation is available, W_l and W_{ss} are adjusted according to Equations 5.1 and 5.2, where W_Δ is a constant referred to as window-size incremental coefficient. This policy is similar to the packet-loss feedback policy presented in Section 4.2.1, but here the congestion-window size, instead of the streaming data rate, is what is controlled.

$$W_l = \hat{r}\hat{T}_{brtt} + W_\Delta \tag{5.1}$$

$$W_{ss} = W_l \tag{5.2}$$

The idea behind the flow-control Equation 5.1 above is that SCP assumes that $\hat{r}$ is an approximation to the actual network available bandwidth, and calculates the bandwidth-delay product of the network connection when buffering is kept minimum: $\hat{r} \times \hat{T}_{brtt}$. This product is the amount of data SCP should keep pending (sent but not acknowledged) inside the network in order to maintain maximum throughput with minimum buffering.

Since the network is shared and highly dynamic, the available bandwidth for the session in question changes. If SCP just keeps $\hat{r} \times \hat{T}_{brtt}$ amount of data pending, when the available bandwidth decreases, $\hat{r}$ would decrease, thus SCP would trace the change by reducing the amount of pending data. Unfortunately if the available bandwidth increases, there is no way for SCP to detect that. To solve this problem, SCP pushes W_Δ amount of extra pending data. When the available network bandwidth increases, releasing the extra data buffered by the network results in an increase in packet rate $\hat{r}$, which in turn results in a larger amount of pending data as defined by the increased W_l to keep up the pressure. W_Δ determines how quickly and aggressively SCP can adapt to the increase.

Though at the beginning $\hat{r}$ may not be a reliable estimate of the actual available bandwidth, if a stable state exists, the policy will eventually converge to it though iterations. If the current $\hat{r}$ is lower than the available bandwidth, the extra pressure by W_Δ will increase it. Otherwise, if $\hat{r}$ is higher than the available bandwidth, the increasing number of packets buffered inside the network connection will increase the RTT and result in a reduced $\hat{r}$, and thus a smaller amount of data pending. Eventually, $\hat{r}$ will converge to the actual available bandwidth, and W_l, $\hat{r}$ and $\hat{T}_{rtt}$ will stabilize. As discussed in the previous paragraph, later, when network conditions such as available bandwidth or RTT change, the steady state policy traces the changes closely.

The *steady* state lasts until either a gap in ACK is detected, or a timer expires, which indicates network congestion. When either event happens, SCP performs back-off and enters the *congested* state. SCP also enters the *paused* state if the session becomes idle and there is no new packet to send.

5.2.6 Exponential Back-off Upon Network Congestion

Whenever network congestion is detected, as triggered by events such as gap in ACK, or timer expiration, SCP backs off by reducing its congestion window size in half:

$$\begin{aligned} W_l &= \frac{W_l}{2} \\ W_{ss} &= W_l \end{aligned}$$

The data rate estimator $\hat{r}$ is no longer valid and so is reset and disabled. If the back-off is triggered by the expiration of a timer, this triggering may be an indication that the existing timer duration $\widehat{rto}$ is too short, so $\widehat{rto}$ is also doubled. With this exponential decrease in congestion window size and exponential increase in timer duration, in the case when the network is so congested that no packet can get through, SCP would back off exponentially until it virtually stops sending any further packets, thus giving a chance for the network to recover quickly. The process of exponential increase in $\widehat{rto}$ stops when the arrival of a new ACK results in a new $\widehat{rto}$ estimation.

After each back-off, future packets will be streamed with the new congestion window size, but there may still be packets pending inside the network. Loss of these packets does not reflect correctly the result of back-off, and thus should be ignored. The *congested* state is designed so SCP waits until the effect of the halved congestion window size is observed before taking further actions. After each back-off, SCP is put into the *congested* state, where the rate estimator $\hat{r}$ and further back-off are disabled until the first packet sent in the current *congested* state is acknowledged, found lost or times out. If the first packet is acknowledged, then SCP enters the *steady* state, otherwise another round of back-off is initiated.

If all packets are acknowledged, and there are no more packets to send, the SCP session becomes idle and enters the *paused* state.

5.2.7 Pause When No Packet to Send

Every real-time streaming session has a finite data rate, and there may not be data to send when the congestion window is open. Also, a user may want to pause a streaming session temporarily in the middle, such as when he or she presses a "pause" button. If the sender has no data to send for a while, W eventually decreases to 0. At this moment, the streaming session becomes idle, and SCP enters the *paused* state.

When an SCP session is paused, the bandwidth previously used by this session will gradually be taken by other sessions. So when the streaming resumes at a later time, it should start with the *slowStart* state at a reduced congestion window size. Currently, an ad-hoc policy is adopted to half the congestion window in every $\hat{T}_{brtt}$ amount of time

elapsed in the *paused* state. Suppose SCP enters the *paused* state at time *pauseTime*, and resumes to the *slowStart* state at time *resumeTime*, the congestion window size is reduced as:

$$\text{new } W_l \longleftarrow \frac{W_l}{(resumeTime - pauseTime)/\hat{T}_{brtt}}$$

5.2.8 Reset Upon Network Interface Switch

When either end of an SCP streaming session has its network interface switched, the route from the sender to the receiver is changed, and the new connection usually goes through links with totally different capacities (e.g., when switching between Ethernet and wireless PPP). Upon each network interface switch event, all the current estimations of the network condition become invalid, and SCP can be reset. The reset operation sets $acked = next - 1$ (to ignore all pending ACKs), $W = 0$, $W_l = 1$, $W_{ss} = L_w$, and resets all the estimators. After reset, SCP enters the *paused* state, and stays there until there is a new packet to send, at which time it invokes the slow-start policy to quickly discover the bandwidth of the new connection.

5.3 Analysis of Bandwidth Sharing Between SCP Sessions

With the steady-state policy stated in Equation 5.1, it is possible for multiple SCP sessions to share network links in a stable manner. In this section, we analyze a simple case, in which two sessions with the same packet size share a single network link. Both sessions send packets at the maximum rate, whenever the congestion window is open.

Suppose in a steady state, session A has estimations $\hat{r}_a$, $\hat{T}_{brtta}$, $\hat{T}_{rtta}$, and $W_{la} = \hat{r}_a\hat{T}_{brtta} + W_\Delta$. Session B has $\hat{r}_b$, $\hat{T}_{brttb}$, $\hat{T}_{rttb}$, and $W_{lb} = \hat{r}_b\hat{T}_{brttb} + W_\Delta$. Since A and B share the same network link, we can make the following observations:

1. The aggregate packet rate $\hat{r}_l$ of the network link is the sum of the two sessions: $\hat{r}_l = \hat{r}_a + \hat{r}_b$.
2. The base-RTT estimator is the actual link base RTT T_{brttl} plus some estimation error: $\hat{T}_{brtta} = T_{brttl} + e_a$ and $\hat{T}_{brttb} = T_{brttl} + e_b$. When $\hat{T}_{brtt}$ is estimated as the minimum of the past RTT measurements of a session, the main component of the

estimation error is the residual buffering — packets of other sessions and this session preventing the network buffers from becoming empty. A less important component is sampling noise.

3. Sessions A and B have the same RTT estimation (when the sampling noise can be ignored): $\hat{T}_{rtta} = \hat{T}_{rttb} = \hat{T}_{rttl}$. Furthermore, the number of packets sent by a session in one RTT equals its congestion window size. Thus the packet rate ratio of A and B equals the ratio of their window size:

$$\frac{\hat{r}_a}{\hat{r}_b} = \frac{W_{la}}{W_{lb}} = \frac{\hat{r}_a(T_{brttl} + e_a) + W_\Delta}{\hat{r}_b(T_{brttl} + e_b) + W_\Delta} \qquad \Longrightarrow$$

$$\hat{r}_a = \frac{\hat{r}_b W_\Delta}{\hat{r}_b(e_b - e_a) + W_\Delta} \tag{5.3}$$

Combining observations (1) and (3), it is clear that there is a single solution for $\hat{r}_a$ and $\hat{r}_b$. The session with a larger residual buffering tends to have a larger error E, and thus gets a larger portion of the bandwidth. In a special case where the two sessions have the same base RTT estimation ($e_a = e_b$), we have $\hat{r}_a = \hat{r}_b = \frac{\hat{r}_l}{2}$, indicating that the two sessions split the network bandwidth evenly.

The analysis above shows that, when multiple sessions share the same set of network links and the variation in residual error diminishes, bandwidth sharing is stable and is determined by factors such as W_Δ, W_l and $\hat{T}_{brtt}$. What the analysis also indicates is that bandwidth sharing can be controlled by setting these factors explicitly and appropriately.

5.4 Implementation of SCP

An SCP package has been implemented as a layer on top of UDP. It is implemented as a C++ class derived from composite base-class in component library of the software feedback toolkit. The implementation makes use of the building blocks in the feedback component class library for the estimators. The feedback-based policies for adjustment of congestion-window size are built as repluggable feedback components. In this section, we discuss the implementation of the parameter estimators and streaming control policies. We also discuss issues in the implementation, such as packet reordering by the network.

5.4.1 Estimation of Parameters

Round-Trip Time Estimation

To measure the RTT of individual packets, for each packet sent, SCP records, in a table called *packetTable*, its sequence number and the time when it is sent (sending timestamp). Whenever an ACK is received, SCP measures the current time (receiving timestamp), and uses the sequence number in the ACK to look up in *packetTable* the time when the corresponding data packet was sent. SCP calculates the RTT as the difference between the two timestamps. The memory space requirement for *packetTable* is limited, since at any time, it needs only W entries to keep the most recent W packets sent but not acknowledged, and each entry contains only a sequence number and a timestamp. W is never larger than the constant L_w

The base RTT $\hat{T}_{brtt}$, average RTT $\hat{T}_{rtt}$, RTT standard deviation $\hat{D}_{rtt}$, and timer duration $\widehat{rto}$ are estimated based on the history of the raw RTT measurements of the current streaming session. Implemented by a *minimum filter* from the toolkit library, $\hat{T}_{brtt}$ is estimated as the minimum of all the past RTT samples (the whole history). There are two factors that may affect the accuracy of this estimator: residual buffering in the network and route change. Firstly, if before a streaming session starts, the network links are already busy, and the buffers inside the routers and switches never become empty during the session, then due to the residual buffering, the $\hat{T}_{brtt}$ estimation is longer than the actual transmission round-trip time. Secondly, routing in the Internet is dynamic; the route from a sender to a receiver may change over time. If the route for a session becomes shorter in base RTT, the $\hat{T}_{brtt}$ estimator is able to trace the change. However, if the route becomes longer in base RTT, the output of the estimator $\hat{T}_{brtt}$ would be shorter than the current actual base RTT. This problem can be solved if the *minimum filter* for $\hat{T}_{brtt}$ only looks back into a limited-depth history instead of the whole one. Fortunately, Paxson has shown that route changes are infrequent events in the Internet [40].

The average RTT $\hat{T}_{rtt}$ and base RTT $\hat{D}_{rtt}$ estimators are implemented as a single mean and deviation composite feedback filter shown in Fig. 3.6 in Section 3.2.4. The $\hat{T}_{rtt}$ and $\hat{D}_{rtt}$ estimations are done in a way similar to that in TCP [23]. The $\hat{T}_{rtt}$ is the result of

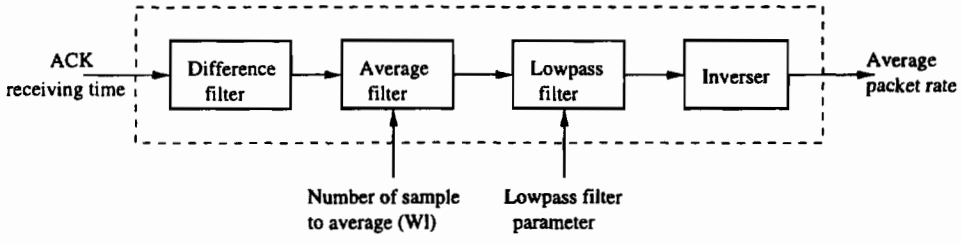


Figure 5.2: Implementation of the SCP packet-rate estimator

applying a lowpass filter to the sequence of raw RTT measurements. $\hat{D}_{rtt}$ is a lowpass filtering of the difference between $\hat{T}_{rtt}$ and the raw RTT measurements.

Timer duration $\widehat{rto}$ is also estimated similar to that in TCP. [23], which proposed a formula $\widehat{rto} = \hat{T}_{rtt} + 4\hat{D}_{rtt}$ This estimator may work well in the TCP implementation in BSD systems, in which coarse-grain $500ms$ clock resolution is used for timers and RTT measurements. However, in our implementation with $10ms$ clock resolution, the above $\widehat{rto}$ estimator results in timers being too sensitive to jitter in RTT. Timers for many ACKs expire though these ACKs are not actually lost and will be received soon. This premature timer-expiration is our experience as well as an observation in LINUX's implementation of TCP[4]. To avoid this problem of too-early false timer expiration, the LINUX TCP implementation modifies the $\widehat{rto}$ estimator to be:

$$\widehat{rto} = \frac{5}{4}(\hat{T}_{rtt} + 4\hat{D}_{rtt})$$

This estimator also works well in our SCP implementation, and thus is adopted. The $\widehat{rto}$ estimator is implemented as a feedback building block with two input ports ($\hat{T}_{rtt}$ and $\hat{D}_{rtt}$) and one output port ($\widehat{rto}$). Its input ports are connected to the two output ports of the RTT mean and deviation estimator composite feedback component.

Packet-Rate Estimation

To estimate the packet rate $\hat{r}$ used by the steady-state policy, whenever an ACK is received, SCP samples the current time, and computes the interval between the current ACK and the previous one. One simple and straightforward $\hat{r}$ estimator would be to apply a lowpass

[4]For detailed information, please see comments around line number 860 in file `net/ipv4/tcp_input.c` of the LINUX version 2.0.18 source code [62].

filter on the raw interval measurement sequence, and invert the resultant smoothed interval to get an average packet rate. However, one property of the packet-switched Internet is that it is asynchronous, and packets paced out smoothly by the sender may arrive at the receiver in clumps. The interval between the packets received within each chunk is much shorter than the sending interval, and many times close to zero. This clumping effect means that even though SCP paces out packets, the ACKs may arrive in clumps, thus resulting in distortion in the estimation produced by the simple rate estimator above. On the other hand, for an SCP streaming session, there are never more than congestion-window-size (W_l) number of packets pending in the network during any round-trip time period $\hat{T}_{rtt}$. Averaging no fewer than W_l interval measurements would eliminate the grouping effect completely. Thus a more reliable packet-rate estimator is to compute the average of at least W_l most recent raw interval samples, apply a lowpass filter to smooth out the average interval, and invert the smoothed interval for average packet rate. Figure 5.2 shows the packet-rate estimator implemented as a composite feedback component built with several building blocks from the toolkit component library. This rate-estimator component takes two parameters: one W_l from the output of the feedback policy components as the number of samples to average, and the other for the lowpass filter.

5.4.2 Implementation of the Feedback Policies

The component feedback polices used in SCP are simple, and are activated or de-activated through guard-based meta-adaptation. We simply pack each of them into a repluggable feedback building block. The output (W_l) of the steady-state policy is also wired to the W_l parameter port of the packet-rate estimator as shown in Fig. 5.2.

5.4.3 Implementation Issues

Handling Packet Reordering

The IP protocol used by the Internet is connectionless and best-effort. IP packets may be dropped, duplicated or reordered. The SCP architecture as described in Section 5.2 deals with ACK loss and duplication properly, but it assumes that ACKs are received

in order. Though packet reordering is usually not a problem in LAN environments, it actually happens in long-haul Internet connections. For example, we have often observed the phenomena of packet reordering in our experiments between hosts at OGI and Georgia Institute of Technology. In the presence of packet reordering, a gap in ACK does not necessarily mean that packets have been dropped, thus leaving room for SCP to speculate. SCP can be aggressive, assuming that the missing ACKs are just late, and wait until the timers for the missing ACKs expire before backing off. Or SCP can be conservative, assuming that the missing ACKs are lost, and back off immediately. SCP can also be adaptive. In this case it assumes temporarily that the missing ACKs are lost, backs off, but hopes that they are actually just late, and hence also saves the current state (including the congestion window size W_l), and continues rate estimation. Later, if the ACKs turn out to just be late, then the saved state is restored. The aggressive approach is less responsive to network congestion and may have problems that prevent the network from recovering from serious congestion quickly. The conservative approach will experience seriously sub-optimal performance in the presence of packet reordering. On the other hand, the adaptive approach responds to potential packet drop immediately, and will not affect streaming performance much if the the missing ACKs are received soon.

SCP takes the adaptive approach. It maintains a table called *ackTable* to record whether individual ACKs have been received or not. Upon each back-off action, just before the triggering event (timer expiration or ACK gap) is processed, *acked*, *next*, W_l and the current state in *state* (which could be *slowStart*, *steady* or *congested*) are saved in variables *saved_acked*, *saved_next*, *saved_*W_l, and *saved_state* respectively. So these variables actually save the state just before the most recent back-off action was taken. The back-off is caused by the absence of some ACKs with sequence number between *saved_acked* + 1 and *acked* inclusive. When a late ACK (whose sequence number is no larger than *acked*) is received while SCP is in the *congested* state, SCP checks *ackTable* to see if all the ACKs with sequence number from *saved_acked* + 1 to *acked* have been received. If the answer is positive, then it restores the saved previous state (except if the previous state is *congested*, when SCP transfers to the *steady* state instead of restoring to *congested*).

The size of *ackTable* is limited, with no more than L_w entries. At any time, no more than $(acked - (saved\_acked + 1) + 1 = acked - saved\_acked)$ entries need to be recorded. The current *acked* value is smaller than *saved_next*, which is the *next* at the time SCP entered the *congested* state ($acked < saved\_next$), otherwise another round of back-off should have been initiated, and the *saved_** variables should have been updated. So we have $(acked - saved\_acked) \leq (saved\_next - saved\_acked)$. Furthermore, $(saved\_next - saved\_acked)$ should be no larger than $saved\_W_l$, which is in turn no more than L_w. As a result, it is certain that *ackTable* needs to accommodate the ACK status of no more than L_w contiguous data packets.

Congestion-Window Adjustment at the Beginning of the Steady State

In some cases, the performance of SCP's steady-state policy, as described in Equation 5.1, can be further improved. Firstly, in the steady-state policy, there is a constant window-size incremental coefficient W_Δ. This W_Δ defines how many extra data packets are maintained outstanding inside the network. It also effectively sets a limit on the minimum congestion window size to W_Δ. This limit remains in place even when the optimal window size may be smaller than W_Δ. In this case SCP would cause continuous network congestion. Secondly, since the steady-state window size relies on the ACK rate estimation $\hat{r}$, it is desirable for $\hat{r}$ to be smooth. But a smooth $\hat{r}$ estimation would require a lowpass filter with a small coefficient (long time constant), thus requiring a long time to react. This long estimation-time means that at the beginning of SCP's steady state, no rate estimation will be generated, and the congestion window size will remain unchanged, even though the network condition is changing. Having a smaller-than-ideal window size for a long time hurts the streaming throughput significantly. To improve the performance of SCP in these two cases, each time after SCP enters the *steady* state, it invokes a TCP-style policy to increase the window size additively. It stays with the TCP-style policy until the congestion window size becomes no less than the minimum window size (equal to W_Δ) and ACK-rate estimation is available. The steady-state policy defined by Equation 5.1 is invoked afterwords. At any time, if network congestion is detected, SCP performs back-off immediately.

Congestion Window Adjustment for Low-Data-Rate Sessions

Every real-time streaming session has a bounded data rate. It is possible that when SCP is in the *slowStart* state, for a long period of time, the rate of a session is lower than the available bandwidth (so that no packet is dropped) but is big enough to keep the session active so that the *paused* state is not entered. In this case, the session will stay in the *slowStart* state, since no events triggering the transition to other states (increase in latency, packet loss, or stream pause) will happen. As a result, the congestion window size will be increased quickly to the limit L_w, which is usually big. On the other hand, only a small number of packets are outstanding. If at a later time there is a big burst of packets (due to variation in stream data rate), they will all be sent out back-to-back, potentially causing heavy packet loss. To prevent this situation, in the *slowStart* state, SCP increases W_l only when the congestion window is sufficiently full, and shrinks it if it is almost empty.

5.5 Experimental Results

To evaluate the performance of SCP, the test program used in Section 4.4 for testing the adpative packet-rate-control feedback has been extended to support SCP streaming as well as TCP and raw UDP. It is also extended to read experiment scripts and parameter settings from text files, and collect statistics in files for analysis. The control panel in the test program has been extended to display and set the parameters relevant to SCP, and the scope panel extended to display SCP congestion window size (which is carried in all data packets).

The network configuration for the experiments is shown in Fig. 5.3. We have two LANs, one at the OGI CSE department, and the other at Georgia Institute of Technology (Georgia Tech), connected through the long-haul Internet with almost 30 hops. At the OGI side, there are four hosts on a 10BaseT subnet (subnet 1): one $100Mhz$ notebook PC running LINUX, called *anquetil*, one $166Mhz$ desktop PC also running LINUX, called *bartali*, and two HP 9000 HPUX workstations, called *lemond* and *smoo*. The notebook *anquetil* also has a $28.8Kbps$ PPP link, and can be optionally connected to another 10BaseT subnet

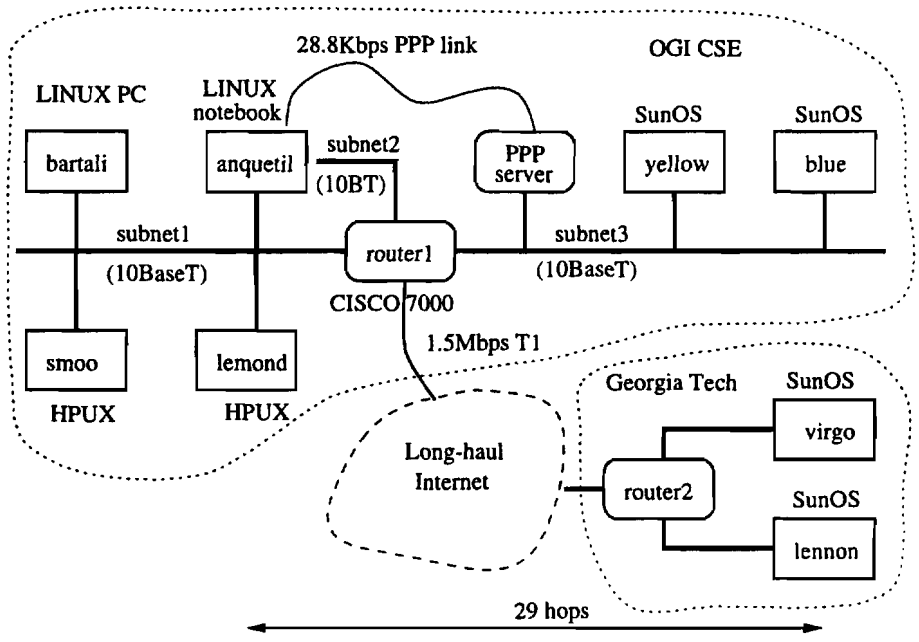
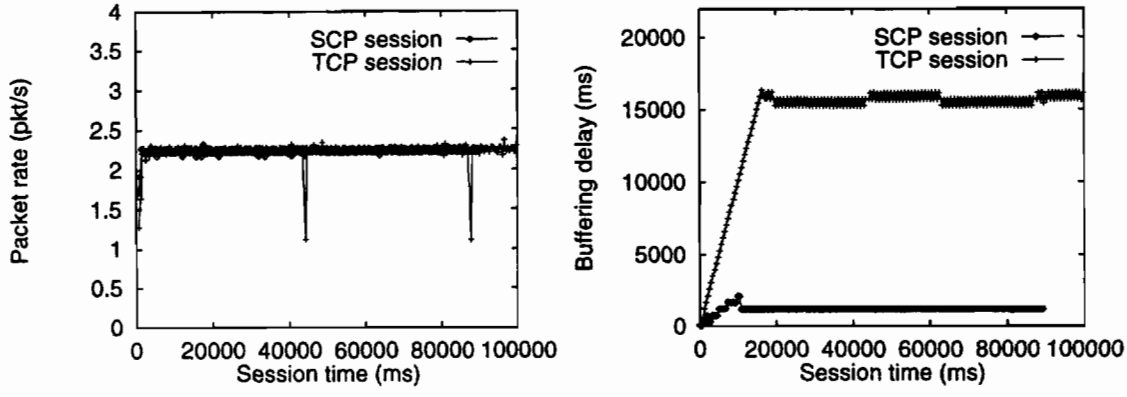


Figure 5.3: Network configuration for the SCP experiments

(subnet 2). On a third 10BaseT subnet (subnet 3) are two Sun SPARC's workstations *yellow* and *blue*, both running SunOS 4.1.3. These 3 subnets are connected through a CISCO 7000 router, as part of the OGI CSE department LAN. At the Georgia Tech side, there are two workstations on two different subnets: *virgo* and *lennon*, both of which are Sun SPARC running SunOS 4.1.3. The link connecting the OGI network to the Internet is a $1.5Mbps$ T1 line. This configuration covers typical types of Internet connections: PPP, LAN (same subnet, and with a single router), and WAN. Network interface switches in mobile environment can be simulated by having the notebook *anquetil* switch between its Ethernet and PPP interfaces.

Experiments have been performed to evaluate the performance of SCP in various network configurations. We evaluate SCP's efficiency, ability to stream smoothly and maintain low latency, reaction to network congestion, and bandwidth sharing between multiple SCP or TCP sessions.

For all the experiments, unless stated explicitly, the following parameter values are used: steady-state congestion-window-size incremental-coefficient $W_\Delta = 3$, RTT threshold coefficient $K = 2$, rate-estimator lowpass-filter parameter $\alpha = 0.1$, data packet size



(a) Packet rate (the packet rate of the two sessions are very lose most of the time)

(b) Buffering delay

Figure 5.4: Performance of single SCP and TCP sessions over PPP

$1472B$[5], and a limit on congestion window size $L_w = 43$.[6] These parameters are not necessarily optimal for all network configurations, but they seem to yield satisfactory performance in all the experiments conducted across the network shown in Fig. 5.3. The size of TCP and UDP/SCP socket sending and receiving buffers are set up to 64KB for better throughput.

5.5.1 Experiments Across a PPP Link

The first set of experiments are between *anquetil* as the receiver and *lemond* as the sender across the $28.8Kbps$ PPP link with an MTU of $1500B$.

First, single TCP and SCP sessions are played. For the SCP sessions, the receiving data rate is $2.24pkt/s \approx 26.9Kbps$, which is close to the $28.8Kbps$ PPP link speed. Figure 5.4(a) shows the packet-rate-over-time of an SCP session and a TCP one. It shows the raw packet rate (inverse of packet interval) versus the session time (the time relative to the beginning of the session). Figure 5.4(b) shows the buffering delay[7] of these two sessions. After a

[5]IP packet size $= 20+8+1472 = 1500B$ which is the Ethernet MTU. $1472B$ is the UDP/TCP payload size, including the SCP header (in the case of SCP), the various fields carried in the packet, followed by a randomized data.

[6]The value of $43 \approx \frac{64KB}{1.5KB}$, where $1.5KB$ is the Ethernet MTU, and $64KB$ is the maximum UDP/TCP socket buffer size for many UNIX systems.

[7]The time T from when the sender SCP or TCP accepts a packet until the receiver gets it, minus the

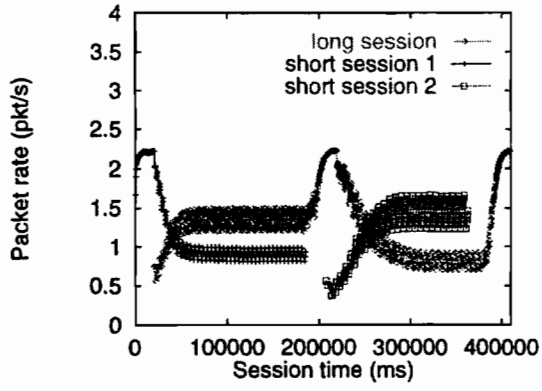


Figure 5.5: Smoothed packet rate of two SCP sessions over PPP

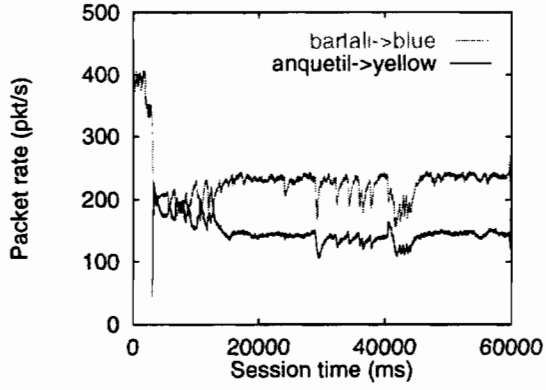


Figure 5.6: Smoothed packet rate of two SCP sessions across a single router

startup phase, the SCP session yields a smooth packet rate, and utilizes the bandwidth sufficiently. The TCP session has a similar data rate, except for the periodical downward spikes, which are caused by packet loss (by the PPP server) and retransmission. However, the SCP and TCP sessions maintain different levels of buffering inside the network. The SCP session has a steady-state buffering delay at about 1.2 second, while the TCP session pushes the buffering delay up to 16 seconds (which corresponds to a full and periodically overflowing buffer in the PPP server) and stays there. Different TCP implementations have very different behaviors over PPP. SunOS TCP has a steady-state behavior similar to that of HPUX TCP, but a much worse start-up phase. LINUX TCP seems to have introduced some techniques for the PPP case to manage the buffering. It pushes the delay to a maximum of 7 seconds, and reduces to about 3 seconds in its steady state.

Figure 5.5 shows the smoothed[8] packet rate of three SCP sessions, when two of them are played simultaneously from *bartali* to *anquetil* across the PPP link. It can be seen that the two competing sessions eventually reach a stable share of the PPP bandwidth. This experiment also shows the effect of the residual buffering error in the base RTT estimation on bandwidth sharing. The base RTT estimation of the long session has a residual buffering caused by the first short session, thus it gets a bigger share of the

transmission latency estimated as the minimum of all the T's in the whole session.

[8]The smoothing is done by applying a lowpass filter with $\alpha = 0.1$ to the raw data. The value $\alpha = 0.01$ is used for all non-PPP packet rate figures.

bandwidth. The second short session then gets a residual buffering error caused by the long session, and also gets a bigger portion of the PPP bandwidth. Further experiments show that if the start times of two competing SCP sessions are close enough, they will split the bandwidth evenly.

Experiments have also been carried out to play one SCP session and one TCP session simultaneously across the PPP link. Since SCP controls the amount of network buffering, the bandwidth partition depends on how aggressive a specific TCP implementation is. For sessions from *bartali* to *anquetil* the packet rates are $0.6pkt/s$ for SCP and $1.7pkt/s$ for TCP. HPUX and SunOS TCP implementations are more aggressive. When SCP and TCP sessions are played from *lemond* to *anquetil*, virtually all bandwidth is taken by the TCP session. These results as well as the long latency of a single TCP connection over PPP imply that TCP is generally too aggressive for streaming across PPP. TCP traffic should be prevented from competing against streaming sessions for PPP bandwidth.

5.5.2 Experiments On a Single Subnet

All the single SCP sessions played from *bartali* to *anquetil* across subnet 1 yield smooth and stable throughput of about $700pkt/s = 8.4Mbps$. The buffering delay of these sessions, which is mainly caused by MAC-level back-off and host processing latency, is kept within $4ms$. For comparison, individual TCP sessions yield less smooth $670 \sim 680pkt/s$ and up to $40ms$ of buffering delay. These results indicate that SCP is efficient, smooth, and has low latency.

When two sessions, either two SCP sessions, or one SCP and one TCP, or two TCP sessions, are played simultaneously from *bartali* to *smoo* and from *anquetil* to *lemond* respectively across subnet 1, in all cases, the two sessions are able to share the Ethernet, but they are all very jerky, with highly variable throughput ($100pkt/s$ to $700pkt/s$) and buffering delay (up to 0.8 second). SCP also drops packets. This result is to be expected and is caused by MAC-level exponential back-off and retransmission failure upon collision. Not much can be done by SCP or TCP to eliminate the problem except for explicitly limiting the data rate at the sender side.

Experiment configuration	Single SCP *anquetil→yellow*	Single TCP *anquetil→yellow*
Packet rate (pkt/s)	320	380
Buffering delay (ms)	0∼30 around 5	5∼80 around 40
Experiment configuration	SCP: *anquetil→yellow* SCP: *bartali→blue*	SCP: *anquetil→yellow* TCP: *bartali→blue*
Packet rate (pkt/s)	150 : 230	SCP 100, TCP 300
Buffering delay (ms)	0∼40 around 14	SCP 0∼50, TCP 10∼80

Table 5.3: Results of single TCP and SCP sessions across a single router

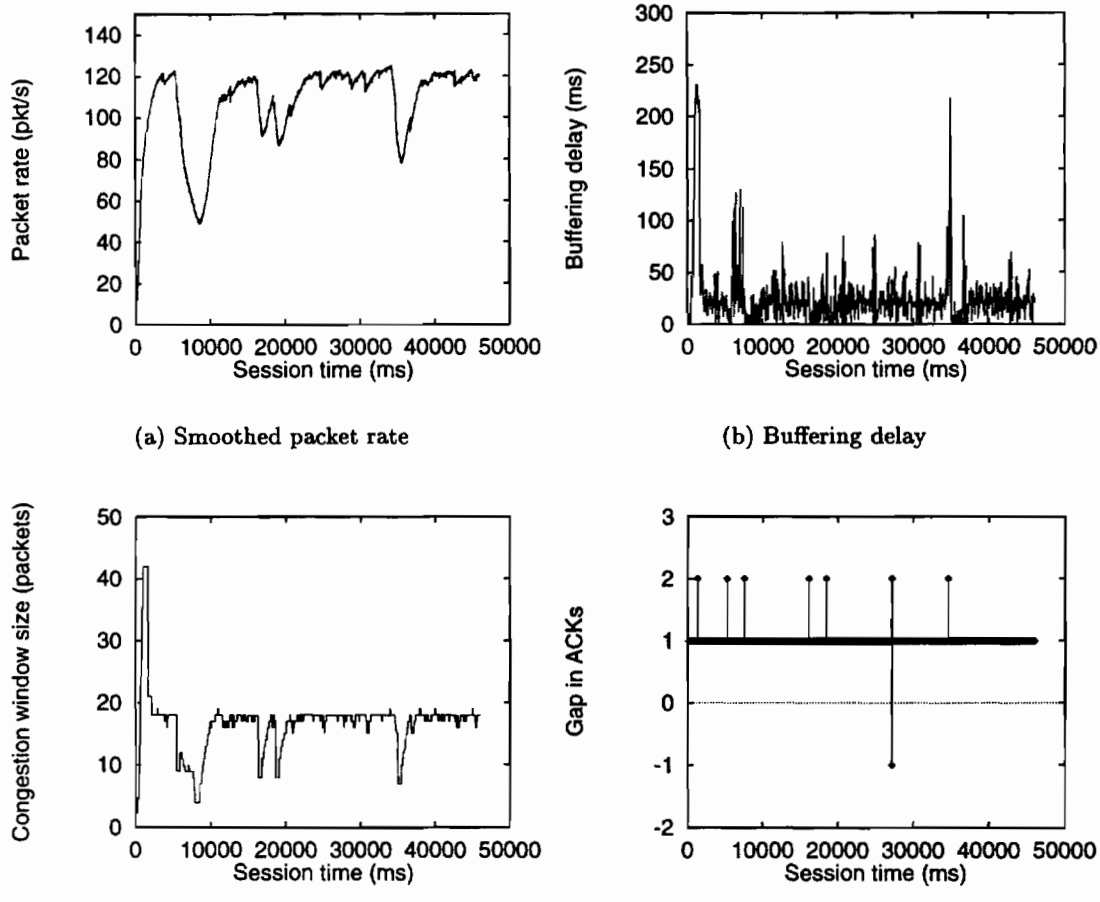
5.5.3 Experiments Across a Single Router

To evaluate the performance of SCP streaming across a single router, *anquetil* is moved to subnet 2, and experiments are performed. Table 5.3 shows the configurations and the overall performance of the experiments. Figure 5.6 shows the smoothed packet rate of two simultaneous SCP sessions. These experiments were done in a late evening when the network is lightly-loaded.

A single SCP session yields a lower throughput than a single TCP session, but has a significantly lower buffering delay. One possible reason for SCP having a lower bandwidth is that SCP maintains a lower level of buffering, thus has more chance of getting empty buffers and idle links.

Figure 5.6 shows that when two SCP sessions are competing for the output port of the single router, they are able to share the bandwidth in a stable manner. To understand why there is a packet rate difference between the two sessions, a closer look at the statistical data shows that the two sessions have the same steady-state congestion-window-size of 5, but they have a different RTT. When played separately, the RTTs for sessions *bartali→blue* and *anquetil→yellow* are around $15ms$ and $22ms$ respectively. When played simultaneously, they are around $22ms$ and $35ms$ respectively. The difference in RTT may be caused by the difference in host CPU speeds and other factors, and results in uneven bandwidth partitioning. This difference does not mean that the SCP is unfair.

When an SCP session is competing against a TCP session, they are still able the share the bandwidth. But SCP's steady-state control of network buffering make it less aggressive



(a) Smoothed packet rate

(b) Buffering delay

(c) Congestion window size

(d) Gap in ACKs

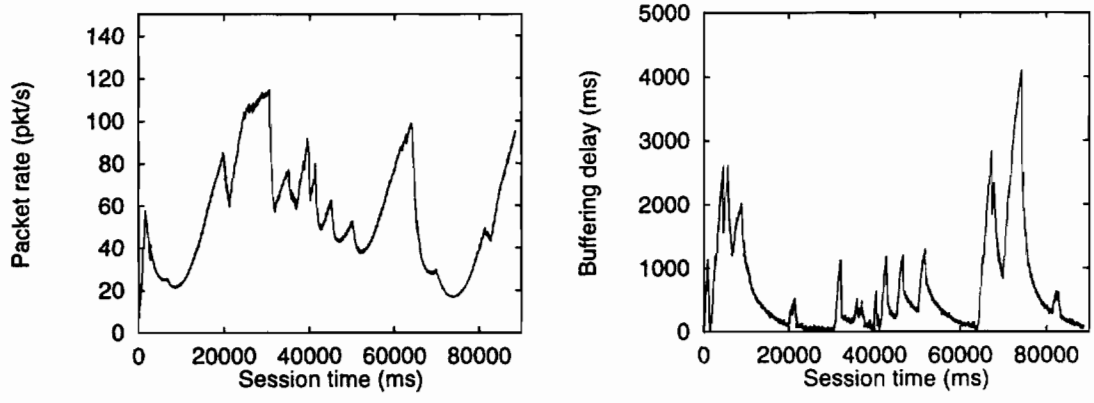
Figure 5.7: An SCP session across the lightly loaded Internet

than TCP and hence it gets a significantly smaller amount of the bandwidth.

5.5.4 Experiments Across the Internet

To evaluate the performance of SCP across the long-haul Internet, SCP and TCP sessions are played between hosts at OGI and Georgia Tech. There is no way to control the Internet. Nevertheless, there are still times, such as midnights and weekends, when the network is relatively lightly loaded, and times such as weekdays when it is heavily loaded.

As shown in Fig. 5.7, when the network is lightly loaded, SCP is able to stream smoothly, and to exploit the network bandwidth sufficiently. From time to time, there



(a) Smoothed packet rate

(b) Buffering delay

Figure 5.8: A TCP session across the lightly loaded Internet

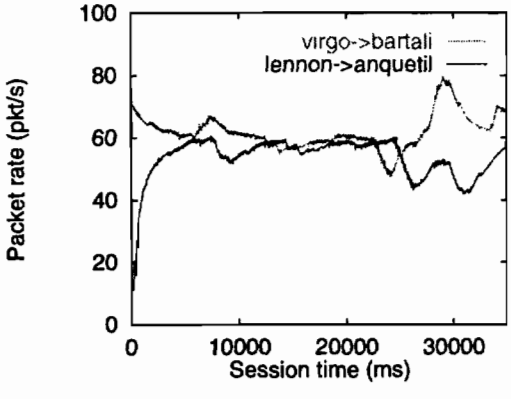


Figure 5.9: Smoothed packet rate of two SCP sessions across the lightly loaded Internet

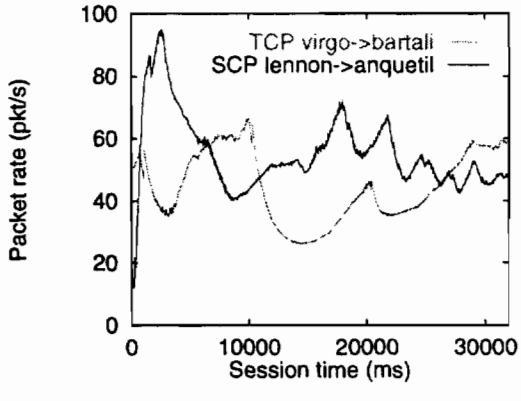


Figure 5.10: Smoothed packet rate of SCP and TCP sessions across the lightly loaded Internet

is still network congestion and packet loss[9], causing SCP to back off. Otherwise, SCP's steady-state rate- and window-based flow control policy maintains a stable congestion window size of about 18 packets, a stable throughput of about $115pkt/s(\approx 1.4Mbps$, close to the $1.5Mbps$ T1 line speed), and low buffering delay. On the other hand, as shown in Fig. 5.8, single TCP sessions have much jerkier throughput, and long and unpredictable buffering delays (up to 4 seconds).

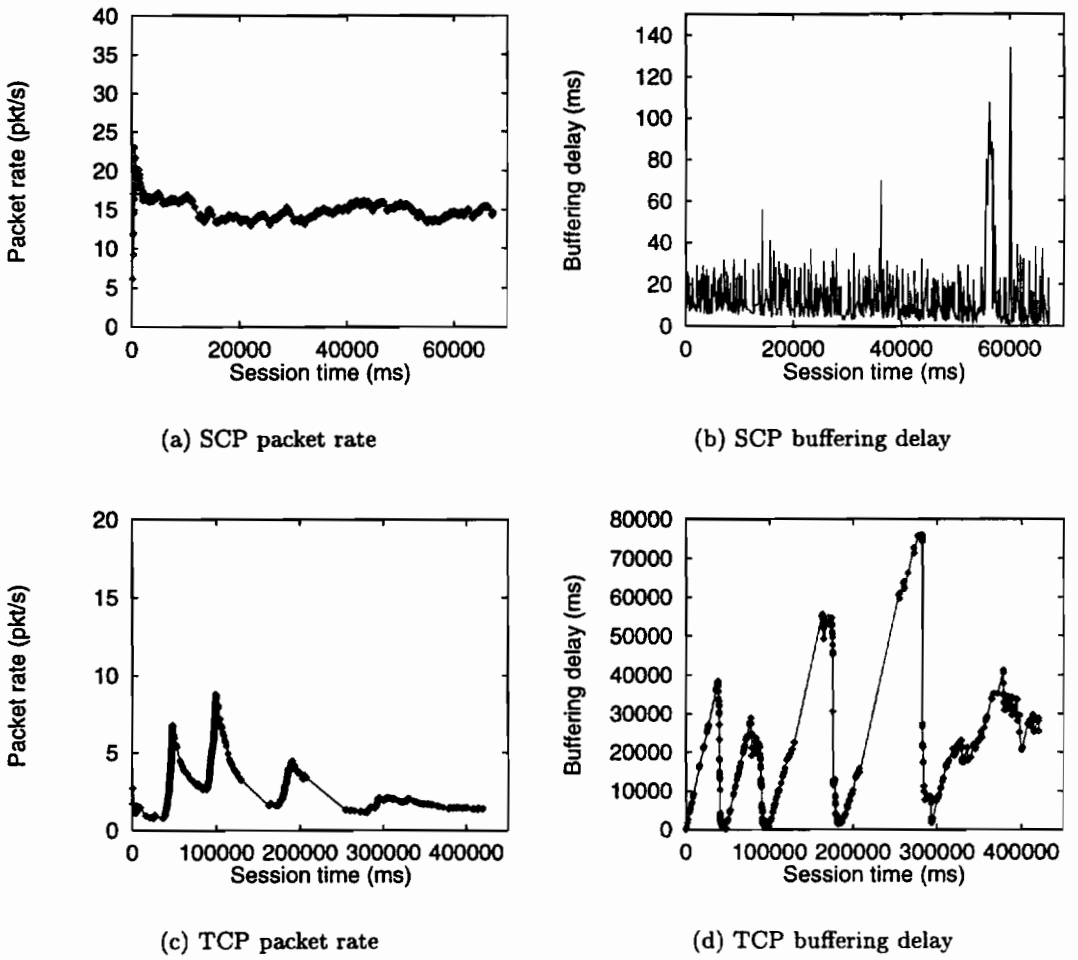
Two simultaneous SCP sessions are able to share the bandwidth in a fair manner. In our experiment, two competing SCP sessions, *lennon→anquetil* and *virgo→bartali*, are played across the lightly loaded Internet. The ratio of the average bandwidth between the two sessions goes from 7 : 4 to 1 : 1. The buffering delay for the sessions are kept mostly below $100ms$. Figure 5.9 shows the packet rates of two simultaneous SCP sessions. Random packet drop in routers upon congestion causes competing sessions to back-off randomly, but on average the bandwidth sharing is fair, and the throughput is not too jerky.

SCP and TCP sessions across the long-haul Internet share the network bandwidth more evenly than when across a single router. Figure 5.10 shows the packet rate of two simultaneous SCP and TCP sessions. The two competing sessions, SCP: *lennon→anquetil* and TCP: *virgo→bartali*, produce a throughput ratio around $SCP5 : TCP4$. While TCP sessions have long latency, the buffering delay of the competing SCP sessions are still below $100ms$ for most packets.

During busy weekdays, it is more obvious that SCP yields smoother throughput and lower and more consistent delay than TCP, while still operating in harmony with the latter. The performance results of single SCP and TCP sessions from *virgo* and *bartali* are shown in Fig. 5.11. In each session, 1000 $1469B$-sized packets were streamed. Due to the difference in throughput, the two sessions lasted different amount of times. The network is so congested that even "`ping virgo`" from *bartali* shows up to 10% packet loss. SCP sessions consistently get about $14pkt/s$ with buffering delay[10] below $100ms$. SCP

[9]The gaps in ACK is shown in Fig. 5.7(d), which also shows that there is an out-of-order packet. In this case, W_t is halved but restored before any packet is sent, thus the back-off is not shown in Fig. 5.7(c))

[10]In a busy network when the buffers in routers are never empty, buffering delay actually reflects the variations in buffering delay.



(a) SCP packet rate

(b) SCP buffering delay

(c) TCP packet rate

(d) TCP buffering delay

Figure 5.11: Performance of single SCP and TCP sessions across the busy Internet

sessions observed up to 10% loss in ACKs. Due to the heavy packet loss and frequent retransmission, TCP sessions get only 2 to $6pkt/s$, with delay up to 70 seconds, and the throughput is very jerky. A delay of 70 seconds is simply not acceptable to any streaming applications!

5.5.5 Experiments with Network Interface Switching

To evaluate SCP's ability to adapt to different network connections upon the event of network interface switch, both the PPP and Ethernet interfaces of the notebook *anquetil* are activated, and SCP sessions are played from *bartali* to *anquetil* through either the PPP or the Ethernet link. Since the version of LINUX running on the notebook *anquetil* does not support IP level hot-swapping, switching between the PPP and LAN interfaces is simulated by a utility that reconfigures the default IP route, and sends a HUP (hung-up) signal to the receiver of the on-going streaming session. The test program is extended so that whenever the receiver receives a HUP signal, it re-establishes the control and data connections to the sender, and optionally resets the SCP state on the data connection, and continues the session from where it was interrupted.

In our experiment, upon network switch, SCP is able to adapt to network changes quickly and utilize the available bandwidth right away, with or without being reset. When switching from Ethernet to PPP, buffering in the PPP dial-in server rises quickly to a level and stabilizes at it. When switching from PPP back to Ethernet, the high bandwidth is utilized instantly. The reason for this behavior is that the same congestion window size, around 5 packets, is optimal for both the PPP and the Ethernet link. If the network switch is between a PPP link and a WAN connection such as the one between OGI and Georgia Tech, with an optimal congestion window size of 18 packets (Fig. 5.7(c)), we expect that reset on SCP would make a difference. The slow-start policy following the reset helps figure out the new window size quickly. Otherwise, upon switching either the PPP server buffering would be unnecessarily high or overflow, or it would take a long time for SCP's steady-state policy to increase the congestion window size from 5 to 18.

However, our experiments show that resetting SCP helps keep the parameter estimations accurate. The base RTT for a $1500B$ packet over Ethernet is less than $1ms$, but it

is about half a second across a PPP link. If SCP is not reset, the base RTT estimation would stay at less than 1 millisecond after switching from Ethernet to PPP. According to the SCP steady-state policy in Equation 5.1, the severe under-estimation of base RTT would result in the calculated congestion window size much lower than it should be. A reset solves this problem by resetting the base RTT estimator and discarding the no-longer invalid estimation. Fortunately, in our experiments, the minimum steady-state congestion window size of $W_\Delta = 3$ is sufficient for exploiting the network bandwidth of both PPP and Ethernet LAN, so the under-estimation does not hurt the throughput much. We expect that the severe under-estimation would hurt throughput of a WAN session seriously, in which a big congestion window size is required for sufficient utilization the network bandwidth.

5.6 Discussion

In this chapter, we have presented SCP, an effective flow- and congestion-control scheme for real-time media streaming applications. SCP eliminates the unpredictability in latency caused by retransmission of lost packets. It uses positive acknowledgements to detect network congestion, and rate- and window-based policies to control the streaming flow effectively. During the start-up phase, SCP employs a slow-start policy to open the congestion window quickly. When the network is not congested, SCP's rate- and window-based policy ensures smooth streaming, sufficient use of the network bandwidth and low latency. It also enables a stable sharing of network bandwidth between multiple streams. Upon detection of network congestion, SCP backs off exponentially. SCP has mechanisms to handle the limited data rate and possible pauses in streaming. With the ability to reset its internal state and parameter estimators, SCP also has mobility awareness. Upon a network interface switch, such as between PPP and Ethernet, SCP can be reset, thus its invalidated states and estimations are discarded, and the slow-start policy is invoked to adapt to the new network environment quickly.

The SCP flow-control policies are designed and implemented with the software feedback toolkit. SCP is composed of guarded feedback policies for slow-start, steady-state,

congestion and traffic pause. The domains for these policies, associated events and guards, and meta-adaptation actions upon each event are identified. During a streaming session, a policy is invoked whenever the network condition falls into its domain. Switches between the guarded policies are triggered by the events indicating changes in network conditions. The estimators for various parameters and the policies are then implemented as feedback components. Building blocks from the feedback component class library are used whenever possible to expedite implementation. The tools in the toolkit then help instrumentation of the performance of SCP in real applications, and parameter tuning.

Experiments have been conducted to stream packets over typical network configurations, including PPP, a single subnet, across a single router (LAN), and across the long-haul Internet (WAN). The experiments demonstrated that SCP is able to stream data packets smoothly in all the configurations. The buffering delay of the stream is low and predictable. Multiple SCP streams are able to share the network bandwidth in a stable and in many cases fair manner. SCP streams are also able to share the network bandwidth with TCP streams across the same network links. These properties of SCP make it a promising protocol for real-time multimedia streaming across the Internet. The experiments also revealed that the residual buffering error in the base RTT estimation plays a significant role in determining the network bandwidth partitioning among multiple SCP sessions sharing the same network links. Thus minimizing the base RTT estimation error is an important part of future work.

Chapter 6

An Adaptive Real-Time Distributed Video Player

6.1 Introduction

This chapter presents an adaptive MPEG video player [10] for streaming compressed digital video and audio in real-time across the Internet. The Internet is characterized by great diversity in host processing speed and network bandwidth, the lack of a common clock, and wide-spread resource sharing. Consequently, the availability of resources such as network bandwidth and host CPU cycles changes dynamically and unpredictably. Similarly, parameters such as network latency and latency variation also change rapidly. With the emergence of mobile computing, the Internet is becoming even more dynamic. The bandwidth and latency of a mobile link may change significantly after each relocation. A mobile host may have its network interfaces switched dynamically, e.g., from Ethernet to wireless PPP or vice versa, causing network bandwidth and latency to change by several orders of magnitude.

To ensure that the video player works well in this highly dynamic environment, it is necessary for it to be adaptive. This adaptability is realized through extensive use of software feedback mechanisms for quality-of-service (QoS) control and client-server synchronization, as well as the use of SCP, a media streaming protocol described in Chapter 5. The feedback toolkit methodology and components greatly help the design and implementation of the QoS and synchronization feedback mechanisms in the player.

This chapter focuses on the design, implementation and experimental evaluation of

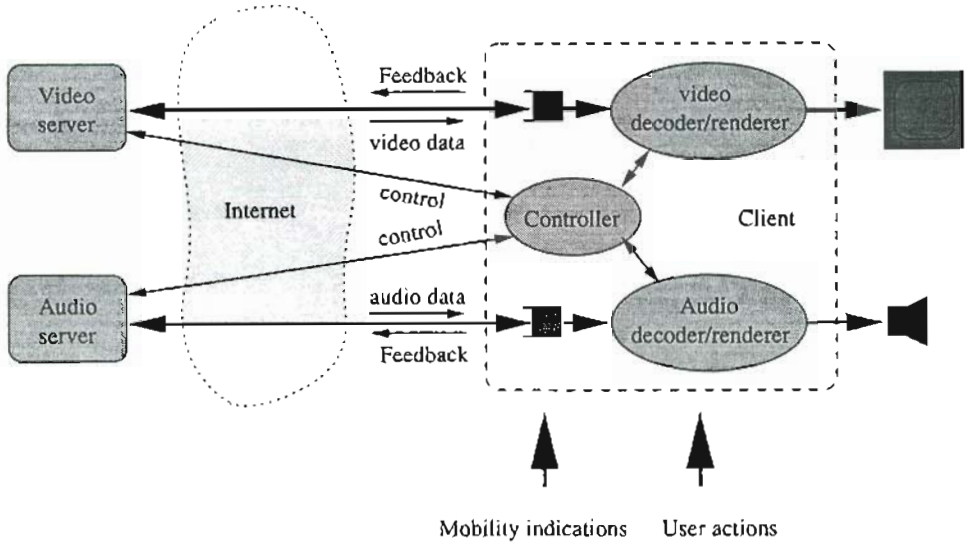


Figure 6.1: Architecture of the real-time adaptive distributed video player

the QoS and synchronization feedback mechanisms used in the player. A more detailed description of the player can be found in [10]. In the rest of this chapter, first the architecture of the player is briefly described. Then the design and implementation of the QoS control feedback is presented. Next, a section is devoted to the client-server synchronization feedback. Finally, experimental results are given to demonstrate the effectiveness of the feedback mechanisms used in the player.

6.2 System Architecture

The distributed video player has a client-server style architecture as shown in Fig. 6.1. It has a video server, an audio server and a client connected through a TCP/IP network (Internet). The video and audio servers store and retrieve compressed digital audio and video clips. The client is further composed of a video decoder-renderer, an audio decoder-renderer, and a controller. Reliable TCP connections are used for control messages between the video and audio servers and the client controller, while SCP connections are employed for streaming video and audio data from the servers to the client and for shipping feedback messages in the reverse direction. The video player supports VCR-style operations, such as play, speed change, random positioning, fast-forward and fast-backward.

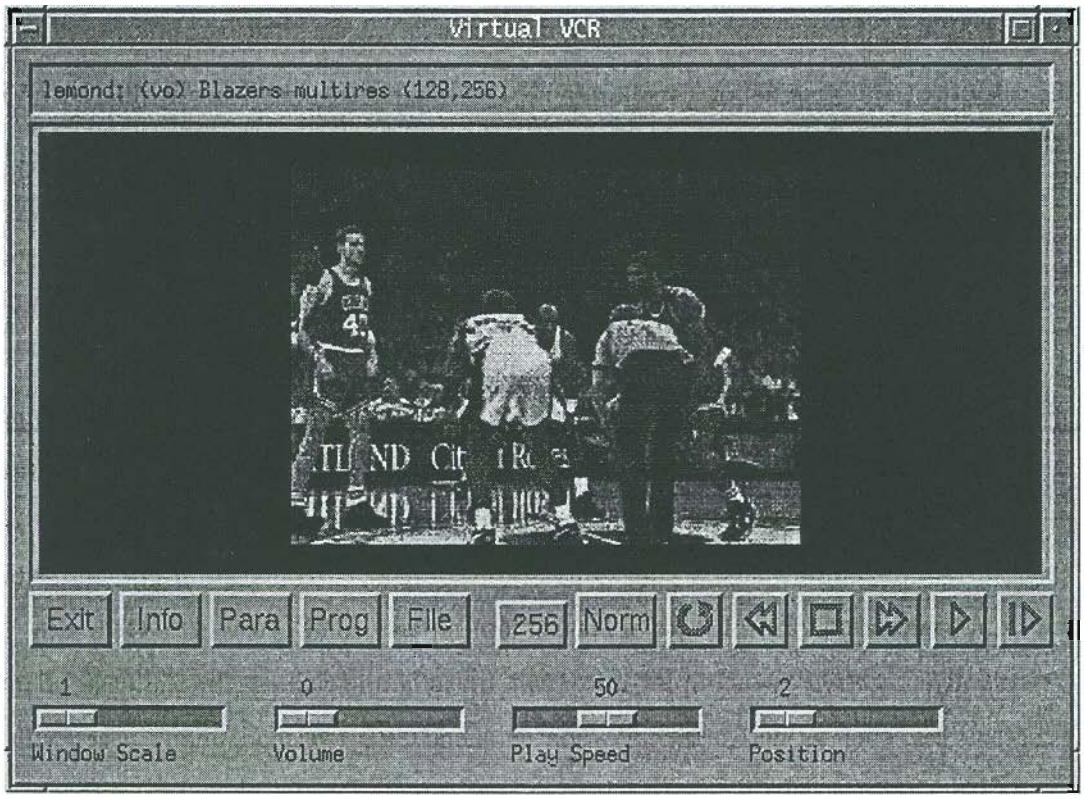


Figure 6.2: GUI of the real-time adaptive distributed video player

Through its graphical user interface (GUI) as shown in Fig. 6.2, the user can initialize the player with specified video and audio clips, start or stop playback sessions, change the playback speed, perform fast-forward or fast-backward, etc. The user can also specify preferences on video quality, such as trade-off between video resolution and frame rate, through a parameter panel of the GUI. This panel is displayed with a click on 'Para' button on the GUI.

During a playback session, video frames are fetched by the video server from its storage, streamed in real-time at a given speed through the network, and buffered, decoded and rendered by the client. Similarly, audio samples are fetched by the audio server, streamed through the network, and buffered, decoded and rendered by the client. The video server, video network connection, and video decoder-renderer form a video pipeline, within which the video frames flow. Similarly, an audio pipeline is also formed to process audio samples.

Several reasons make it advantageous for the player to support separate pipelines for

video and audio, instead of a single pipeline for multiplexed audio-and-video streams. The resource requirements for streaming video and audio are significantly different. Video usually consumes much more resources than audio. The users may have different requirements on video and audio quality. When there are not sufficient resources, different policies are needed to gracefully degrade video and audio quality. In particular, when resources are scare, the users may want to reduce the video quality first, or even disable one of the video and audio pipelines. The current version of the player supports MPEG1 video and μ-law audio streams. A clip can contain video only (video clip), audio only (audio clip), or both video and audio (video-and-audio clip). MPEG1 video streams have variable bitrates, with average bitrates ranging from $500Kbps$ to $2Mbps$. MPEG1 video streams also require powerful CPUs for software decoding. In contrast, μ-law audio has a $8bit$ sample size and a sample rate of 8000 samples-per-second. Very little CPU time is needed to convert $8bit$ μ-law samples into $16bit$ linear audio values. In the case of insufficient resources, such as network congestion or client CPU overload, many users will prefer to reduce the video quality first.

In the prototype player, synchronization between the video and audio streams is realized by a sequence-number-based mechanism implemented in the client [10]. A video stream consists of a sequence of video frames, each of which is tagged with a sequence number. An audio stream consists of a sequence of audio samples. We assume that, in a video-and-audio clip, the video and audio streams are recorded strictly synchronously. We refer to a contiguous subsequence of audio samples corresponding a video frame as an *audio block*. Therefore, there is a one-to-one correspondence between video frames and audio blocks. During real-time playback of a video-and-audio clip, the controller in the client synchronizes the video and audio streams by rendering video frames and audio blocks in pairs. Per-pipeline client-server synchronization feedback mechanisms are then used to maintain appropriate buffer-fill-levels at the client side for individual pipelines.

As a research prototype, the main purpose of the adaptive video player is for investigating the adaptation mechanisms for presentation quality and client-server synchronization. Since MPEG1 video imposes a much bigger challenge than μ-law audio, most effort has been payed in designing, implementing and experimenting feedback mechanisms for the

video pipeline. For the audio pipeline, client-server synchronization is implemented, but quality adaptation in the face of insufficient resources is left for investigation in the future. In the rest of this chapter we will focus only on the video pipeline.

In our video player, video streams can be played at variable speed. *Play speed* is a parameter indicating how fast the video is played relative to the speed at which the video was recorded (normal speed). For example, we can have the play speed at 0.5, 1.0, or 2.0 times the normal speed. When the normal speed is known in terms of frames-per-second (fps), e.g., $30fps$ for NTSC video, then the play speed can also be represented in frames-per-second (fps). For NTSC video, 0.5, 1.0, or 2.0 times the normal speed correspond to $15fps$, $30fps$ and $60fps$, respectively. The client of our video player renders a video stream in real-time at a given speed by mapping its logical time (defined by video frame sequence number) into its system time (real time, in seconds). Suppose the system time at which frame(i) is displayed is T_i, and the current play speed is $Pfps$, then the time at which frame($i+1$) is played is $T_{i+1} = T_i + \frac{1}{P}$. The video server also maps the video stream's logical time into its own system time during the retrieval and streaming of video frames. It is possible that the clocks in the server and client are not synchronized. This problem will be addressed by a synchronization feedback mechanism described later in this chapter.

The video player plays video with multiple spatial resolutions. A source video clip may be compressed into multiple MPEG files at different resolutions. These files are treated as components of the same video stream, referred to as a multi-resolution video stream. During a playback session of a multi-resolution video stream, based on the current resource availability and the user preference, the player selects the appropriate resolution automatically and dynamically.

In the case that a stage[1] in the video pipeline becomes overloaded, it independently drops excessive frames, either randomly, or based on the local information it has gained from the video frames already processed. One metric to measure the actual QoS level of a video playback session is *display frame rate* — number of frames-per-second displayed by

[1]The source (server) stage is an exception. As to be further discussed in Section 6.3, the server drop video frames intelligently.

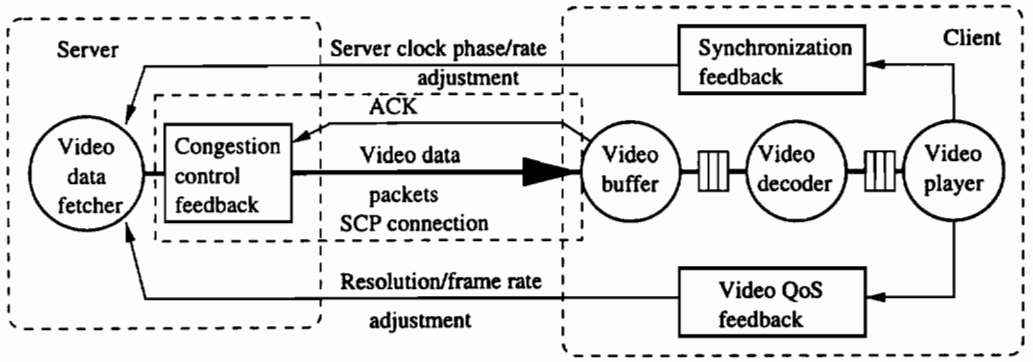


Figure 6.3: Feedback systems in the real-time adaptive distributed video player

the client. Display frame rate should not be confused with play speed, which can also be specified in frames-per-second. A valid display frame rate is always equal to or lower than the current play speed, and the difference between them is the aggregate frame drop rate caused by all the stages. For example, Suppose in a playback, the play speed is $Pfps$, and the display frame rate is $Ffps$ ($0 \leq F \leq P$), then the frame drop rate is $(P-F)fps$, and $\frac{P-F}{P} * 100\%$ of all frames are dropped by the player.

The user can express a preference on various aspects of the video quality. Example user preferences express trade-offs between video spatial resolution, frame rate (timing resolution) and signal-to-noise ratio, parameters such as acceptable maximum or minimum frame rates, or even trade-offs between video quality and costs. In the prototype player, the user can specify the maximum display frame rate, and the trade-off between spatial resolution and frame rate. The latter is accomplished by specifying a scale-up frame rate threshold and a scale-down frame rate threshold. When the display frame rate passes the scale-up threshold, the player switches to the next higher resolution. Similarly, when the player can not meet the scale-down threshold, it switches to the next lower resolution. The player does not reserve resources to enforce guarantees on video quality. Instead, through the feedback mechanisms for video quality and client-server synchronization, it preserves real-time playback while accommodating the user preferences with the currently available resources in a best-effort and adaptive manner.

As shown in Fig. 6.3, the video player effectively adapts to changes in the network

and client-server hosts through extensive use of toolkit-based software feedback mechanisms. SCP, the media streaming protocol described in Chapter 5, is used in the network connection from the server to the client to ensure efficient streaming of the video data and avoid network congestion. Above SCP, the player employs two end-to-end feedback mechanisms: QoS feedback to select the appropriate video resolution and frame rate; and client-server synchronization feedback to minimize client-side video data buffering while ensuring that the buffer does not under-flow.

The video player has mobility awareness [20]. Different types of network interfaces and links, such as such as Ethernet, WaveLAN, and wired or wireless modem links, have totally different characteristics such as link bandwidth and transmission latency. For example, a typical wireless modem link has a bandwidth of $9.6Kbps$, while the bandwidth of an Ethernet link is $10Mbps$ or even $100Mbps$. The speeds of these two types of links are different by three orders of magnitude. A mobile host may have multiple network interfaces of different types. When it is moved from one environment to another, such as being docked or un-docked, while still connected to the network, dynamic switching between interfaces happens, and results in sudden and large changes in available bandwidth and network latency. In some cases, such as when switching between PPP and Ethernet, the IP addresses or subnet number also needs to be changed.

The player uses mobility indication events to trigger meta-adaptation operations of the feedback mechanisms. For example, upon receipt of an event indicating a switch in network interfaces during a playback session, the player re-establishes the control and data connections between the client and the servers and continues the playback session from where it was interrupted. Upon these events, all the feedback mechanisms, including SCP, the QoS feedback and the synchronization feedback, perform meta-adaptation to reset themselves to discard no-longer valid states, and then switch to slow-start policies to quickly figure out the new network parameters such as bandwidth, latency and variation, and adapt the video quality to the new network environment.

6.3 Adaptive QoS Control Feedback

In this section, we discuss the QoS feedback which controls the video quality. The QoS control feedback is an end-to-end mechanism that adapts the video resolution and frame rate to the currently available resources as well as user preferences. It continuously monitors the rate at which the client displays video frames, and accordingly adjusts the resolution and the rate at which the video server streams frames in the future. Since the feedback is from the ultimate sink to the ultimate source of the video pipeline, it adapts to the availability of all resources used by all the pipeline stages. The feedback also reacts to user actions and other explicit events, such as network interface switches, using guard-based meta-adaptation.

6.3.1 Video Pipeline QoS Model

As shown in Fig. 6.1, the video pipeline, with which the QoS feedback works, includes the video server, the network connection and the video client. Each stage may be a sequence of finer-grain sub-stages. For example, the server may have two processes, one fetching frames from disks, and the other pacing the fetched frames out to the network. Similarly, the client may have separate processes for decoding, dithering and displaying video frames.

During a playback session, video frames are streamed in real-time from the server, through the network, and buffered, decoded and rendered by the client. If the server needs to drop video frames, either when it observes network congestion or when instructed by the client through the QoS feedback, it drop frames intelligently. It only sends out a P or B frame[2] if all the reference I or P frames are already sent out. If any stage except for the server is overloaded, it independently drops all the excessive frames, either randomly, or based on the local knowledge it has gained from the video frames already processed. No dropped frames are retransmitted. The server and client have their own clocks, and stream out or render frames according to their own timing reference. Due to the client-server clock

[2]An MPEG video stream consists of a sequence of I, P or B frames. An I frame, called intra-coded frame, is self-contained. A P frame, called predictive frame, contains the difference between the frame it encodes and a past I or P reference frame. A B frame, called bidirectional frame, contains the difference between the frame it encodes and the interpolation between two reference I or P frames (one in the past and one in the future).

asynchrony and latency introduced by the server and network stages, a frame may already be late when it arrives at the client. These late frames are also dropped by the client. To avoid this problem, the server works ahead of the client for an appropriate amount of time, and a client-side buffer is used to hold the frames coming from the network before decoding and rendering. It is the job of the synchronization feedback, further discussed in Section 6.4, to ensure that the server work-ahead time is as small as possible, with the condition that the client-side buffer does not underflow.

For a video pipeline with the behavior above, the bandwidth of the bottleneck stage in terms of frame rate defines the effective bandwidth of the whole video pipeline. The pipeline model for the QoS feedback is similar to that shown in Fig. 3.14 of Section 3.6. Since the data units processed by the video pipeline are video frames that are variable in size and decompression processing overheads, only average effective bandwidth is meaningful. Furthermore, the effective bandwidth is associated with video stream parameters such as spatial resolution and compression ratio, so it may change after each switch in resolution or change in play speed, and the video player needs to react accordingly.

6.3.2 The QoS Control Feedback Policies

The goal of the QoS control feedback policies is to make the best use of the currently available resources in the video pipeline in accordance to user preferences, which include a maximum frame rate and a trade-off between video resolution and frame rate. There are two policies, one for frame rate adaptation, and the other for resolution adaptation.

Frame-Rate Adaptation Policy

The frame-rate adaptation policy detects and traces the effective bandwidth of the pipeline as well as conforming to the user-specified maximum frame-rate. This policy is similar to the packet-loss feedback policy discussed in Section 4.2.1. It keeps a preset rate of frame dropping in the pipeline, and works through iterations. At each step, the policy measures the current display frame-rate, and generates a target server-streaming frame-rate for the next step — the next-step server-streaming frame-rate — by adding an incremental coefficient to the current display frame rate. However, if the resultant rate is greater than

the user-specified limit, then the latter is used instead as the next-step server streaming frame rate.

Suppose that the frame rate incremental coefficient is Δ, and the user-specified maximum frame rate (limit) is γ_{max}. Also suppose that at step k, the measured display frame rate is $\hat{\gamma}_k$, then the next-step server streaming frame rate λ_{k+1} is:

$$\lambda_{k+1} = min(\hat{\gamma}_k + \Delta, \gamma_{max})$$

Spatial-Resolution Adaptation Policy

The resolution adaptation policy implements the user preference on the trade-off between spatial resolution and frame rate. Suppose a multi-resolution video has N predefined resolutions, denoted in ascending order of resolution as $S(1), \ldots, S(N)$. In the prototype video player, users specify their resolution preferences by a pair of parameters: scale-up frame rate threshold (high water mark) γ_{up} and scale-down frame rate threshold (low water mark) γ_{down} ($\gamma_{down} < \gamma_{up}$). The resolution-adaptation policy works through iterations. At step k, depending on the current resolution $S_k = S(i)$ and measured display frame rate $\hat{\gamma}_k$, the policy takes one of the following three actions to set the next-step resolution S_{k+1}:

1. If $i < N$ and $\hat{\gamma}_k \geq \gamma_{up}$, then scale up: $S_{k+1} = S(i+1)$.
2. If $i > 1$ and $\hat{\gamma}_k \leq \gamma_{down}$, then scale down: $S_{k+1} = S(i-1)$.
3. Otherwise, no switch in resolution is performed: $S_{k+1} = S_k = S(i)$.

After each resolution switch, the effective bandwidth of the pipeline at the new resolution is not immediately known. One way to predict it is based on the current pipeline bandwidth estimation and the ad-hoc relationship between pipeline effective bandwidth and resolution in the context of the video stream in question. This prediction involves mapping from user-level QoS to resource requirements [57], and is complicated. A much simpler way, which is used by the resolution-adaptation policy in the video player prototype, is to detect the new effective bandwidth by injecting frames at a high enough rate to the pipeline. After scaling down, the policy assigns a next-step server streaming frame rate $\lambda_{k+1} \leftarrow \hat{\gamma}_k$, while after scaling up, $\lambda_{k+1} \leftarrow \gamma_{max}$. In our video player, this approach

does not cause network congestion, since SCP is used underneath this QoS feedback for network flow and congestion control.

To prevent the resolution adaptation policy from oscillating between neighboring resolutions, it is necessary to introduce some hysteresis. This hysteresis implies that the gap between γ_{down} and γ_{up} should be large enough so that, if a video pipeline has an effective bandwidth of γ_{up} at resolution S_i, its effective bandwidth at resolution S_{i+1} should be larger than γ_{down}. How to set γ_{down} and γ_{up} so that hysteresis exists is related to the specific MPEG compression format and parameters used, as well as the video content itself, and will be further exploited in the future.

Interaction between the Two Policies

The frame-rate adaptation policies can be viewed as a guarded feedback component with the resolution policy as a guard on its performance (display frame rate). At each iteration step k, the estimated display frame rate $\hat{\gamma}_k$ is applied to the resolution policy, which determines if the display frame rate is out of the range specified by the user. If either the higher or lower frame rate water mark is crossed, the guard is triggered, and meta-adaptation action is performed by the resolution adaptation policy to change the video resolution, and to send a high next-step frame rate to the server in order to quickly re-detect the pipeline effective bandwidth at the new resolution. Only in the absence of a invocation of the resolution policy, the frame-rate adaptation policy is invoked to compute a next-step server-send frame-rate based on the current display frame rate. The final output of the overall QoS feedback mechanism consists of a next-step resolution S_{k+1} and frame rate λ_{k+1}.

6.3.3 The QoS Control Feedback Architecture

The QoS (resolution and frame-rate) control feedback policies adapt video quality to changes in the availability of various resources. Time-based lowpass filtering is needed in estimation of display frame rate. Upon radical changes in pipeline conditions such as available bandwidth, it will take a while before the policies detect the changes and react. Thus the combined QoS feedback policies assume a domain with the property of continuity

Event	Meta-adaptation action
Session start	The feedback is initialized.
Session stop	The feedback is stopped.
Playback speed change	Parameters such as frame rate limit are changed. The display frame-rate estimator and feedback policies are reset, and the new pipeline effective bandwidth is detected by having the server to stream at the maximum frame rate.
Network interface switch	The rate estimator and feedback policies are reset, and the new pipeline effective bandwidth is detected. SCP is also reset in order to detect the new network bandwidth.

Table 6.1: Events guarded by the QoS feedback and meta-adaptation actions

in the transition of video pipeline conditions. On the other hand, radical changes (jumps) in pipeline conditions can happen as a result of explicit events caused by the client, such as switches in client network interfaces or changes in play speed. The dynamics of the video pipeline can be viewed as a domain composed of multiple sub-domains with gaps between them. All the sub-domains are continuous and associate with the same combined QoS feedback policies, except that they have different state values. In order for the overall QoS feedback to react quickly to jumps in pipeline conditions, events indicating the jumps are guarded, and trigger meta-adaptation by the overall QoS feedback, to detect the new conditions and update the state of its feedback policies.

The QoS feedback adapts to the explicit events in the form of guard-based meta-adaptation. Guards are placed around the feedback policies to guard for these events. When a guard is triggered, the feedback reacts by changing parameters or updating its states. Table 6.1 shows the events guarded and the meta-adaptation actions taken upon detection of them. Upon a change in play speed or a switch in network interface, the frame rate estimation and feedback policies are reset, and the server is requested to stream at a high frame rate so as to quickly detect the new and unknown pipeline effective bandwidth. This high frame rate will not cause network congestion due to the SCP protocol used. Upon a network interface switch, SCP for the client-server connection is also reset, so that the new network bandwidth is quickly determined by the slow-start phase which follows.

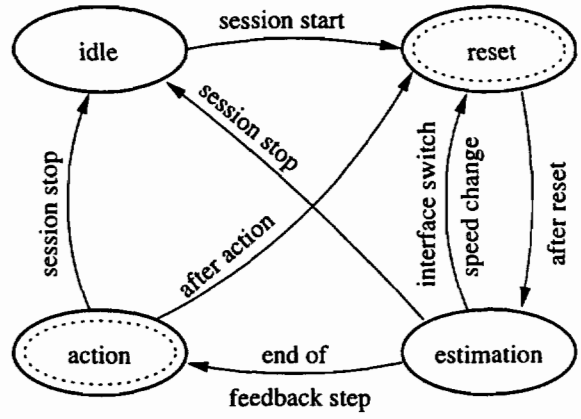


Figure 6.4: States of the QoS feedback and their transitions

Taking into consideration the QoS feedback iteration and meta-adaptation, the feedback has the following four states.

1. *idle* The QoS feedback is in this state when there is no active playback session.
2. *reset* This state is transient. After reset, the feedback enters state *estimation*.
3. *estimation* In this state, the client-display frame-rate is being estimated.
4. *action* At the end of each step, the QoS feedback enters this state, and sends a feedback message with the next-step resolution and frame-rate to the video server. This state is also transient. After each action, the feedback enters the *reset* state.

Transitions among these states upon the events are shown in Fig. 6.4. In this figure, dotted double ovals represent transient states. Normal states are shown as solid single ovals.

6.3.4 Implementation of the QoS Control Feedback

Overall Structure

The QoS feedback is implemented as a composite feedback component built on top of various building blocks from the software-feedback-toolkit component-library. With an overall structure shown in Fig. 6.5, it has a display-frame-rate estimator, a component implementing the two QoS adaptation policies, an action component to generate feedback messages, as well as two gates (*Gate*1 and *Gate*2, both of which are *timeGate* from the component library) for meta-adaptation and feedback state transition. Upon display of

each frame, the time when the frame is displayed is input to the QoS feedback component. When *Gate1* is open, frame display time samples are passed to the *rate estimator* for estimation of the average display frame rate. Later when *Gate2* is open at the end of each iteration step, the estimation is passed to the policy component, which produces the next-step resolution and frame rate. Finally the action component sends feedback messages to the video server. The QoS feedback exports several parameters, such as the parameter for the frame-rate estimator, delays for the two gates, the resolution scale-up and scale-down frame rate thresholds, frame rate limit, etc. Via the parameter ports, the application is able to specify the user preference, and to adapt the feedback to its current environment. Upon detection of explicit events, meta-adaptation is done through changing the parameters, controlling the delay components and resetting the whole feedback.

Estimation of Display Frame Rate

The display frame rate estimator is a composite feedback component with a structure shown in Fig. 6.6. It takes in the display time of each frame, computes the interval between the current frame and the previous one with a difference filter, applies the raw interval measurements to a lowpass filter to get an average interval, and inverts it into an average display frame rate. The parameter of the lowpass filter is exported for adjustment.

The QoS Feedback Policy Component

The QoS feedback policy component is itself a composite feedback component, shown in Fig. 6.7. The average display frame rate is input to the resolution policy sub-component *ResolutionPolicy*, which implements the resolution adaptation policy described in Section 6.3.2. If resolution adaptation happens, *ResolutionPolicy* generates the next-step resolution and frame rate, and does not invoke the frame rate adaptation policy. Otherwise, it generates a next-step resolution equal to the current one, and passes the average display frame rate through to the rate adaptation policy sub-component, *RatePolicy*. *RatePolicy* implements the frame-rate adaptation policy stated in Section 6.3.2 with two building blocks. Since it is not feasible to implement continuous adjustment on the server streaming frame rate, the next-step frame rate generated by either of the policies is quantized at

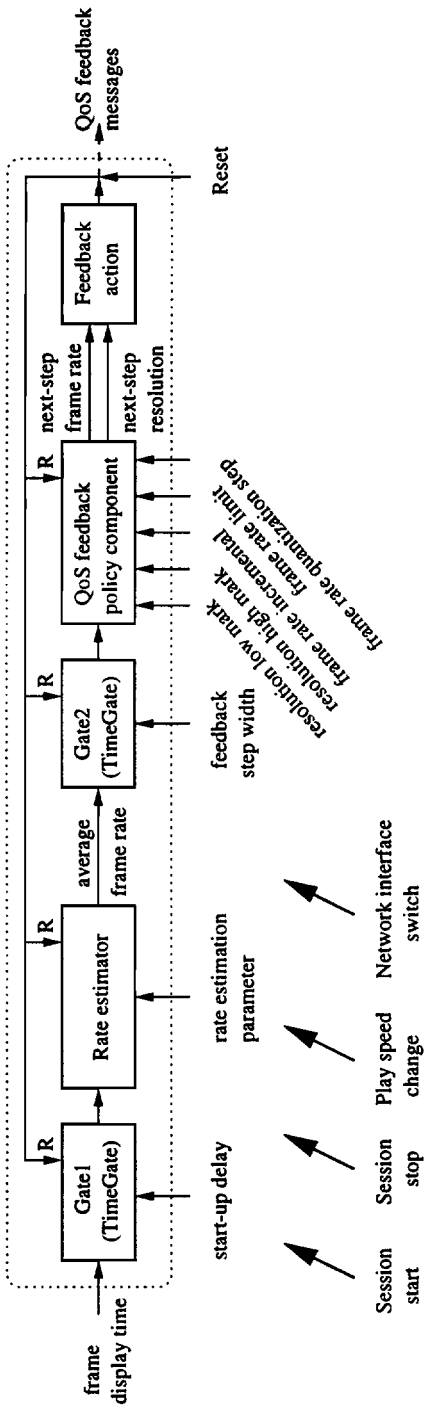


Figure 6.5: Overall structure of the QoS feedback implemented as a composite feedback component

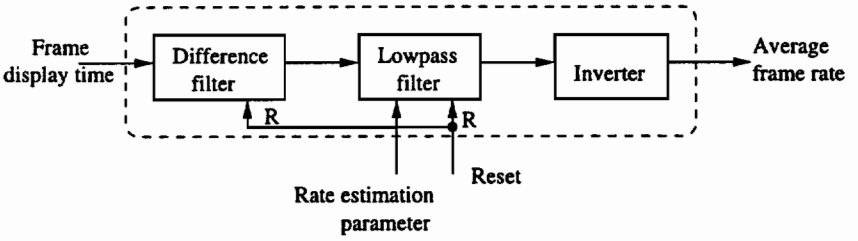


Figure 6.6: Implementation of the QoS-feedback display-frame-rate estimator

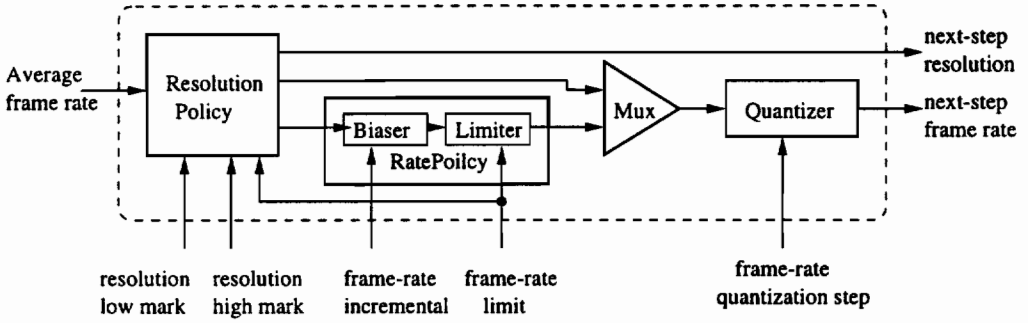


Figure 6.7: Implementation of the QoS-feedback policy component

a resolution specified externally by the player or the user. Finally, the next-step resolution and quantized frame rate are sent to the action component.

Generation of QoS Feedback Messages

Each iteration step of the QoS feedback has a duration specified by the component *Gate2*. At the end of each step, the feedback policies are invoked to generate a next-step resolution and frame rate. These next-step parameters are packaged by the *feedback action* component and are sent to the video server. Finally, the QoS feedback is reset in preparation for the next step.

Meta-Adaptation and Feedback State Transition

Meta-adaptation and state transition of the QoS feedback is performed by manipulating its parameter ports and reset port. During initialization, all the parameters are specified, and the feedback is reset. After that, upon display of each frame, a raw display frame rate measurement is sent to the input port of the QoS feedback. Components *Gate1*, *Gate2* and *Action* control the transition between different states. After a specified delay, *Gate1*

opens, pushing the feedback into the *estimation* state. After another delay, *Gate2* enables, transferring the state to *action*. At this moment, the *feedback action* component sends out a feedback message and resets it, transferring its state to *reset*. At any time upon detection of either a speed change or a network interface switch, the QoS feedback is also reset immediately.

Delay component *Gate1* is used to accommodate the non-zero round-trip-time (RTT) from the client to the server and back. After each feedback message is sent, it will take some time before the effect of the feedback action can be propagated back to the client, and there may already be frames in the pipeline at different resolutions. These frames will be passed through the pipeline, and rendered at the client, but should be ignored by the QoS feedback. Otherwise, inaccurate measurements may be generated and inaccurate actions may result, such as variable-speed playback, fast-forward, fast-rewind, and random positioning. Through its parameter port, *Gate1* is dynamically set with a delay long enough to cover the RTT.

Changing the delay time of the delay component *Gate2* changes the interval of interaction steps, and thus the responsiveness of the feedback, thus helps make the video player more adaptive. For example, in a playback session, if the video display frame rate is high, then it needs only a short time to get a reliable frame rate estimation. On the other hand, if the display frame rate is very low, due to either thin network links, or a slow client, then a longer time is needed before a reliable frame rate estimation can be generated. In another example, upon explicit events such as switching from Ethernet to PPP, the amount of resources (in our example the network bandwidth) reduce in several orders of magnitude, the player is required to detect the change and reduce the video resolution and frame rate quickly in order to avoid excessive network congestion. Otherwise the player would become virtually stalled while PPP is slowly shipping large-sized frames. In all these cases, the player can be made adaptive by giving an appropriate *Gate2* delay time.

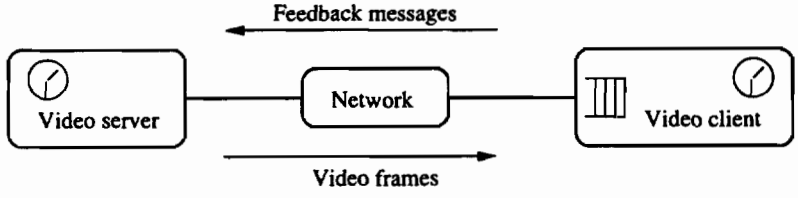


Figure 6.8: The video pipeline of the player concerning client-server synchronization

6.4 Adaptive Client-Server Synchronization Feedback

While the QoS feedback maintains that the video pipeline be fully exploited but not overloaded, the goal of the client-server synchronization feedback is to minimize client-side buffering while ensuring smooth playback. The synchronization feedback keeps the buffering to a minimum level with the condition that the buffer does not underflow, so that no frames arrive late at the client. It also adapts the amount of buffering to changes in pipeline conditions. The feedback reaches its goal by synchronizing the clock of the video server to that of the client, and maintaining an appropriate amount of time that the server works ahead of the client.

6.4.1 Video Pipeline Synchronization Model

As far as client-server synchronization is concerned, the video pipeline can be viewed as one with the server (source) and client (sink) having independent clocks, and with a buffer in the front of the client, as shown in Fig. 6.8. Other than this single client-side buffer, little buffering is provided between other neighboring stages or sub-stages. During a playback session, video frames are streamed in real-time along the video pipeline stages. The video server begins streaming frames upon receipt of a request from the client containing a start position and play speed. It streams out video frames at the given play speed (frame rate) according to its local real-time clock. Each frame is tagged with an incremental sequence number, and a frame dropped by the server results in a gap in the sequence number. The server can be viewed as having a logical clock that is reset at the beginning of a playback session, and then runs at the current play speed. At each tick a next frame is tagged with the current clock value (sequence number) and sent out to the network, or dropped as a

result of QoS feedback. The client also has a logical clock mapped to its own local real-time clock. After sending the request to the server, the client waits until the client-side buffer fills up to a specified level, then resets and starts its logical clock at the current playback speed. At each tick of its clock, the client reads video frames from its buffer, drops all the late frames (whose sequence number are smaller than the logical clock value), and renders the current frame (whose sequence number equals the logical clock value) if it is available in the buffer. In the rest of this chapter, when there is no confusion in the context, "clock" is simply used in the place of either real-time clock or logical clock.

Since the server and client (real-time) clocks are independent, the server logical clock may become late compared to that of the client due to their rate mismatch or clock drift, resulting in buffer underflow. In this case, some or all frames will already be late when they arrive at the client and will be dropped, causing degraded video quality. If the server (logical) clock is too far ahead of the client one, the fill level of client-side buffer will be too high, or the buffer may even overflow. Excessive buffering results in the player being sluggish in response to user actions such as speed changes, and causes excessive memory consumption. Buffer overflow results in packet drops and reduced video quality. Other events, such as changes in workload and switches in network route or interface, result in either gradual or radical changes in the latency or latency variation of the video pipeline stages, especially the network. These events also cause changes in the buffer-fill level, or in the amount of buffering required.

The synchronization problems above can be solved by adjusting the server logical clock in terms of rate and phase. The client can request the server to skip a certain number of frames, or stall for a given number of frames of server time. It can also ask the server to stream out frames at a slightly faster or slower play speed, while keeping the client-side play speed unchanged. Phase adjustment causes the buffer-fill level to jump, while rate adjustment makes it change gradually. Since the adjustments cause a frame with a certain sequence number to be streamed earlier or later, they can be viewed as adjustments to the server logical clock. In the video player, it is the responsibility of the client-server synchronization feedback to adjust the server logical clock based on the observation of the difference between the server and client's logical clocks. The goal is to maintain an

optimal target buffer-fill level, and adapt it to changes in the video pipeline conditions.

6.4.2 The Synchronization Feedback Policies

The synchronization feedback has two policies: a target buffer-fill level adaptation policy to determine the optimal buffer-fill level, and a server clock adaptation policy to adjust the server logical clock to maintain the target buffer-fill level.

Target Buffer-Fill-Level Adaptation Policy

The target buffer-fill level is determined by the magnitude of variation (or jitter) in the actual buffer-fill level. The more variation presents, the more buffering is needed to reduce the chance of buffering underflow. An exponential adaptation policy is adopted for adjusting the target buffer-fill level. Suppose the current target buffer-fill level is λ, and the buffering variation estimation is ϵ, then the new target level λ_{new} is defined by following formula:

$$\lambda_{new} = \begin{cases} 2\epsilon & \text{if } \epsilon \geq \lambda \text{ or } \epsilon \leq \frac{\lambda}{4} \\ \lambda & \text{otherwise} \end{cases}$$

With this policy, the target buffer-fill level is always greater than the buffer variation level. Each time the target level changes, it either at least doubles (in the case of an increase) or at least halves (in the case of a decrease). The policy maintains that $\frac{\lambda}{4} \leq \epsilon \leq \lambda$, and implements a hysteresis to prevent the target value from oscillating. This hysteresis occurs because after each adjustment, the new target is two times the current buffering variation. Therefore, the variation needs to be either doubled or halved before another adjustment happens.

Server Clock Synchronization Policy

The server clock synchronization policy is similar to the phase-lock loop discussed in Section 2.5.4 and Section 3.5, which adjusts the VCO based on the phase difference between it and the reference clock. However, since the goal for the policy here is to maintain the target buffer-fill level instead of a strict synchronization between the server and client clocks, the deviation of the actual buffer-fill level from the target level instead of the clock

phase difference is used to control the server clock. Also in the video player, it is sufficient to keep the actual buffer-fill level in the neighborhood of the target level instead of having the former converge to the latter exactly. To avoid the overhead associated with the periodic and frequent feedback actions as in the PLL, low and high water marks around the target level are introduced. The synchronization policy is invoked only when the low or high water mark is crossed. With these two water marks, in a stable environment, it is possible that after a start-up period, the synchronization feedback policy does not need to be invoked at all.

The synchronization policy adjusts the rate and phase of the server logical clock when the buffer-fill level hits either of the two water marks. Suppose that the low water mark, target buffer-fill level and high water mark are λ_l, λ and λ_h respectively, and minimum interval between feedback actions is T (to accommodate the client-server RTT). Also suppose that the current feedback step starts at time t_0 with a initial buffer-fill level $\hat{\eta}_0$. Beginning at time $t + T$, the estimated buffer-fill $\hat{\eta}$ is continuously compared against the two water marks. When either λ_l or λ_h is crossed at time t_1, the server logical clock rate and phase is adjusted as follows.

1. Phase: if $\hat{\eta} \leq \lambda_l$, then the server clock skips $\lambda - \lambda_l$ amount of time. Otherwise if $\hat{\eta} \geq \lambda_h$, the clock stalls for $\lambda_h - \lambda$ amount of time.
2. Rate: adjust the clock rate by $-\frac{\hat{\eta}-\hat{\eta}_0}{t_1-t_0}$. A negative value means acceleration, and a positive one means deceleration.

The goal of this policy is to bring the buffer-fill level close to the λ immediately after the feedback action, as well as to match the server logical clock rate to that of the client clock. Similar to the PLL, as long as T is sufficiently longer than the client-server RTT, the synchronization feedback will be stable and will maintain its target buffer-fill level independent of the actual RTT. Though a formal analysis will be be given in the thesis.

Interaction Between the Two Policies

The target buffer-fill level adaptation policy and the server clock synchronization policy work together to form a high-order feedback mechanism, with the former performing

meta-adaptation on the latter by changing its parameters. The target buffer-fill level λ is determined by the target level policy based on its estimation of the variation in buffering, and is used to calculate the high and low water marks λ_l and λ_h. A simple way to get the high and low water marks, as used in the prototype player, is to have $\lambda_l = \frac{1}{2}\lambda$ and $\lambda_h = \frac{3}{2}\lambda$. The target level λ and the two water marks λ_l and λ_h are then taken as parameters by the server clock synchronization policy.

6.4.3 The Synchronization Feedback Architecture

Similar to the QoS feedback, the synchronization feedback also reacts to explicit events in the form of guard-based meta-adaptation. The combined synchronization feedback policies expect a domain with continuity in pipeline state transition. On the other hand, the overall synchronization feedback has a domain composed of multiple continuous sub-domains. But there are gaps between these sub-domains. Upon events indicating jumps in network condition, the synchronization feedback performs meta-adaptation actions to change the parameters or state of the feedback policies. The events guarded and the meta-adaptation actions are listed in Table 6.2. Upon each change in play speed, the speed of both the server and client logical clocks is updated. The latency and variation estimators are reset, and the feedback starts a new iteration step. A switch in network interface results in a new client-server network connection with unknown and perhaps totally different properties, so the synchronization feedback is reset upon this event. All the previous estimations and calculations become invalid and are ignored. The meta-adaptation mechanism in the feedback also detects the possible overflow and underflow in the client buffer, and compensates by adjusting the server clock phase.

To represent the different stages in feedback policies and meta-adaptation, the synchronization feedback has following three states.

1. *idle* The feedback is in this state when there is no active playback session.
2. *reset* This state is transient. After reset, the feedback enters the ***estimation*** state.
3. *estimation* In this state, the buffer-fill level and variation are being estimated.
4. *action* At the moment when the buffer-fill level hits either water mark, the

Event	Meta-adaptation action
Session start	the feedback is initialized.
Session stop	The feedback is stopped.
Playback speed change	The buffer-fill level and variation estimator are reset, and the feedback starts a new iteration step.
Network interface switch	The feedback is reset to ignore the no-longer-valid previous states. Buffer underflow or overflow is also compensated upon radical change in network properties.

Table 6.2: Events guarded by the client-server synchronization feedback and meta-adaptation actions

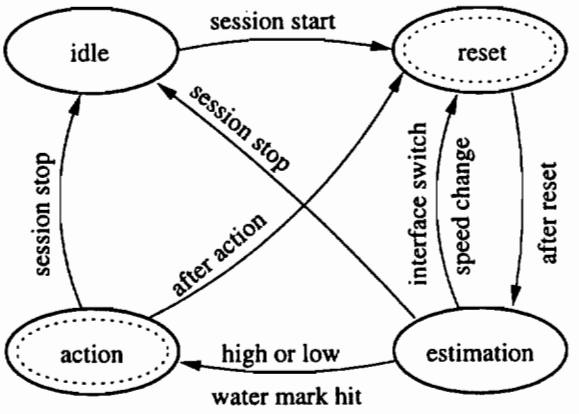


Figure 6.9: States of the synchronization feedback and their transitions

synchronization feedback enters this state and sends a phase and rate adjustment message to the server. This state is also transient, since after a feedback action, the feedback returns to the *reset* state.

Transition between these states is triggered by events such as crossing of the high and low water marks, feedback actions, as well as explicit events, as shown in Fig. 6.9.

6.4.4 Implementation of the Synchronization Feedback

Overall structure

Similar to the QoS feedback, the synchronization feedback is implemented as a composite feedback component on top of various building blocks from the toolkit component library. With a structure shown in Fig. 6.10, it consists of a lowpass filter for average buffer-fill level estimation, a buffering variation estimator, and components for the two feedback policies. Each time a video frame is received by the client, the current buffer-fill level is calculated and input to the feedback component. When *TimeGate* is open, samples are passed along the various components for estimation of buffer-fill level and variation, and invocation of the two policies. Several parameters, including the delay time and the lowpass filter parameters, are exported to the application for adjustment. Meta-adaptation is performed on the feedback component by updating its parameters and resetting it.

Estimation of Buffer-Fill Level

Literally, buffer-fill level means the amount of data in a buffer, either in terms of the number of bytes or in terms of the number of video frames. In our video player, the sizes of different MPEG frames differ greatly. Frame count reflects more accurately the timing and hence is better. But this measurement is still not the best for the real-time video player, in which the buffer-fill level is expected to accurately reflect the amount time the server works ahead of the client. One drawback is that when the buffer underflows, the frame count of zero does not correctly reflect the amount by which the server clock is late. Furthermore, the stages of the real-time player, including the server and the network, may drop frames, in which case the number of frames in the client-side buffer does not

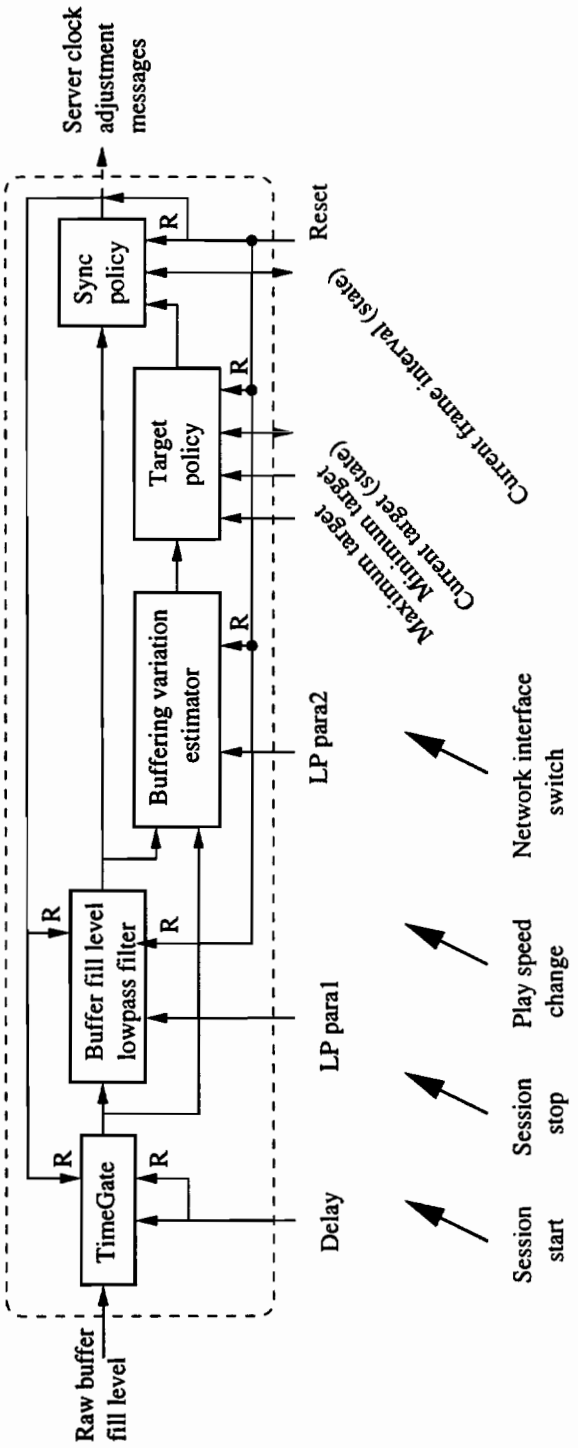


Figure 6.10: Overall structure of the synchronization feedback implemented as a composite feedback component

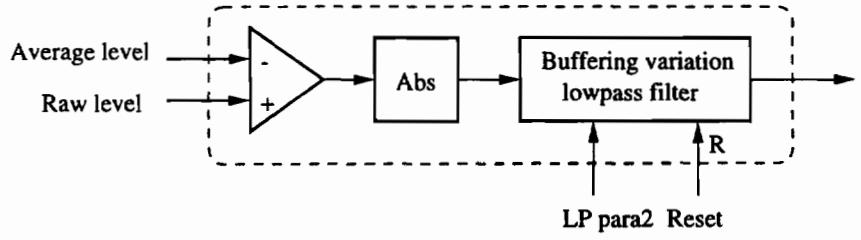


Figure 6.11: Implementation of the synchronization-feedback buffer-fill-level-variation estimator

correspond to the actual phase difference between the client and server clocks.

To avoid the drawbacks above, upon receiving each frame by the client, the buffer-fill level is measured as the time difference between the frame being displayed and the frame being received. This measurement is obtained by calculating the sequence number difference between the two frames, and multiplying it by the frame interval (the inverse of play speed). With this measurement, buffer underflow results in a negative measurement, and a large absolute value indicates that the server clock is well behind the client one. Frames dropped by the server or client do not reduce the accuracy of the measurement. Since the buffer-fill level measures the difference between the server logical-time and the client logical-time, it can also be viewed as the *server work-ahead time* as observed by the client.

The average recent buffer-fill level, or server work-ahead time, is estimated through a simple lowpass filter building block with an adjustable parameter.

Estimation of Buffering Variation

There are several alternative approaches to measuring variation [23, 29]. Conventional measurements include the variance and standard deviation, and have solid mathematical properties. Unfortunately, they involve costly computations such as calculation of squares and square roots. A cheaper alternative is mean deviation, which is the average of the absolute difference between the raw sample and the mean. The latter is usually a good approximation of the standard deviation, and is used in the player.

The buffering variation is measured by the *variation estimator* as the mean deviation of the input buffer-fill level sample sequence. As shown in Fig. 6.11, the estimator is

composed of three building blocks. It takes in the raw and average buffer-fill levels, computes the difference, gets the absolute values, and passes them to a lowpass filter for the final smoothed variation estimation.

Calculation of the Target Buffer-Fill Level

The *target policy* component implements the target buffer-fill level policy. It has a state port, through which the initial target level can be set, and the current target level can be saved or restored by the application in the case of meta-adaptation. To account for the latency incurred by the client stage, and to prevent the buffer from overflowing, minimum and maximum target levels are specified through the two parameter ports. These two limits set the output range of the *target policy* component.

Adjustment to the Server Clock

The *sync policy* component implements the synchronization policy, and generates feedback messages to the server when the current buffer fill level hits either of the water marks. It receives target level updates from the *target policy* component, and updates the high and low water marks accordingly. In the video player, the server logical clock jumps in terms of number of frames, and its rate is represented as an inter-frame interval (inverse of play speed). Thus the *sync policy* component keeps the current frame interval as part of its internal state, and exports it through its state port. Through this state port the application can specify the initial frame interval during initialization or upon speed change events.

Upon each feedback action, if the synchronization policy yields an exceptionally large server clock rate adjustment, such as $> 5\%$, then it is actually ignored, and only a phase adjustment is performed. The original synchronization policy in Section 6.4.2 adjusts both the server clock phase and rate on each feedback action, except for when explicit events are detected. The underlying assumption is that, except for when explicit events happen, the change in buffer-fill level is caused by client-server clock drift and is gradual. But in practice, some events not caused directly by the client, such as switches in the route of the network connection between the server and the client, also result in jumps in the

buffer-fill level. Since computer clocks are driven by crystals, the rate difference between any two of them should lie in a small range (such as $\pm \leq 1\%$). Whenever a big difference in clock rate is detected (such as $> 5\%$), it is likely that there are jumps in the buffer-fill level during the current estimation period, thus the rate adjustment is invalid and should be ignored. Only the phase adjustment part is sent to the server.

To ensure that the video stream is smooth when the server clock phase is adjusted, when the server receives a request to skip frames, it does not actually skip the given number of frames. Instead, it speeds up its logical clock at a rate much larger than any possible clock drift rate (such as 20% higher than normal rate), until the requested number of extra frames have been streamed out. This speed-up causes the client-side buffer to fill up gradually until it crosses the low water mark and approaches the target level. During this speed-up period, the synchronization feedback may react repeatedly by requesting to skip a decreasing number of frames, but the rate adjustment is ignored since it is too high to be considered clock drift. The server clock will not be over-compensated, since the server does not accumulate the phase adjustment requests. With this scheme, if further implicit events happen, their effects are also compensated appropriately.

Meta-adaptation and Feedback State Transition

Meta-adaptation and feedback state transition are implemented through manipulation of the parameters, state and reset ports of the synchronization feedback component. The parameters, such as the lowpass filter parameters and limits on target buffer-fill level, can be set at any time through the parameter ports. During initialization, the feedback is reset, and the target buffer-fill level and frame interval are set to default values. After a specified time, *TimeGate* opens, and buffer-fill level samples are passed to various components for buffer level and variation estimation as well as target buffer-fill level calculation. Whenever the *sync policy* component detects that the buffer-fill level hits either water mark, it takes a feedback action by sending clock adjustments to the server, and resetting *TimeGate* and filters for buffer-fill level and variation estimation. Resetting these sub-components brings the feedback to a new iteration step. After each change in playback speed, the feedback component is reset, and a new frame interval derived from the new play speed

is set through its frame-rate interval state port. The current target buffer fill level is preserved by reading from its state port before reset, and writing back to the state port afterwards. After each switch in the network interface, the feedback is reset with a default target buffer-fill level, since the existing target level is no longer valid.

Similar to the *Gate1* in the QoS feedback (Fig. 6.5), *TimeGate* in the synchronization feedback defines a minimum interval between feedback actions to accommodate the non-zero client-server round-trip time. Upon each feedback action or explicit event, the gate is closed for a certain amount of time as specified through its parameter until the effect of the feedback action reaches the server and propagates back to the client.

6.5 Experimental Results

6.5.1 Player Configuration for the Experiments

A prototype of the adaptive distributed MPEG video player has been implemented [8], and experiments have been conducted to evaluate the QoS and synchronization feedback mechanisms. In the prototype, the control channels between the server and the client are reliable TCP connections, while the data channels, including both the server-to-client direction for data streaming and the client-to-server for feedback message, can either be raw UDP, SCP, or TCP. As shown in Fig. 6.12, the configuration for the experiments has a $100MHz$ Pentium PC notebook *anquetil* as the client, and two other workstations, *lemond* and *virgo* as servers. The server *lemond*, an HP 9000/712 running HP-UX, is a file server at the CSE department of OGI. The connection between *lemond* and *anquetil* can be a $10Mbps$ Ethernet link or a $28.8Kbps$ PPP link. The workstation *virgo* is a SUN Sparc running SunOS residing at Georgia Institute of Technology (Georgia Tech), more than 20 Internet hops away from the client.

Two video clips are played in the experiments. These clips are from the same video source, and compressed at different resolutions. The lower-resolution clip has a resolution of 256×192 pixels, and the higher-resolution one has a resolution of 128×96 pixels. Both clips are compressed at 30 frames-per-second (fps), with MPEG picture group pattern IBBPBBPBBPBB. The size of individual video frames in these two clips are plotted in

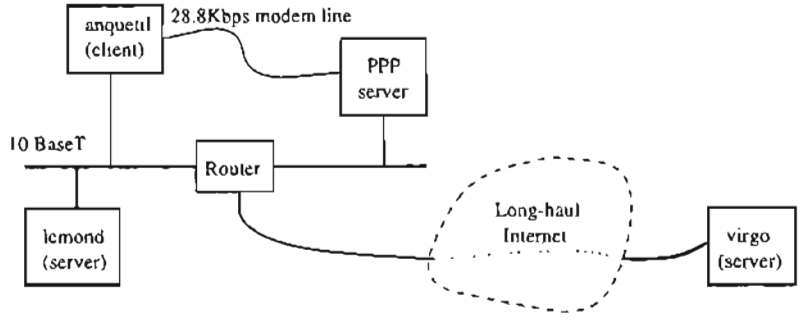
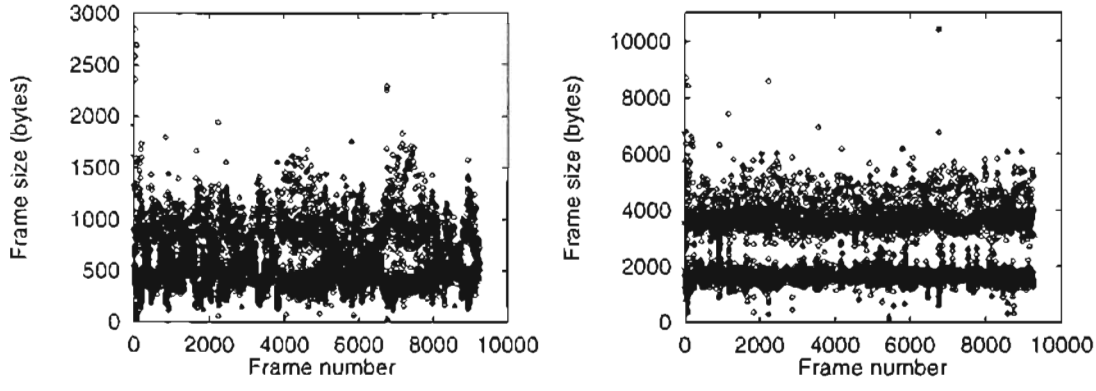


Figure 6.12: Configuration of the adaptive distributed video player for the experiments



(a) 128 × 96 video clip

(b) 256 × 192 video clip

Figure 6.13: The size of individual video frames in the two video clips played in the experiments

Fig. 6.13 (a) and (b) respectively. It can be seen from these two figures that in each clip, the frames roughly fall into two groups based on their size. The group with smaller frame size is composed mainly of B frames, and the group with bigger size is mostly I and P frames. Some statistics of the frame sizes of the clips are listed in Table 6.3. These two clips can be played individually, or can be combined to form a single multi-resolution clip. These video clips stored on the local server *lemond* are each about 5 minutes long (9259 frames). But the remote server *virgo* only holds the first 2 minutes of each clip, due to a limited disk quota.

Resolution	I frames			P frames			B frames			Ave.
	max	ave	min	max	ave	min	max	ave	min	fr. size
128 × 96	2585	1065	538	2850	846	90	1265	429	25	586
256 × 192	10425	4531	2589	8598	3612	2021	3934	1644	39	2376

Table 6.3: Frame size (in bytes) statistics of the video clips used in the player experiments

6.5.2 Video Playback Performance Metrics

There are several performance metrics associated with a video playback session: client-side buffer-fill level, video resolutions, frame rates sent by the server or displayed by the client, frame-drop ratio by the network and client stages of the video pipeline, and video smoothness. The client-side buffer-fill level measurement reflects the performance of the client-server synchronization feedback. Comparing the average buffer fill level of playback sessions with networks of different latency variation levels reveals how effectively the synchronization feedback associates the target buffer-fill level with the latency variation. Through figures of buffer-fill-level over time, the feedback's reaction to buffer-fill level variations, clock drift and other events can be observed clearly. By comparing the display frame rate and frame-drop ratio between playback sessions with or without the feedback mechanisms enabled, the effect of the feedback mechanisms on improving video quality and resource usage can be evaluated. Finally, display-frame-rate-over-time figures of playback sessions give insight into the dynamics of the pipeline conditions and feedback mechanisms.

Display frame rate is a important measurement of the user-observed video quality of a playback session. However, the display frame rate alone is not sufficient for accurate description of the video quality. Consider two playback sessions of the same video stream at the same play speed and having the same display frame rate. If one drops frames more evenly than the other, the former will be smoother than the latter, and may be considered "better" with respect to video quality. Hence, we need a metric to quantify this smoothness aspect of the video playback quality.

One smoothness measurement is based on *presentation jitter* as proposed by Staehli [58]. In our player, we assume that the mapping from the logical time of a video stream to the

system time of the client is precise [10]. All the frames that are displayed are displayed on time, and all late frames are dropped. Based on this assumption, we define the presentation jitter in terms of logical display time (video-frame sequence number). Consider a video stream consisting of a frame sequence $F = (f_1, f_2, \ldots f_n)$, and a playback session displaying a subsequence $D = (f_{d_1}, f_{d_2}, \ldots, f_{d_m})$, where for all $f_{d_j} \in D$, $f_{d_j} \in F$ (i.e., $1 \leq d_j \leq n$), and for all $1 \leq j < m$, $d_j < d_{j+1}$. At each logical display time i ($1 \leq i \leq n$), frame $f_i \in F$ is expected to be displayed. But since f_i may be dropped by the video pipeline, the player may actually still be displaying an earlier frame[3] instead of f_i. Suppose the actually displayed frame is f_{d_k} (the kth frame in the display subsequence D), then we have $d_k \leq i$ (the frame displayed is either the expected one or an earlier frame) and $d_{k+1} > i$ (the next frame in the display sequence D is a future frame). We can calculate the logical time error $e_i = i - d_k$ (the staleness of the frame being displayed relative to the expected one), and produce an error sequence $E : (e_1, e_2, \ldots e_n)$. The smoothness S of a playback session is then defined as the standard deviation of the error sequence E from that of the perfect playback session that drops no frame and thus has an E of all zeros.

$$S = \sqrt{\frac{\sum_{e \in E} (e - 0)^2}{n}}$$

For example, suppose we have a video stream with a frame sequence $F = (f_1, f_2, f_3, f_4, f_5)$. One playback session displays a subsequence $D = (f_1, f_3, f_5)$. Then the error sequence of this playback session is $E = (0, 1, 0, 1, 0)$, and its smoothness measurement is

$$S = \sqrt{\frac{(0-0)^2 + (1-0)^2 + (0-0)^2 + (1-0)^2 + (0-0)^2}{5}} = \sqrt{\frac{2}{5}}$$

This definition of S has some favorable properties. It only relates to frame-drop rate, and is independent of the play speed and frame rate themselves. A playback with a smaller value of S can generally be considered being smoother. A perfect playback in which no frame is dropped has $S = 0$. Though this smoothness measurement is somewhat ad-hoc[4], and no proof of its formal property is available, we can still see that: (1) adding

[3] In our video player, a frame remains being displayed in the video window until updated by a new frame.

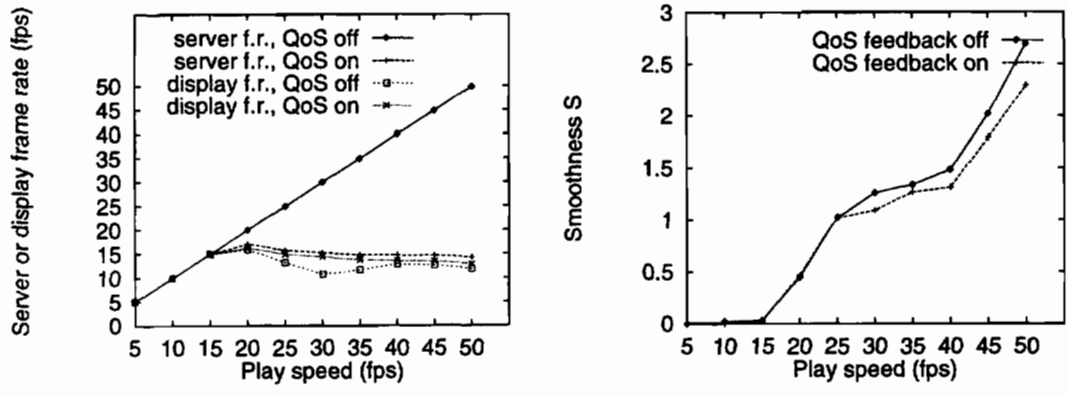
[4] We introduce this ad-hoc S because no other standard and widely accepted smoothness measurement has been identified.

a frame to an existing display frame sequence results in a display sequence with lower S value; and (2) moving one frame in an existing display frame sequence towards the center between its two neighbors results in a display sequence of lower S value. Suppose frame sequence $F = (f_1, f_2, \ldots, f_9)$, and there are two playback sessions with display frame sequences D_1 and D_2, and smoothness S_1 and S_2 respectively. In the first case, if $D_2 = (f_1, f_3, f_5, f_7, f_9)$ is the result of adding an additional frame f_5 to $D_1 = (f_1, f_3, f_7, f_9)$, then we have $(S_2 = \frac{2}{3}) < (S_1 = \frac{4}{3})$. In the second case, $D_1 = (f_1, f_3, f_4, f_7, f_9)$ and $D_2 = (f_1, f_3, f_5, f_7, f_9)$ have the same frames except for a single one: $f_4 \in D_1$ and $f_5 \in D_2$, the frame f_5 in D_2 is more evenly spaced than the frame f_4 in D_1. In this case, we also have $(S_2 = \frac{2}{3}) < (S_1 = \frac{\sqrt{7}}{3})$.

6.5.3 Experiments Across the Local Area Network

Experiments have been conducted with the server *lemond* and the client *anquetil* connected through the Ethernet LAN. In these experiments, the server and the network have enough resources to handle all the frames requested. The client CPU, due to its insufficient processing speed to decode MPEG frames in software, becomes the weakest stage of the video pipeline.

To see how the QoS feedback improves video playback quality and saves resources, an experiment is carried out, in which the 256 × 192 clip is played multiple times at play speeds ranging from $5fps$ to $50fps$. In the playback sessions in this experiment, and in all other playback sessions discussed in this chapter, no limit on frame rate is specified. The play speed is increased from $5fps$ to $50fps$ at steps of $5fps$. We can see when the client is saturated, and how the QoS feedback reacts upon overload. Figure 6.14(a) shows the average server-sent and client-display frame rates for all the sessions. Figure 6.14(b) shows the smoothness measurements of these sessions. From Fig. 6.14(a), we can see that for play speeds up to $15fps$, the client is able to decode and render all the frames, so the QoS feedback makes no difference. But from speed $20fps$, the client becomes overloaded and begins to drop frames. Without the QoS feedback, the server and the network will continue to send frames at the full frame rate, and excessive ones are received and dropped by the client. At the play speed of $50fps$, more than 73% of total frames are dropped, thus



(a) Frame rate sent by the server, and displayed by the client

(b) Smoothness measurement S

Figure 6.14: Video quality comparison between playbacks with QoS feedback enabled and disabled, in the LAN environment

wasting a lot of server, network and client resources. In contrast, when the QoS feedback is enabled, it monitors the pipeline congestion condition, and controls the rate at which the server streams out frames. The QoS feedback is able to keep the frame drop ratio within 10%, and generally yields higher frame rates due to less resources being wasted at the client. Figure 6.14(b) shows that the QoS feedback also in general results in smoother playbacks.

The synchronization feedback is able to adapt the client buffer-fill level to the video pipeline latency and variation, as well as compensating clock drifts and reacting to other events. Since the latency and jitter of the pipeline from *lemond* to *anquetil* through the LAN is low, on all the playback sessions, the target buffer-fill levels are kept around 4 frames (130 milliseconds), which is the minimum number of frames to keep all the substages in the client busy. In the experiment configuration, the clock rates of *lemond* and *anquetil* are almost identical. To see how clock drift is compensated, the server logical clock is adjusted so that it runs initially 0.4% slower than that of the client, and the 128×96 clip is played twice, once with the synchronization feedback on and once with it off. Figure 6.15 shows the buffer-fill level of the two playback sessions over time. For the session without the synchronization feedback, the buffer-fill level keeps decreasing, and at

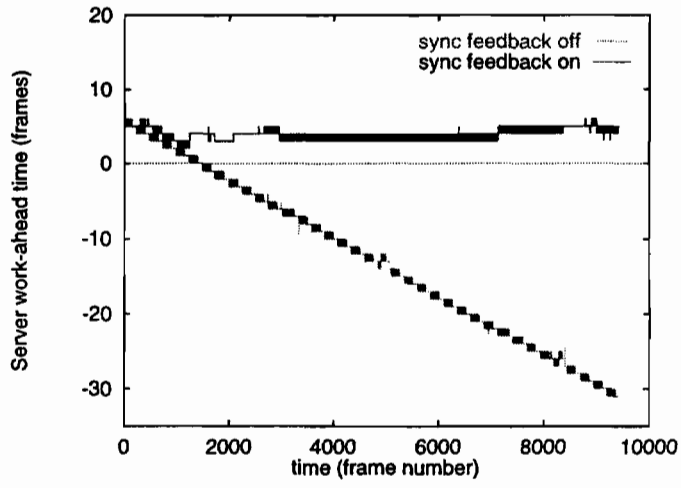


Figure 6.15: Server clock drift compensation by the synchronization feedback, LAN experiment

some time around frame number 1500, it falls below zero. Beyond this point the player stalls since all the frames are late when they arrive at the client[5]. In the other session, in which the synchronization feedback is enabled, the feedback reacts to the clock drift by adjusting the server clock phase and rate accordingly to keep the buffer-fill level around 4 frames.

6.5.4 Experiments Over PPP

When *lemond* and *anquetil* are connected through a PPP instead of an Ethernet link, due to PPP's low link bandwidth ($28.8Kbps$), the bottleneck of the video pipeline shifts from the client to the network connection. Also due to PPP's low link bandwidth, the time for streaming video frames of different size differs greatly. For example, a minimum-sized frame ($25B$) in the 128×96 clip takes about 10 milliseconds, while a maximum-sized frame ($2850B$) takes about 1 second. The maximum-sized frame in the 256×192 clip has $10425B$

[5]To avoid a totally stalled video, in the prototype player, there is a "no-drop" option. When this option is selected, the player plays a late frame if there is no other more recent frames available in the client. So when the server is running behind the client, instead of seeing no video at all, the user will still see a "sluggish" one. In this case, however, the player fails to maintain a real-time playout guarantee on all the frames. This option is used for all the experiments in this section. In a playback session, if the server runs behind the client for some time, the smoothness S measurement value is better (smaller) than it should be. This smaller S is because some frames are displayed later than the times as indicated by their sequence number (logical time).

and takes about 4 seconds. Long client-side buffering time is needed to accommodate this large variation in latency. Another property of PPP (at least for the PPP service at CSE OGI) is that the PPP servers have relatively large (more than 10 seconds) buffer space for individual links. The resultant large buffering delay imposes a great challenge to the feedback mechanisms employed in the video player.

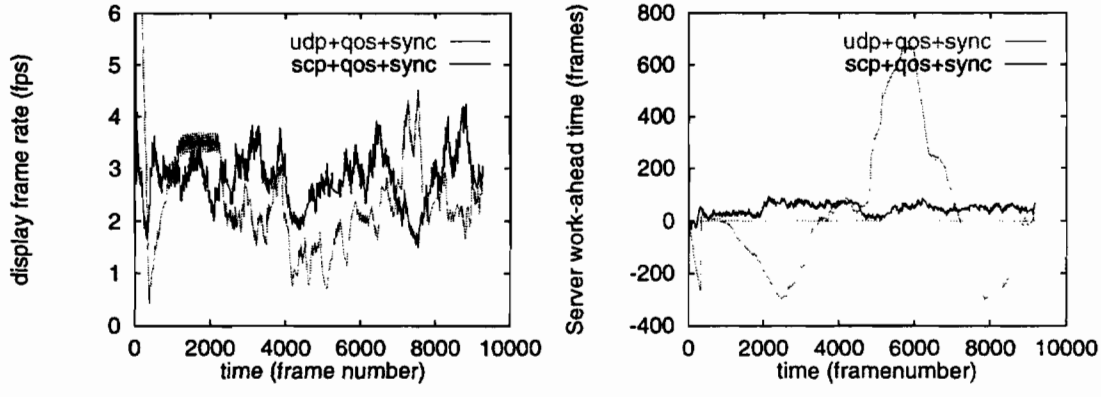
The SCP protocol plays an important role in preventing excessive buffer build-up, and eventual frame drop. Without SCP, the pressure from the server as maintained by the QoS feedback would build up the PPP buffer until it overflows and frames are dropped by the PPP server. Then the QoS feedback reacts by reducing the frame rate, which eventually empties the buffer. This repeated process of buffer build-up, frame drop, and buffer emptying leads to an unstable video frame rate. To make the problem even worse, to the synchronization feedback, the significant latency changes caused by large buffering variations look just like client-server clock asynchrony as caused by clock drifts or other factors. Thus it can be falsely triggered, and can accelerate the server when the buffer builds up, and can slow the server down when the buffer empties. This false synchronization action coupled with the long buffering latency can cause the server work-ahead time to oscillate wildly.

To evaluate the performance of the feedback mechanisms over the PPP link, the 128×96 resolution video clip is played at normal speed ($30fps$) five times in each of the configurations with SCP, the QoS feedback or the synchronization feedback enabled or disabled. The configurations experimented are (1) *udp−qos−sync* – none of the three feedback mechanisms are used; (2) *udp+qos+sync* – the QoS and synchronization feedback mechanisms are enabled, but raw UDP instead of SCP is used; (3) *scp-qos+sync* – SCP and the synchronization feedback mechanisms are enabled, but the QoS feedback is disabled; and (4) *scp+qos+sync* – all three feedback mechanisms are enabled. Table 6.4 shows the configurations, and the mean and variance of the average server-sent and client-display frame rates, the smoothness and the average buffer-fill level (server work-ahead time) of the five playback sessions in each of the configurations[6]. Figure. 6.16(a) shows

[6]In the table, for server-sent and client-display frame-rate, and smoothness, the mean and variance of the (average) measurements from the five sessions are shown. But for server work-ahead time, it is possible

Configuration	udp−qos−sync		udp+qos+sync		scp−qos+sync		scp+qos+sync	
	mean	variance	mean	variance	mean	variance	mean	variance
Server sent fps	30.00	0.00	4.35	0.024	2.82	0.019	2.75	0.028
Display fps	0.408	0.033	2.36	0.090	2.81	0.017	2.75	0.031
Smoothness S	116	18.1	18.8	2.73	9.06	0.022	9.04	0.017
Work-ahead	−337	47.7	−96.7	170	27.6	17.2	35.5	19.0

Table 6.4: Configuration and performance results of playback sessions across a 28.8$Kbps$ PPP link



(a) Display frame rate

(b) Server work-ahead time

Figure 6.16: Video quality comparison between playbacks when UDP or SCP is used, across the PPP link

the display frame-rate of one session in each of the two configurations *udp+qos+sync* and *scp+qos+sync*. The frame-rate samples are gained by applying the display time of of all displayed frames in a playback session to the frame-rate estimator shown in Fig. 6.6, with a parameter of 0.02. Figure 6.16(b) shows the the buffer-fill levels (server work-ahead time) of these same two sessions over time.

In the first configuration *udp−qos−sync*, none of the feedback mechanisms are enabled, so the server streams out all the frames at an average data rate of $586 \times 30 \times 8 \approx 141 Kbps$. With a link speed of only 28.8$Kbps$, the PPP server randomly drops over 80% of the input packets, as well as introducing more than 10 seconds of latency. Even worse, since frames

for the client to measure the work-ahead time (in terms of number of frames) when each individual frame is displayed (this is not true for other measurements by their definitions). So in the table we show the mean and variance of all the raw measurements from all the five sessions. This measurements reflects more precisely the fluctuation in server work-ahead time within a session.

larger than $1500B$ (MTU of the PPP link) are fragmented by the video server into several packets, and the decoding of B and P frames relies on the availability of their reference I or P frames, most of the packets and P or B frames that made it to the client were dropped by the client frame reassembler or decoder. As a result, the video shown on the screen is an extremely jerky and unstable one (smoothness S having mean 116 and variance 18.1) with an average frame rate of only about $0.4fps$, as shown in Table 6.4.

With the use of the QoS and synchronization feedback mechanisms, in configuration *udp+qos+sync*, the video playback quality improves significantly. The video displayed is much smoother and stabilizes at a much higher frame rate of about $2.4fps$ (Table 6.4). The main reason for this improvement is that the server streams at only about $4.4fps$, so the PPP server drops relatively few packets, and most packets arrive at the client and can be reassembled into frames, decoded and displayed. However, not all problems are solved by the QoS and synchronization feedback mechanisms. As shown in Table 6.4, the display frame rate of the sessions are lower than that when SCP is used ($2.36fps$ versus about $2.8fps$), and the smoothness S is much higher (18.8 versus about 9), indicating unstable playback. As shown in Fig. 6.16(b), the interaction between the QoS and synchronization feedback causes the buffer-fill level to fluctuate wildly. This fluctuation is also reflected in Table 6.4 by the below-zero average work-ahead time -96.7 and the huge variance of 170. It can be seen from Fig. 6.16(a) and (b) that the QoS feedback yields periods in a playback session with noticeably different frame rates. The display frame rate is generally higher when the buffer is being filled up (server work-ahead time is decreasing), and lower when the buffer is being emptied.

SCP, working with the synchronization and QoS feedback mechanisms, greatly helps optimize video playback quality and resource usage. As shown in the last two configurations of Table 6.4, virtually all the frames streamed out by the server are passed along the video pipeline to the video screen. All the measurements, including the server-sent and display average frame rates, the smoothness, and the average server work-ahead time are very stable (indicated by the small variance). The display frame rate and buffer-fill level for the first playback session in the *scp+qos+sync*, as shown in Fig. 6.16(a) and (b), indicate that both the frame rate and buffer-fill level in these configurations are stable. Careful

readers may notice that in Fig. 6.16(a), in the session with configuration *scp+qos+sync*, there are two places around frame number 4500 and 7500 where the display frame rate drops significantly. Comparing this display-frame-rate figure against the frame size plot in Fig. 6.13(a), it is clear that these drops in display frame rate are a result of larger I and P frames around those two positions.

Comparing the results of the last two configurations in Table 6.4, it can be seen that the two configurations *scp−qos+sync* and *scp+qos+sync* yield very similar resource usage and video quality. The results to some extent indicate that the QoS feedback results in slightly lower server-send and client-display frame rates, but a slightly smoother video (lower smoothness S). A student-t test (explained in Appendix A) on these results shows that the performance difference between the two configurations are statistically insignificant. The student-t values for the server send frame rate, client display frame rate and smoothness S are 0.722, 0.612 and 0.226 respectively. But for two sets, each with 5 samples, to have a different mean at 90% confidence, the student-t value should be no smaller than $t_{0.90,8} = 1.40$. A plausible reason that the results are similar is that even without the QoS feedback, the video server still sees, with the help of SCP, the bottleneck network stage, and drops frames selectively (e.g., making sure that a P or B frame is not sent if not all its reference frames are already sent). Since the PPP link is not shared, and has a hard link bandwidth limit, the video quality is mainly determined by the bandwidth of the PPP link. To explain the potentially lower frame rate and lower S (smoother video) caused by the QoS feedback, it can be seen that when the QoS feedback is enabled, the server will get a next-step frame rate from the client, and plans frame dropping accordingly in advance. This scheduled frame dropping may result in a smoother video than simply dropping frames that are already late. On the other hand, due to the wide variation in MPEG frame sizes as shown in Fig. 6.13, scheduled frame dropping may result in less utilization of the PPP link when sizes of the frames are small, and thus lower video frame rate.

Table 6.3 also shows that the playback sessions with QoS feedback enabled (*scp+qos+sync*) have a higher average server work-ahead time than that when the QoS feedback is disabled (*scp−qos+sync*) (27.6 versus 35.5). We believe that this difference is

actually not significant. The synchronization feedback computes target work-ahead time in an exponential manner and brings the actual work-ahead time close to the target. Unless the difference in the average work-ahead time is large (at least a factor of two), it should not indicate any significant difference in actual pipeline latency jitter.

6.5.5 Experiments Across the Long-Haul Internet

Experiments over the long-haul Internet are presented in this section. In these experiments, the player has the remote Georgia Tech workstation *virgo* as the server and *anquetil* as the client, and is configured with various combinations of feedback mechanisms. The 256×192 resolution video is streamed from *virgo* to *anquetil* across more than 20 Internet hops. The experiments are performed during rush hours on a busy weekday, when the Internet is highly congested, so that the network is the bottleneck in the pipeline. Otherwise, the client would still be the bottleneck and we would have results similar to those in the LAN case. The experimental results show that SCP is a better protocol than TCP for real-time video streaming[7] and that SCP and the QoS feedback work together to improve the video quality. The results also show that the synchronization feedback is able to maintain a higher average buffer-fill level to cope with the larger latency variation than in the LAN case.

Unlike for the LAN, the long-haul Internet environment is characterized by properties such as a high degree of dynamics, unpredictability and uncontrollability. An Internet connection is composed of many network links and routers belonging to completely different entities, and the player shares these links and routers with many other network sessions. Thus the bandwidth availability is highly dynamic and unpredictable. Furthermore, without coordinating all the parties using the Internet, there is no way to control the sharing. As a result, it is impossible to repeat an experiment in the same environment. If the network environment changes too rapidly, the results of individual experiments (playback

[7]Even though UDP with the QoS feedback may yield a better video quality, it does this by sacrificing all other TCP connections sharing the same links. Since the QoS feedback has time-based estimation and does not do exponential back-off, it is more aggressive than TCP, and will drive all the sharing TCP connections to back-off when network bandwidth is scare. Thus UDP is not an appropriate protocol for video streaming across the shared Internet. It is unfair to compare the playback performance between UDP and SCP or TCP.

sessions) may not be able to clearly demonstrate the effectiveness of the feedback mechanisms. In this case, statistical analysis can be used. A video clip is played repeatedly a sufficient number of times, with the feedback to be evaluated being enabled or disabled alternately. Then the mean and variance of the performance measurements of the playback sessions with each of the two configurations are calculated, and compared against each other to see whether their difference is statistically significant or not (Appendix A).

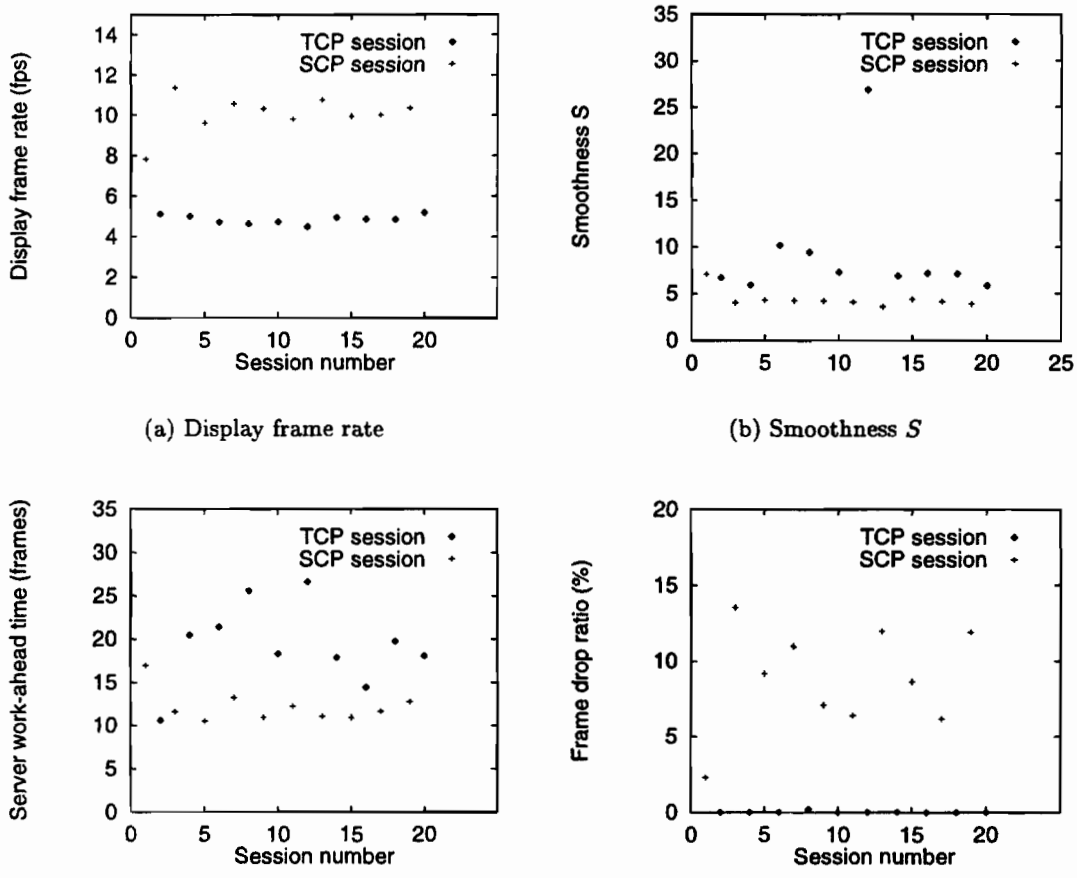
In the following sections, we discuss the experiments performed to evaluate the feedback mechanisms and the results.

SCP versus TCP Network Connections

SCP is designed for real-time media streaming across the Internet, while TCP is for reliable data transfer and is not appropriate for streaming. TCP's inappropriateness for streaming is partially due to its infinite retransmission, which results in wasted bandwidth (for retransmitting late data), lower throughput, and higher and more unpredictable latency. To see the advantage of SCP over TCP, the 256 × 192 video clip is played repeatedly for 20 times, with both the QoS and synchronization feedback mechanisms always on, and TCP and SCP connections used alternatively. The performance results of the experiment, as shown in Fig. 6.17 (a), (b), (c) and (d), demonstrate that SCP is a clear winner over TCP, except in frame drop ratio due to TCP's reliability. According to Fig. 6.17(a), (b) and (c), in most of the sessions, SCP yields significantly higher display frame rate, better smoothness, and a lower buffer-fill level. Figure 6.17(d) indicates that SCP improves the playback performance while successfully keeping the frame-drop ratio reasonably low (within 15%). From these figures, some abnormalities can also be observed, such as significantly lower display frame rate and higher buffer-fill level in SCP session 1 (Fig. 6.17 (a) and (c)). We believe these are just indications of large variations in the Internet conditions.

The Effect of QoS Feedback

An experiment has also been conducted to show that the QoS feedback improves the video playback quality. In this experiment, the 256 × 192 resolution video clip is played



(a) Display frame rate

(b) Smoothness S

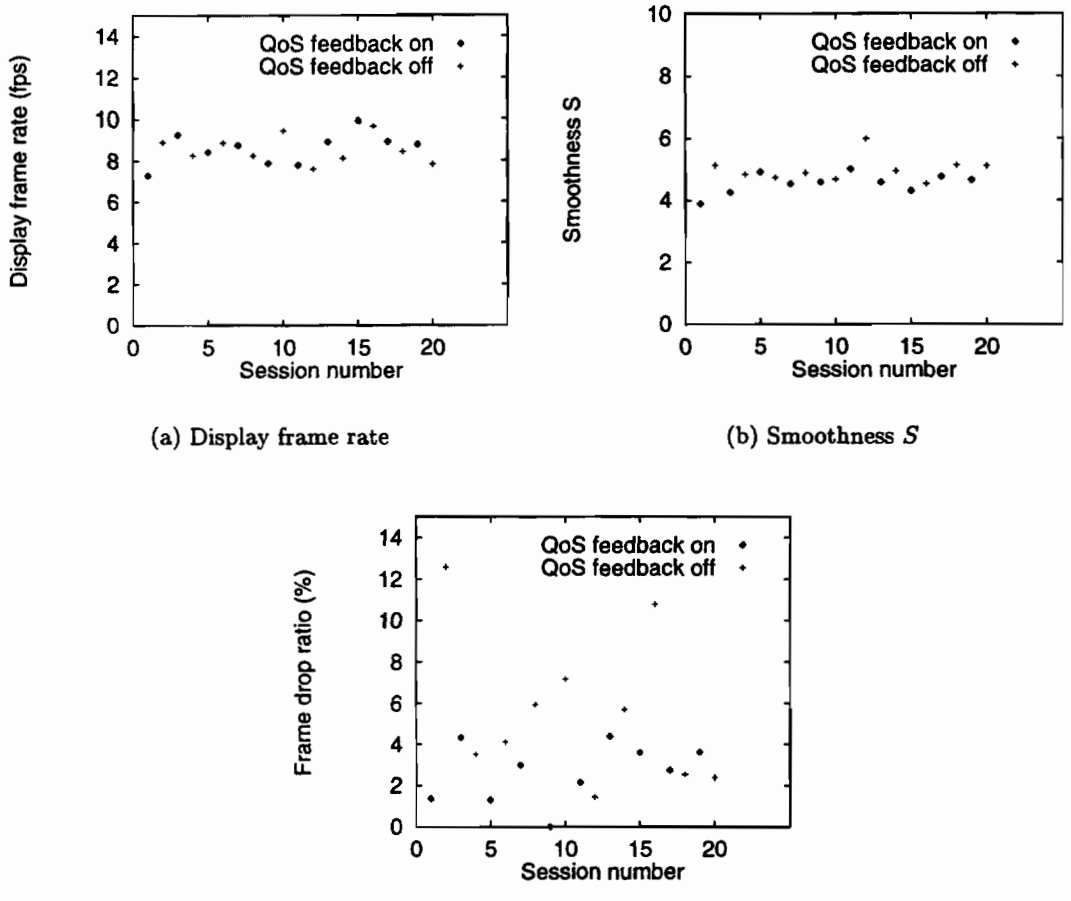
(c) Server work-ahead time

(d) Frame drop ratio

Figure 6.17: Playback performance comparison between when SCP or TCP is used, across the long-haul Internet

continuously 20 times, with the QoS feedback enabled and disabled alternately. In all the playback sessions, SCP and the synchronization feedback are used. The performance of the individual sessions including display frame rate, smoothness and frame-drop rate are shown in Fig. 6.18(a), (b) and (c). Table 6.5 shows the sample mean and variance of these measurement of the two sets of playback sessions. This table also contains the results of the student-t tests, assuming a confidence coefficient of 0.9 (Appendix A), on the difference between the means when the QoS feedback is enabled or disabled. We are sure that the network is the bottleneck, since the display frame-rate measurements shown in Fig. 6.18(a) are significantly lower than those in the LAN case shown in Fig. 6.14(a). But, in both the LAN and long-haul Internet experiments, the same client is used. Furthermore, experiments with raw UDP network connections and the QoS feedback disabled show the remote server is able to send frames at full rate.

From the figures and the table, we can see that the QoS feedback improves significantly on smoothness and the packet drop ratio, but the display frame rate is hardly affected. Similarity in display frame rates indicates that the QoS feedback is able to sufficiently exploit the network bandwidth, since the other configuration, in which the server streams frames as quickly as possible, should have sufficiently exploited the network bandwidth. The smoother video and lower frame-drop ratio means that the QoS feedback can improve the video quality and resource usage with the limited network bandwidth. One reason for this result is that the server, based on the next-step frame rate provided by the QoS feedback, is able to schedule frame dropping in advance. This scheduled frame dropping should result in a smoother streaming as well as less packet drop in the network connection. This effect of the QoS feedback on the smoothness of video playback sessions across the long-haul Internet is similar to, and more significant than, that of the QoS feedback on video playbacks across PPP, as discussed in Section 6.5.4. The QoS feedback is able to improve video smoothness and reduces the packet-drop ratio while exploiting the network available bandwidth sufficiently.



(a) Display frame rate

(b) Smoothness S

(c) Frame drop ratio

Figure 6.18: Playback performance comparison between when QoS feedback is on or off, across the long-haul Internet

\ Configuration Measurements \		QoS feedback enabled	QoS feedback disabled	Student-t value	difference
Display frame rate (fps)	mean	8.583	8.529	0.017	insignificant
	variance	0.773	0.681		
Smoothness S	mean	4.546	4.995	1.655	significant
	variance	0.330	0.406		
Frame drop ratio (%)	mean	2.649	5.616	3.661	significant
	variance	1.442	3.684		

Table 6.5: Statistics on the performance measurements for the playback sessions with the QoS feedback enabled or disabled, across the long-haul Internet. Student-t percentile $t_{0.9,18} = 1.33$

Resolution Adaptation

When a multi-resolution video clip is played, the resolution adaptation policy in the QoS feedback also adapts the resolution to the amount of resources available in the video pipeline, which in the Internet experiments is usually the network available bandwidth. To demonstrate the effect of resolution adaptation, the two video clips are combined into a multi-resolution clip, and streamed from *virgo* across the Internet to *anquetil*. Since it is not easy to catch a period in which the available bandwidth of the network connection changes drastically from time to time, in our experiment, UDP packets are sent from *anquetil* to the *virgo* and back at a certain rate to create an certain amount of background traffic. Figure 6.19 shows the display frame rate of a playback session. In this experiment, the resolution adaptation policy is set to scale up when frame rate is close to $30fps$, and to scale down at $7fps$. From the figure, it can be seen that the playback session starts with the higher (256×192) resolution. At about frame number 700, the background traffic is injected, so the frame rate goes down. After a while the resolution policy detects the quality degradation and switches to the lower (128×96) resolution. At about frame number 1400, the background traffic is removed. From then on, the display frame rate tends to increase slowly, though the random Internet packet drop still causes big variations. At about frame number of 1800, the display frame rate goes up to close to $30fps$ and the player switches back to the higher resolution. Later at about frame number 3000, the background traffic is injected again and triggers another round of resolution switching.

The Effect of Synchronization Feedback

With the network latency from server *virgo* and client *anquetil* being more than 100 milliseconds, and a variation much higher than in the LAN environment, the synchronization feedback maintains the average client buffer-fill level around 12 frames (0.4 second), as indicated by the SCP sessions in Fig. 6.17(c). However, due to the high degree of dynamics of the Internet, occasionally, the latency variation can become much higher than usual, and the synchronization feedback reacts by increasing the target and actual buffer-fill levels to close to one second. When the buffer-fill level measurements of the TCP sessions

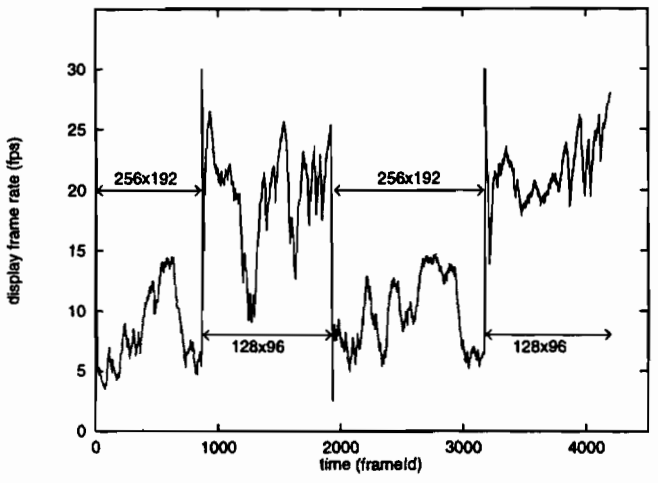


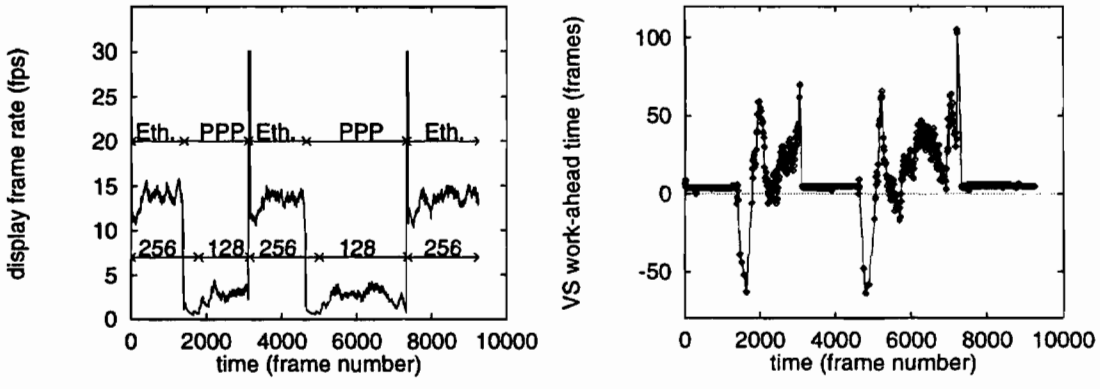
Figure 6.19: Display frame rate of a video playback session with resolution adaptation, across the long-haul Internet

in Fig. 6.17(c) are examined, we discover that the buffer-fill levels for these sessions are around 20 frames (0.7 second), which higher than that of the SCP sessions. This result is due to the higher latency and greater variation caused by TCP's data retransmission.

6.5.6 Experiments with Network Interface Switching

Experiments have also been carried out to test how SCP, the QoS and synchronization feedback mechanisms react with meta-adaptation operations upon a client-side network interface switch between PPP and Ethernet. In these experiments, the local host *lemond* serves as the video server, and *anquetil* as the client. The client *anquetil* is configured to have both the PPP and Ethernet interfaces active. During a playback session, an external utility is used to switch *anquetil*'s default route between the PPP and Ethernet, and send a signal to the active player client after each switch. Upon receiving a network interface switch signal, the player client re-establishes both the control and data connections to the video server, resets SCP, and performs meta-adaptation to have the QoS and synchronization feedback mechanisms quickly adapt to the explicit event[8].

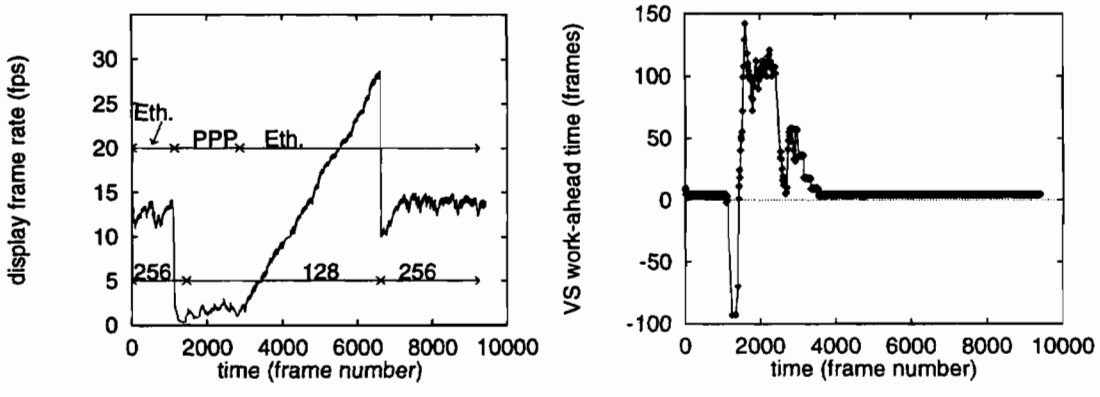
[8]The reason for using an external utility to simulate an interface switch and to notify the relevant application is due to the lack of IP-level plug-and-play in LINUX, the operating system on client *anquetil*. In a fully plug-and-play platform, we would expect that the network cards can be removed or inserted while the IP stack is active. Then the IP stack is reconfigured, and the active applications are notified automatically through some kind of signalling mechanism.



(a) Video resolution and display frame rate

(b) Server work-ahead time

Figure 6.20: Video resolution and display frame rate, and server work-ahead time, of a playback session with an interface switch. The event triggers meta-adaptation in the feedback mechanisms.



(a) Video resolution and display frame rate

(b) Server work-ahead time

Figure 6.21: Video resolution and display frame rate, and server work-ahead time, of a playback session with an interface switch. Meta-adaptation on the feedback mechanisms is disabled.

The video resolution and display frame rate and server work-ahead time of an example playback session are shown in Fig. 6.20(a) and (b). In this session, the player starts with the Ethernet interface. The higher resolution (256×192) video is played at about $15fps$, and the server work-ahead time is kept at within 5 frames. At about frame number 1400, the network is switched from Ethernet to PPP. With this switch, the network bandwidth suddenly drops from $10Mbps$ to $28.8Kbps$. Since the 256×192 resolution is still being played, which according to Table 6.3 has an average I frame size of $4531B$, the frame rate drops down to below $1fps$ (Fig. 6.20(a)) and the transmission latency increases to more than two seconds. As shown in Fig. 6.20(b), the long network latency results a server work-ahead time of about -60 frames. After a few frames are played, the QoS and synchronization feedback mechanisms detect the performance degradation, and react by switching to the low resolution video, and advancing the server clock value. With these feedback actions, the video display frame rate goes up to about $3fps$, and the server work-ahead time quickly increases to around 30 frames (one second). After a while, at about frame number 3200, the network is switched back to Ethernet. The feedback mechanisms are reset again upon this event to re-detect the new environment. The sudden big increase in network bandwidth results in the low resolution video being played at full $30fps$ (Fig. 6.20(a)), and the transmission latency being reduced to a few milliseconds, as reflected by the big upward jump in server work-ahead time (Fig. 6.20(b)). With the greatly improved playback performance, the QoS and synchronization feedback mechanisms react much faster than when switching from Ethernet to PPP. The video resolution is scaled up to 256×192, and the server work-ahead time lowered down to about 5 frames. At about frame numbers 4600 and 7400, the network is again switched to PPP and back to Ethernet, respectively. According to Fig. 6.20(a) and (b), the QoS and synchronization feedback mechanisms also react quickly to adapt the playback performance to the new network conditions.

For comparison purposes, the prototype video player is implemented with an option to disable meta-adaptation operations upon a network interface switch. The player still re-establishes the control and data channels between its client and server. However, the

feedback mechanisms simply inherit their previous states. We played the same multi-resolution video clip once with the meta-adaptation turned off. The video resolution, frame rate and server work-ahead time over time are shown in Fig. 6.21(a) and (b), respectively. Comparing Fig. 6.21 against Fig. 6.20, while no obvious difference can be seen when switching from Ethernet to PPP, since the bandwidth of PPP is so low that it needs a long time before it can be correctly detected anyway, the meta-adaptation makes the speed of adaptation much faster when switching from PPP to Ethernet. Figure 6.21(a) shows that it takes the steady-state QoS feedback mechanism a long time (120 seconds) to increase the display frame rate from $2fps$ to $30fps$ and finally switch to the higher resolution. This sluggishness is because that the QoS feedback only increases the server-sent frame rate additively if the video pipeline is not overloaded. The additive increase in frame rate makes the QoS feedback a good network citizen, but at the same time means a slow adaptation in the increase in resource availability. Figure 6.21(b) shows that after switching from PPP to Ethernet, the server work-ahead time reduces exponentially, instead of jumping directly to the right value. This result is due to the exponential adjustment on server work-ahead time by the synchronization feedback, plus the (sluggish) lowpass filtering in the estimation of the average buffer-fill level and variation.

6.6 Discussion

In this chapter, a real-time adaptive distributed MPEG player is described, with a focus on the design, implementation and experiments of the QoS and client-server synchronization feedback mechanisms. Due to the dynamic behavior and unpredictability of the Internet, the feedback systems need to be highly adaptive, adapting to changes in environment parameters such as network bandwidth, transmission latency and variation, and host processing power, as well as reacting to explicit events including playback speed change and network interface switches. The software feedback toolkit greatly helps the development of these complex QoS and synchronization feedback mechanisms. The feedback toolkit methodology is followed to decompose the feedback mechanisms into components policies, to identify the events, and to place the guards. In the implementation of the feedback

mechanisms, many components are taken from the toolkit component library. The instrumentation tools then help in visualizing the feedback effects in real-time, and tuning the parameters.

Extensive experiments have been conducted to evaluate the SCP, QoS and synchronization feedback mechanisms of the player in several types of network configuration, including Ethernet LAN, PPP, long-haul Internet and network interface switching between PPP and Ethernet. The experimental results show that feedback improves the video playback performance and resource usage in all the environments. The feedback mechanisms also react quickly to events such as network interface switches, and adapt the playback quality accordingly.

Chapter 7

Related Work

The software feedback toolkit is part of the Synthetix project, which is based on the earlier Synthesis project. The Synthesis project applied techniques such as run-time code synthesis (specialization) and software feedback to improve the execution performance and adaptability of operating systems. The Synthetix project investigated frameworks to apply the Synthesis techniques systematically to build adaptive systems. The software feedback toolkit provides a framework for using the software feedback techniques. Other research work related the software feedback toolkit include toolkits based on control theories, and existing feedback systems for network flow and congestion control, clock synchronization between Internet hosts, and intra- and inter-stream synchronization and QoS adaptation in streaming multimedia applications. These existing feedback systems serve as potential applications of the software feedback toolkit, for systematic design, software reuse, and improvement in performance and extensibility.

7.1 Synthesis: Feedback-Based Adaptive Scheduling

The Synthesis operating system [31, 32, 46] is an efficient implementation of fundamental operating system services through techniques such as run-time code synthesis and feedback-based fine-grain scheduling. Through run-time code synthesis, frequently-used kernel routines are optimized by creating specialized versions of their executable code at run-time, when new opportunities for optimization become known. The feedback-based fine-grain scheduling performs frequent scheduling actions and policy adjustments, resulting in an adaptive and self-tuning system that provides effective support for real-time

multimedia applications.

The concept of software feedback was first identified in Synthesis, and is used for fine-grain adaptive scheduling of interdependent jobs such as threads in a pipeline [32]. Each pipeline consists of multiple threads as stages, coupled by their input and output queues. Data elements such as audio samples are passed along the pipeline stage-by-stage, and processed at each stage. Information such as the length of a (stage) thread's input and output queues, which is local to the thread, reflects the thread's progress relative to other stages in the same pipeline. The CPU quantum and scheduling frequency of a thread reflect its execution speed. When a thread's input or output queues are too full or too empty respectively, the thread is falling behind. Conversely, when a thread's input or output queues are too empty or too full, respectively, the thread is running too fast. The feedback-based fine-grain scheduling of each thread uses the thread's local information to adjust its execution speed. It continuously monitors the length of its input and output queues, and changes its CPU quantum or scheduling frequency accordingly. Due to the run-time code synthesis, Synthesis enjoys fast interrupt processing, fast context switching and low thread-dispatching overhead. Furthermore, feedback-based scheduling of a thread only uses its local information and thus is simple and fast. As a result, the scheduling of each thread can be invoked frequently, so as to track the thread's needs at a fine temporal granularity. Though the feedback-based fine-grain scheduling of each thread uses only the thread's local information, interaction between the schedulers through the inter-stage input-output queues brings all schedulers together. Through this interaction, the threads in a pipeline eventually reach a dynamic equilibrium, in which all threads get an appropriate share of the CPU, the length of all the queues are maintained at appropriate levels, and data flows in the pipeline smoothly. In the feedback-based fine-grain scheduling, filters such as lowpass filters, differential filters and integrator filters are used to avoid potential oscillation, to speedup convergence, and to improve tracking accuracy.

Based on the work in Synthesis, Pu and Fuhrer envisioned a toolbox approach to feedback-based scheduling in a position paper [45]. The proposed toolbox would contain a set of standard and relatively simple components with well-defined performance and

functionality characteristics. A filter design tool would be used for interactive filter specification, composition, and generation. Simulation packages would then used to evaluate the behavior and performance of the generated filters and feedback systems.

In this thesis we investigated and extended the idea of the toolkit approach to software feedback, and presented a composition methodology and an implementation of the feedback toolkit. We proposed a methodology for the design of complex wide-range feedback systems through hierarchical composition and guard-based meta-adaptation. We then implemented the toolkit using C++ base classes for implementing new building blocks and composing complex component, a library of feedback components, and a set of tools for simulation and on-line instrumentation. We also demonstrated how the feedback toolkit can be used in building adaptive multimedia systems. The software feedback toolkit described in the thesis provides a framework for applying software feedback techniques of Synthesis to building adaptive systems. These techniques were previously used only by "artists" or "craftsmen" who could build on-of-a-kind systems. But with the help of the toolkit, they can be used by ordinary "engineers".

7.2 Synthetix: Optimistic Incremental Specialization for Adaptive Systems

The Synthetix project [38] investigates the application of *optimistic incremental specialization*, a technique for generating specialized code optimistically for system states that are likely to occur but not certain, to build high-performance adaptive operating systems. Synthetix is a follow-on from Synthesis [31]. It extends the results of the latter with a conceptual model of specialization, the idea of incremental and optimistic specialization, application of the idea in commercial operating systems [44], and a set of tools for automating the specialization process [38].

With optimistic incremental specialization, operating systems are optimized incrementally whenever opportunities are identified, both at compile-time and at run-time [44]. This contrasts to the traditional approach that operating systems are optimized statically for a single "common case" (which may not actually be a real common case in specific

situations) at coding or compile time. Traditionally, program specialization, also called partial evaluation, is a program transformation process aimed at customizing a program based on parts of its input [13]. In essence, program specialization consists of constant propagation and folding, and can be applied to programs that exhibit interpretation. However, it can be generalized to include arbitrary computations that help improve the performance of a software system in specific conditions. Operating systems exhibit a wide variety of *invariants* — assertions which stay valid for a long time or even for the lifetime of the systems. Example invariants are word size, cache size, whether a processor has a FPU, whether a file being manipulated is on local disk or NFS-mounted, etc. These invariants may become known either at compile-time (such as word size), or at various stages during run-time (such as location of a file, known when it is opened). Given a list of invariants (available statically at compile-time or dynamically at run-time), it should be possible to apply compile- or run-time specialization (optimization) to generate specialized code. Repeated dynamic application of specialization is called *incremental specialization.* While many invariants stay valid throughout the life cycle of an operating system, there are also many assertions which are *likely*, but not certain, to be true for a long period of time (during run-time). For example, it is likely, but not certain, that files will not be shared concurrently. This type of assertion is called a *quasi-invariant.* Specialization based on these most-of-the-time-valid quasi-invariants should also improve system performance. Correctness can be preserved by guarding every place where quasi-invariants may become invalid. A *guard* is a predicate testing whether the guarded quasi-invariants holds or not. For example, a guard can be placed in the open system call to test whether the file being opened is being accessed by other processes concurrently. When a guard is triggered indicating that a quasi-invariant becomes invalid, the guarded specialized code should be replaced by another version that does not exhibit the specialization based on the invalidated quasi-invariant. This process of dynamically replacing one version of code with another version is called *dynamic replugging.* The overall repeated process of specialization based on quasi-invariants is referred to as *optimistic incremental specialization.*

Optimistic incremental specialization has been successfully used to specialize a variety of OS components [44]. A set of tools (collectively referred to as the *specialization toolkit*)

has also been built for automatic identification and placement of the guards for a given set of quasi-invariants, generation of specialized code, and dynamic replugging [38].

The software feedback toolkit is part of the overall Synthetix project. Firstly, the overall goal of Synthetix is to investigate frameworks for building adaptive systems with techniques previously required great talent and creativity to apply. The two techniques we focused on — program specialization for improvement of execution performance and software feedback for system adaptability — both originated from the Synthesis project [31]. The software feedback toolkit provides a framework for application of software feedback, while the specialization toolkit provides a framwork for program specialization. Secondly, the software feedback toolkit is an application of the idea of optimistic incremental specialization in the context of software feedback systems. Software feedback can be seen as one form of specialization that specializes the policies used in a software system based on the specific conditions observed, whereas the specialization toolkit emphases removal of interpretations from the code. The concepts of *invariants*, *guards* and *dynamic replugging* play an important role in the methodologies for software feedback composition. Also, the feedback toolkit provides a framework for applying software feedback techniques to building adaptive systems. Thirdly, the feedback systems built on top of the building blocks from the toolkit usually contain a lot of cross-module interpretation, thus are potentially excellent targets for application of the specialization toolkit. Finally, it is also possible to use feedback-based techniques to identify quasi-invariants for dynamic specialization. For example, a lowpass filter can be used to monitor the recent pattern of access to a file, to see if the access is mostly sequential. If a value 1 is used for a sequential access and 0 is used for non-sequential access, then the closer the lowpass filter output is to 1, the more likely the access is sequential. If the filter output passes a given threshold, it is likely that specializing the file access code for the case of sequential access would increase execution speed.

7.3 Toolkits Based On Control Theories

Several commercially available toolkits, such as Matlab [59] and MATRIX$_X$ [21], support building control systems based on control theories such as linear systems theory [3, 5], nonlinear systems theory [14], fuzzy [34, 67] and neural [16, 67] control. They provide various pre-defined building blocks from control theories, GUI-based tools for control system composition, simulation and analysis, and generation of the code of the constructed control systems. The target applications of these control-theory-based toolkits are traditional hardware or embedded control systems. Taking Matlab [59] as an example, it has toolboxes for linear and nonlinear, continuous and discrete control, robust control, fuzzy logic, and neural control, etc. Each toolbox consists of building blocks as well as utilities for simulation and analysis. Matlab has a GUI-based environment called Simulink for graphical design, simulation and prototyping of continuous and discrete systems. It also has facilities to generate C, C++ or Ada code of the control systems from proprietary script- or GUI-based representations.

While control-theory-based toolkits are convenient and powerful in building adaptive hardware and embedded systems, several factors limit their applicability in building adaptive systems in highly dynamic computer environment where unpredictable radical changes are a common case. Hardware and embedded systems are usually well-defined and have predictable dynamics. The transition of their state is gradual, thus satisfies the continuity assumption of control-theory-based feedback systems. As a result, the feedback systems architecture supported by the toolkits does not provide sufficient means for meta-adaptation of the feedback systems themselves when system dynamics goes beyond the originally expected domain, or when jumps in system state happen. Traditionally, these toolkits are for design, simulation and analysis of hardware and embedded control systems. The designed control systems are then re-implemented in hardware, or in a low-level language in the case of embedded systems. Though latest versions of these toolkits began to generate code in C, C++ or Ada, the generated routines are tightly coupled with their run-time environments, or sub-optimized in execution performance. These limitations explain why they are not popular among software engineers. On the other hand, the software

feedback toolkit proposed in thesis is designed specially for use in highly dynamic and unpredictable software systems. The hierarchical feedback composition and guard-based meta-adaptation in our toolkit methodology facilitate the development of highly sophisticated wide-range feedback systems. The software feedback systems are implemented directly as C++ classes, and can be easily incorporated into software systems.

7.4 Existing Software Feedback In Adaptive Systems

Software feedback already exists in many forms in adaptive software systems. It is used for flow and congestion control [2, 4, 17, 23, 26, 30, 47, 55, 66], synchronization between Internet hosts [35], intra- and inter-stream synchronization in distributed multimedia systems [28, 48, 49, 50, 51], and multimedia presentation QoS adaptation [22, 24, 52, 53].

Software feedback has been used extensively in network flow and congestion control. TCP [2, 23] adjusts its congestion window size based on acknowledgements from the receiver to control data flow and avoid network congestion. It also continuously estimates the mean and variance of the round trip time (RTT) in order to adapt its retransmission timer to changing network conditions. Several other rate-based feedback flow and congestion control schemes have been proposed [4, 17, 26, 30, 55]. Some other schemes simply turn the source data flow on/off based on feedback from the routers, switches or receivers indicating network congestion [47, 66]. Several of the schemes have also been analyzed theoretically with control and queuing theories [17, 26, 47, 55, 66].

Software feedback also plays an important role in synchronization between Internet hosts, between the server and client of a multimedia stream, and between multiple media streams. The Network Time Protocol (NTP) [35], a protocol for clock synchronization between Internet hosts, has been widely deployed. On each host, this protocol uses various filters to identify a reliable remote time (reference clock) server and to estimate the frequency and phase error between the local clock and the reference clock. A phase-lock-loop (PLL) is then implemented to synchronize the local clock to the chosen reference clock. The PLL is parameterized to handle a broad range of clock drift. Feedback based

techniques have also been proposed for intra- and inter-stream synchronization in distributed multimedia streaming systems. Examples are the schemes proposed by Rangan in [48, 49, 50, 51] and that by Little and Ghafoor [28].

In multimedia streaming applications, software feedback has been used for effective dynamic control of the presentation quality. Jacobs and Eleftheriadis proposed an Internet video system architecture [22] in which the media pump at the server adapts the media quality (bit rate) to the available bandwidth through a technique called dynamic rate shaping. In Rowe's continuous media player [52, 53], Vosaic [12], and commercial streaming video players such as VXtreme [63], the video quality (usually video frame rate) presented to the user is continuously monitored by the client, and the measurements are fed back to the server to adjust the rate at which it streams out future video frames. In Jeffay's rate-based execution abstraction [24], Synthesis-style feedback-based scheduling [31] is used to allow multimedia application processes to adapt their pattern of execution based on the availability of resources.

While all the mechanisms above for system adaptation are based on software feedback, they are generally implemented in a custom manner, and hard-coded for particular applications. These feedback mechanisms are potential applications of the software feedback toolkit, which will help in systematic design, rapid prototyping, software reuse, simulation and instrumentation. As a demonstration of how the software feedback toolkit facilitates the development of adaptive systems, the design and implementation of SCP, a multimedia streaming flow and congestion control protocol, and a highly adaptive streaming video player have been presented in the thesis.

Chapter 8

Conclusions and Future Work

8.1 Summary of the Contribution

In this thesis, we have presented a software feedback toolkit and its application in adaptive multimedia systems. We proposed a methodology for hierarchical composition of complex feedback systems on top of simple building blocks, and introduced the concept of guard-based meta-adaptation for building wide-range feedback systems. We described an implementation of the toolkit in C++, with a library of components as C++ classes, and a set of tools for simulation and instrumentation. This software feedback toolkit provides a framework for development of feedback mechanisms in adaptive software systems. It facilitates the building of highly modular, adaptive and extensible feedback systems, and helps the reuse of existing feedback components. Then we showed how the software feedback toolkit is used to in the construction of adaptive real-time packet-rate control and flow and congestion control mechanisms for multimedia streaming, and an adaptive real-time distributed video player.

8.1.1 A Methodology For Software Feedback System Composition

In the software feedback toolkit, a complex feedback system is composed hierarchically from building blocks. To facilitate the hierarchical feedback composition, all feedback components export a common interface. Each component has a set of standard ports for feedback signal input and output, parameter change, internal state retrieval and update, and component reset. With this common interface, different feedback components, no

matter what functionalities they have, can be connected in a uniform way. Feedback systems can be built hierarchically out of existing building blocks and less complex composite components, which in turn are recursively composed of even simpler components.

The concepts of guard-based meta-adaptation, guarded feedback component, and dynamic component replugging are introduced for developing wide-range feedback systems used in highly dynamic computer environment in which unpredictable radical change in system state is a common case. An individual feedback policy (or algorithm) has a limited domain in which it is applicable. On the other hand, software systems, especially distributed multimedia systems across the Internet, have an unprecedented level of dynamics. These highly dynamic systems need to employ different feedback policies in different situations, and need to react properly and responsively in the face of unpredictable system state jumps. To facilitate development of complex wide-range feedback systems out of simple individual feedback policies, we make guard-based meta-adaptation explicit, and introduce the concept of guarded software feedback components. A wide-range feedback system is composed of multiple feedback policies, each of which is implemented as a guarded feedback component. The domain of the overall feedback system is the union of that of all the participating components. The domain of a guarded feedback component is guarded against events signaling that it is entered or left. Upon triggering of the guards, meta-adaptation actions are taken to switch from one feedback component to another one, in order to adapt to the changed environment. There are several types of meta-adaptation actions: light-weight parameter change or state reset on the active feedback policy components, heavy-weight replugging of existing policy components with new ones, or exceptions signalling to the application when the environment is out of the domain of the whole feedback system.

Guard-based meta-adaptation facilitates the application of control theories in software feedback systems. Feedback policies based on control theories, such as linear systems theory, are well defined, but have limited domains. With guard-based meta-adaptation, a wide-range feedback system can have multiple repluggable components implementing different linear policies, each of which is dynamically invoked when it is applicable. Though the linear components are simple, the overall feedback system can be powerful and highly

adaptive.

8.1.2 Implementation of the Software Feedback Toolkit

A prototype of the software feedback toolkit has been implemented in C++, with a component class library and a set of tools for simulation and instrumentation. In the component class library, base classes are defined for feedback components (in general) and composite feedback components. These base classes define a common interface consisting of a set of public member functions implementing the input, output, parameter, state and reset ports. They also implement methods for dynamic component replugging. The library provides a set of building blocks including various filters (lowpass filter, average median filter, integrator, difference filter etc.), regulators (gain unit, delay unit, biaser, etc.), and connectors (trigger, multiplexer, etc.). To facilitate feedback system simulation and instrumentation, the toolkit implements tools such as GUI-based control panel, parameter panel, oscilloscope, as well as signal generators and various components for file access. All the tools have the same interface as regular feedback components, thus they can easily be connected to the feedback systems to be simulated or instrumented on-line.

The contribution of the software feedback toolkit prototype has several aspects. The prototype demonstrates that the feedback-composition methodology can be reduced to practice. It provides a detailed design of components that others may want to follow. The toolkit prototype also makes the technology developed in the thesis directly available to software developers.

8.1.3 Application of the Toolkit in Adaptive Multimedia Systems

The software feedback toolkit has been used in the development of several feedback mechanisms for adaptive multimedia applications, such as an adaptive packet rate control mechanism, a network streaming flow and congestion control protocol, and feedback systems in a distributed adaptive video player for dynamic control of video quality and synchronization between clients and servers. These applications demonstrate the feasibility of the software feedback toolkit in developing adaptive systems.

The first application of the software feedback toolkit is for adaptive packet rate control

in real-time multimedia streaming. The target scenario is a server-client style unicast-media-streaming application, in which feedback is used at the client side to continuously monitor the quality of packet reception, and determine the rate at which the server sends future packets. Two types of network connections are identified: heavily buffered and lightly buffered. Heavily buffered connections have deep buffers in routers and switches. Upon network congestion, they show significant increase in delay before any packets are dropped. On the other hand, lightly buffered connections experience packet drop before a significant increase in packet delay can be detected. The rate control feedback system consists of two policies: a packet-latency feedback policy for heavily buffered network connections, and a packet-loss feedback policy for lightly buffered ones. These policies are implemented as repluggable guarded feedback components. Based on the network congestion events (packet-loss or increase in network delay), one or both of the policies are plugged into the feedback system dynamically. When both policies are active, the lower of the packet rates generated by the two policies is used by the server. Guarded feedback policies and dynamic replugging make the packet rate feedback adaptive to a wide range of network characteristics, and extensible when new network types are identified.

The feedback toolkit has also been applied in the development of SCP, a rate- and congestion-window-based flow- and congestion-control protocol for adaptive real-time multimedia streaming applications. SCP is designed to be TCP-friendly, while optimized for streaming applications. SCP consists of four guarded feedback policies corresponding to different network and streaming conditions (system state sub-domains): slow-start, steady-state, exponential back-off, and stream-pause. In the start-up phase, the slow-start policy is invoked to discover the available network bandwidth quickly. When the network is congested, the exponential back-off policy makes sure that the network is able to recover from the congestion quickly. When there is no congestion, SCP invokes the steady-state policy to maintain an appropriate amount of buffering inside the network connection, in order to sufficiently exploit available network bandwidth while keeping a low network latency level, as well as tracing the changes in network condition closely. After a pause in the stream, SCP shrinks its congestion window size and invokes the slow-start policy to detect the

already-changed network condition. The feedback policies are guarded against events signalling changes in network conditions and switches between the sub-domains. Upon these events, SCP performs meta-adaptation by dynamically switching between its feedback policies. SCP has built-in mobility awareness. Upon switches in the network interface, which may result in orders-of-magnitude change in network bandwidth and latency, SCP resets the state of all its estimators and invokes the slow-start policy to discover the new network environment quickly. Following the feedback systems composition methodology, SCP is designed in a highly modular and extensible manner. With the building blocks and tools in the toolkit, prototyping is accelerated, and performance instrumentation and fine-tuning are made easy. Internet experiments showed that SCP meets it design goals very well.

Finally, the software feedback toolkit greatly helps the development of the QoS and synchronization feedbacks in an adaptive real-time server-client style distributed streaming video player. SCP is used for flow and congestion control on the network connection between the server and the client. In addition, the player has two feedback systems for QoS control and client-server synchronization respectively. The QoS feedback takes the user-specified preference between video spatial resolution and frame rate, and adapts the video quality level to the available server disk bandwidth, network bandwidth, and client CPU processing power. Composed of high-order feedback and repluggable components, both the synchronization feedback and the QoS feedback are themselves highly adaptive. The synchronization feedback synchronizes the clock of the server to that of the client so as to maintain a minimum level of client-side buffering while ensuring smooth video playback. The buffering level is adapted to the network jitter level as observed by the synchronization feedback. In the presence of mobility events such as switches in network interfaces, the player adapts through guard-based meta-adaptation. SCP resets its state and switches to the slow-start policy. The QoS feedback requests the server to send at a high frame rate to discover the new pipeline bandwidth quickly. The synchronization feedback resets its estimators in order to discard invalid pipeline delay jitter and target buffer level estimations. As demonstrated by the Internet experiments, the three feedback systems (SCP, QoS and synchronization) collectively make the video player highly

adaptive, and robust in the highly dynamic and unpredictable Internet environment.

8.2 Future Work

A significant amount of work has been done on the software feedback toolkit. However, the research is far from being complete, a number of interesting ideas in all aspects (methodology, implementation and application) of the toolkit are to be further explored in the future. In fact, the thesis work and other research in the Distributed Systems Research Group (DSRG) of the Department of Computer Science and Engineering, Oregon Graduate Institute have inspired two new projects which started recently in the group. The first one is "*Microfeedbacks for Adaptive Resource Management*" [43], for further investigation of formal specification and automatic code-generation of high-performance wide-range software feedback systems. The other one is "*Systemic Quality of Service Support for Adaptive Distributed Systems*" [65], for specification of multimedia presentation QoS requirements, and automatic generation of appropriate feedback systems for adaptive resource management.

In the software feedback composition methodology, a more detailed model is to be developed for the specification of guards and meta-adaptation operations such as dynamic component replugging and exception processing. A high-level rigorous specification language is desirable for specification of feedback components and their composition. Executable code of feedback components then can be generated from their high level specifications. In particular, to facilitate the incorporation of control theories into the toolkit, a control-theory-friendly language is preferable for the specification of control-theory-based components and their composition. With a high-level specification language, it is expected that the properties (behaviors and performance) of many feedback components can be expressed or deduced in a somewhat formal manner.

In the software feedback toolkit prototype, more building blocks are to be implemented, especially those from control theories. It is possible to borrow many building blocks from control-theory-based toolkits such as Matlab and MATRIX_X. More sophisticated tools for simulation, instrumentation and theoretical analysis are needed. A GUI-based software feedback studio (development environment) is also to be developed. With the

studio, the users can define building blocks and compose feedback components in graphical form, and perform simulations. Feedback components in graphical form or in high level specification language would be automatically translated to C++ code by a code generator. During run-time, we would like to have a run-time code optimizer, similar to the Synthetix specialization tools, to eliminate the interpretation caused by dynamic composition of the feedback building blocks, so as to improve execution performance. Also, we are in the process of re-implementing the feedback toolkit in Java, in a multi-threaded and distributed environment.

For the application of the software feedback toolkit in adaptive multimedia systems, we are developing methods for translating QoS requirements into feedback systems for adaptive resource management. With a specification of the QoS requirements from the user, it is possible to determine the resources needed to produce the desirable presentation quality, the user's preference, and the adaptation actions to take when not sufficient resources are available. With this information, it is possible to select a set of well-understood and well-behaved feedback components and compose feedback systems which have the expected properties, and adapt the playback quality to the changes in available resources in a way consistent with the QoS specification.

It is also interesting to evaluate, through experiments, the adaptation mechanisms in our adaptive video player and that in other streaming video players, especially commercial ones such as Vxtreme [63], Real Video and Real Audio players [42], and Netscape streaming video players [37]. Some metrics that could be used for evaluation include: the dimensions and ranges of video quality a player can adapt; the ability to avoid network congestion; the response time on user-actions such as starting and pausing a stream, the response time in the presence of big jumps in available network bandwidth; the ability for multiple playback sessions to share network links; and the ability to live in harmony with TCP traffic. In Chapter 6, we have shown that our player is able to adapt the video quality across wide ranges in both the frame rate and spatial resolution dimensions. Our player is responsive to both user actions and drastic changes in network bandwidth. The use of SCP effective avoids network congestion, makes multiple playback sessions share the network bandwidth in a fair manner, and enables the player to live with TCP.

Bibliography

[1] BERNERS-LEE, T., FIELDING, R., AND NIELSEN, H. Hypertext transfer protocol – HTTP/1.0. Internet RFC 1945, `http://ds.internic.net/ds/rfc-index.html`, May 1996, [September 11, 1997].

[2] BRAKMO, L. S., ET AL. TCP Vegas: New techniques for congestion detection and avoidance. In *SIGCOMM'94* (August 1994), pp. 24–35.

[3] BROGAN, W. L. *Modern Control Theory.* Quantum Publishers, Inc., 1974.

[4] BUSSE, I., DEFFNER, B., AND SCHULZRINNE, H. Dynamic QoS control of multimedia applications based on RTP. *Computer Communications, 19* (January 1996), 49–58.

[5] CADZOW, J. A. *Discrete-Time Systems: An Introduction with Interdisciplinary Applications.* Prentice-Hall, Inc., 1973.

[6] CADZOW, J. A., AND MARTENS. *Discrete-Time and Computer Control Systems.* Prentice-Hall, Inc., 1974.

[7] CEN, S. Software feedback toolkit prototype user's manual. Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology. `http://cse.ogi.edu/DISC/projects/synthetix/FBT-manual.ps`, August 1997, [September 11, 1997].

[8] CEN, S. Software of a distributed real-time MPEG video player. Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology. `http://cse.ogi.edu/DISC/projects/synthetix/Player/`, August 1997, [September 11, 1997].

[9] CEN, S., PU, C., STAEHLI, R., COWAN, C., AND WALPOLE, J. Demonstrating the effect of software feedback on a distributed real-time MPEG video audio player. In *Proceedings of the Third ACM International Multimedia Conference and Exhibition* (San Francisco, CA, November 1995), pp. 239–240.

[10] CEN, S., PU, C., STAEHLI, R., COWAN, C., AND WALPOLE, J. A distributed real-time MPEG video audio player. *NOSSDAV'95, Lecture Notes in Computer Science 1018* (1995), 151–162. Springer-Verlag, 1995.

[11] CEN, S., PU, C., AND WALPOLE, J. Flow and congestion control for internet multimedia streaming applications. Tech. Rep. CSE-97-003, Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, June 1997.

[12] CHEN, Z., TAN, S.-M., CAMPBELL, R. H., AND LI, Y. Real time video and audio in the World Wide Web. In *Fourth International World Wide Web Conference* (Boston, Massachusetts, December 1995), pp. 15–27.

[13] CONSEL, C., AND DANVY, O. Tutorial notes on partial evaluation. In *In Proceedings of ACM Symposium on Principles of Programming Languages* (Charleston, South Carolina, January 1993), pp. 493–501.

[14] COOK, P. A. *Nonlinear Dynamical Systems.* Prentice-Hall International (UK) Ltd., 1986.

[15] COWAN, C., ANTREY, T., KRASIC, C., PU, C., AND WALPOLE, J. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the International Conference on Configurable Distributed Systems* (Annapolis, Maryland, May 1996), pp. 108–115.

[16] DAYHOFF, J. E. *Neural Network Architectures: an Introduction.* Van Nostrand Reinhold, 1990.

[17] FENDICK, K. W., RODRIGUES, M. A., AND WEISS, A. Analysis of a rate-based control strategy with delayed feedback. In *Proceedings of SIGCOMM'92* (August 1992), pp. 136–147.

[18] FLOYD, S., AND FALL, K. Router mechanisms to support end-to-end congestion control. Network Research Group, Lawrence Berkeley National Laboratory. `ftp://ftp.ee.lbl.gov/papers/collapse.ps`, February 1997, [September 11, 1997].

[19] GOODWIN, G. C., AND SIN, K. S. *Adaptive Filtering Prediction and Control.* Prentice-Hall, 1984.

[20] INOUYE, J., CEN, S., PU, C., AND WALPOLE, J. System support for mobile multimedia applications. In *NOSSDAV'97* (May 19-21 1997), pp. 143–154.

[21] INTEGRATED SYSTEMS, INC. $MATRIX_X$ family technical specification. `http://www.isi.com /Products/MATRIXx/ Techspec/toc.html`, 1997, [September 11, 1997].

[22] JACOBS, S., AND ELEFTHERIADIS, A. Adaptive video applications for non-QoS networks. In *International Workshop on Quality of Service'97* (Columbia University, New York, May 1997), pp. 161–166.

[23] JACOBSON, V. Congestion avoidance and control. In *SIGCOMM'88* (August 1988), pp. 79–88.

[24] JEFFAY, K., AND BENNETT, D. A rate-based execution abstraction for multimedia computing. In *Proceedings of NOSSDAV'95* (April 1995), pp. 67–77.

[25] JONES, G. *Programming in OCCAM 2.* Prentice-Hall International, 1988.

[26] KESHAV, S. A control-theoretic approach to flow control. In *SIGCOMM'91* (Sept. 1991), pp. 3–16.

[27] LIPPMAN, S. B. *C++ Primer 2nd Edition.* Addison-Wesley Publishing Company, 1991.

[28] LITTLE, T. D. C., AND GHAFOOR, A. Multimedia synchronization protocols for broadband integrated services. *IEEE Journal on Selected Areas in Communication, 9*, 9 (Dec. 1991), 1368–1382.

[29] LJUNG, L., AND SODERSTROM, T. *Theory and Practice of Recursive Identification.* MIT Press, 1993.

[30] MAHDAVI, J., AND FLOYD, S. TCP-friendly unicast rate-based flow control. In end2end mailing list, `ftp://ftp.isi.edu/end2end`, January 1997, [September 11, 1997].

[31] MASSALIN, H. *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Graduate School of Arts and Science, Columbia University, 1992.

[32] MASSALIN, H., AND PU, C. Fine-grain adaptive scheduling using feedback. *Computing Systems 3*, 1 (Winter 1990), 139–173.

[33] MCCANNE, S., AND JACOBSON, V. *vic:* a Flexible Framework for Packet Video. In *Proceedings of the Third ACM Conference and Exhibition (Multimedia '95)* (San Francisco, California, November 1995), pp. 511–522.

[34] McNeill, F. M., and Thro, E. *Fuzzy Logic: a Practical Approach.* Boston: AP Professional, 1994.

[35] Mills, D. L. Network time protocol (version 3) specification, implementation and analysis. Tech. rep., University of Delaware, 1992. DARPA Network Working Group Report RFC-1305, `http://ds.internic.net/ds/rfc-index.html`, March 1992, [September 11, 1997].

[36] Mosberger, D., Peterson, L. L., Bridges, P. G., and O'Malley, S. Analysis of techniques to improve protocol processing latency. In *SIGCOMM'96* (October 1996), pp. 73–84.

[37] Netscape Communications Corporation. Netscape plug-ins: Audio/video. `http://home.netscape.com/comprod/products/navigator/version_2.0/plugins/audio-video.html`, 1997, [September 11, 1997].

[38] Oregon Graduate Institute, Department of Computer Science and Engineering. The Synthetix Project at OGI. Oregon Graduate Institute of Science and Technology, `http://www.cse.ogi.edu/DISC/projects/synthetix/`, 1994-1997, [September 11, 1997].

[39] Papoulis, A. *Probability, Random Variables, and Stochastic Process.* McGraw-Hill, Inc., 1991.

[40] Paxson, V. End-to-end routing behavior in the Internet. In *SIGCOMM'96* (Stanford University, California, August 1996), pp. 25–38.

[41] Postel, J., and Reynolds, J. File transfer protocol. Internet RFC 0959, `http://ds.internic.net/ds/rfc-index.html`, October 1985, [September 11, 1997].

[42] Progressive Networks. HTTP versus RealAudio client-server streaming. `http://www.realaudio.com/help/content/http_vs_ra.html`, 1996, [September 11, 1997].

[43] Pu, C. Microfeedbacks for adaptive resource management. `http://www.cse.ogi.edu/DISC/projects/microfeedback/microfeedback.html`, March 1997, [September 11, 1997].

[44] Pu, C., Autrey, T., Black, A., Consel, C., Cowan, C., Inouye, J., Kethana, L., Walpole, J., and Zhang, K. Optimistic incremental specialization: Streamlining a commercial operating system. In *SOSP'95* (Copper Mountain Resort, Colorado, December 1995), pp. 314–324.

[45] PU, C., AND FUHRER, R. M. Feedback-based scheduling: a toolbox approach. In *Fourth Workshop on Workstation Operating Systems* (October 1993), pp. 124–128.

[46] PU, C., MASSALIN, H., AND LOANNIDIS, J. The synthesis kernel. *Computing Systems, 1*, 1 (Winter 1988), 11–32.

[47] RAMAKRISHNAN, K. K., AND JAIN, R. A binary feedback scheme for congestion avoidance in computer networks. *ACM Transactions on Computer Systems, 8*, 2 (May 1990), 158–181.

[48] RAMANATHAN, S., AND RANGAN, P. V. Adaptive feedback techniques for synchronized multimedia retrieval over integrated networks. *IEEE/ACM Transactions on Networking, 1*, 2 (April 1993), 246–260.

[49] RAMANATHAN, S., AND RANGAN, P. V. Feedback techniques for intra-media continuity and inter-media synchronization in distributed multimedia systems. *The Computer Journal 36*, 1 (Feb. 1993), 19–31.

[50] RANGAN, P. V., RAMANATHAN, S., AND SAMPATHKUMAR, S. Feedback techniques for continuity and synchronization in multimedia information retrieval. *ACM Transactions on Information Systems, 13*, 2 (April 1995), 145–176.

[51] RANGAN, P. V., RAMANATHAN, S., VIN, H. M., AND KAEPPNER, T. Techniques for multimedia synchronization in network file systems. *Computer Communications Journal 16*, 3 (Mar. 1993), 168–176.

[52] ROWE, L. A., ET AL. MPEG video in software: Representation, transmission and playback. In *Symposium on Elec. Imaging Sci. and Tech.* (San Jose, California, February 1994), pp. 15–26.

[53] ROWE, L. A., AND SMITH, B. C. A continuous media player. In *Proceedings of the 3rd International Workshop on Network and Operating System Support for Digital Audio and Video* (San Diego, California, November 1992), pp. 376–386.

[54] SCHULZRINNE, H., CASNER, S., FREDERICK, R., AND JACOBSON, V. RTP: A transport protocol for real-time applications. Internet RFC 1889, `http://ds.internic.net/ds/rfc-index.html`, January 1996, [September 11, 1997].

[55] SHENKER, S. A theoretical analysis of feedback flow control. In *Proceedings of SIGCOMM'90* (September 1990), pp. 156–165.

[56] SPIEGEL, M. R. *Theory and Problems of Probability and Statistics.* McGraw-Hill, Inc., 1975.

[57] STAEHLI, R. *Quality of Service Specification for Resource Management in Multimedia Systems.* PhD thesis, Department of Computer Science and Technology, Oregon Graduate Institute of Science and Technology, January 1995.

[58] STAEHLI, R., WALPOLE, J., AND MAIER, D. Quality of service specification for multimedia presentations. *Multimedia Systems, 3*, 5/6 (November 1995), 251–263.

[59] THE MATHWORKS, INC. Matlab product tour. `http://www.mathworks.com/products.html`, 1997, [September 11, 1997].

[60] THE MATHWORKS, INC. *The Student Edition of MATLAB Version 4 User's Guide.* Prentice-Hall, Inc., 1995.

[61] THE MATHWORKS, INC. *The Student Edition of SIMULINK User's Guide.* Prentice-Hall, Inc., 1996.

[62] TORVALDS, L. LINUX kernal source code version 2.0.18. `http://www.kernel.org/pub/linux/kernel/v2.0/linux-2.0.18.tar.gz`, September 1996, [September 11, 1997].

[63] VXTREME, INC. Vxtreme streaming video player. `http://www.vxtreme.com`, 1997, [September 11, 1997].

[64] WACHSMANN, A., AND WICHMANN, F. OCCAM-light: A multiparadigm programming language for transputer networks. Tech. Rep. tr-rf-93-005, U-Gesamthochschule Paderborn, CS, April 93.

[65] WALPOLE, J. Systemic quality of service support for adaptive distributed systems. `http://www.cse.ogi.edu/DISC/projects/qos/QoS.html`, March 1997, [September 11, 1997].

[66] WANG, Y. T., AND SENGUPTA, B. Performance analysis of a feedback congestion control policy. In *Proceedings of SIGCOMM'91* (September 1991), pp. 149–157.

[67] WHITE, D. A., AND SOFGE, D. A. *Handbook of Intelligent Control: Neural, Fuzzy and Adaptive Approaches.* Multiscience Press, Inc., 1992.

Appendix A

Test of Significance of the Difference Between Two Experiments

In this Appendix, we briefly explain a method to test if the difference between the means of two experiments is statistically significant. Please refer to Spiegel's textbook [56] for a more detailed explanation.

Suppose that there is a set of measurement samples for each of the two experiments. The measurements of the two experiments have normal distributions. Set 1 (for experiment 1) has n_1 samples, a sample mean $\bar{m}_1$ and a sample variance s_1^2. Set 2 (for experiment 2) has n_2 samples, a sample mean $\bar{m}_2$ and a sample variance s_2^2. Also suppose that $\bar{m}_1 < \bar{m}_2$. We want to test if the real means of these two experiments, η_1 (for experiment 1) and η_2 (for experiment 2) are significantly different, with confidence coefficient γ ($0 \leq \gamma \leq 1$) or confidence level $\delta = 1 - \gamma$. We have higher confidence when γ is closer to 1.

We have two hypothesis: $H_0 : \eta_1 = \eta_2$, and $H_a : \eta_1 < \eta_2$. The hypothesis $\eta_1 > \eta_2$ does not make sense since we already have $\bar{m}_1 < \bar{m}_2$. We will see if H_0 is rejected with a confidence level δ. If H_0 is rejected, we would be about γ confident that the difference in means is significant.

To test the hypothesis, we compute:

$$s_p{}^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{\nu}$$

Where

$$\nu = (n_1 - 1) + (n_2 - 1) = n_1 + n_2 - 2$$

is the aggregate degree of freedom for the two sets. A set of n samples has a degree of freedom equal to $(n-1)$, since given the sample average and any $(n-1)$ samples, the only remaining sample can always be calculated, and thus is not free.

Next, we compute following student-t value:

$$t = \frac{|(\bar{m}_1 - \bar{m}_2) - (\eta_1 - \eta_2)_0|}{\sqrt{\frac{s_p^2}{n_1} + \frac{s_p^2}{n_2}}} = \frac{|\bar{m}_1 - \bar{m}_2|}{\sqrt{\frac{s_p^2}{n_1} + \frac{s_p^2}{n_2}}}$$

Finally, we compare the above t against $t_{\gamma,\nu}$ — a student-t value looked up in Appendix D on page 346 of Spiegel's textbook [56]. If $t \leq t_{\gamma,\nu}$ then H_0 holds and $\eta_1 = \eta_2$ with γ confidence, otherwise $\eta_1 < \eta_2$.

Appendix B

Building Blocks in the Software Feedback Toolkit Component Library

In this Appendix, we give a list of control-related building blocks implemented in the software-feedback-toolkit component library. The building blocks include various filters, regulators, and connectors. A complete list of all the feedback components implement in the toolkit library, including all the GUI-based panel components, can be found in the toolkit user's manual [7].

For each component, we describe its constructor and port configuration, and discuss its control function. All the filters and regulators listed in this Appendix have one input port and one output port and a reset port. Some may also have parameter or state ports. For these components, we assume that the input and output sequences are $\{u(k)\}$ and $\{y(k)\}$ respectively, where k is an integer, and for all $k < 0$, $u(k) = 0$ and $y(k) = 0$.

B.1 Filters

`DifferenceFilter()`

`DifferenceFilter` generates the difference between the two most recent input samples. It has one input port, one output port, and one state port. It implements a difference function shown below.

$$y(k) = u(k) - u(k-1) \quad \text{for all } k \geq 0$$

The exported state variable holds the most recent input value.

`IntegratorFilter()`

`IntegratorFilter` generates the integration (sum) of all past input samples. It has one input port, one output port, and one state port. It implements a difference function shown

below.

$$y(k) = u(k) + y(k-1) \quad \text{for all } k \geq 0$$

The exported state variable holds the most recent output value.

`FOLowPassFilter(double a = 1.0)`

`FOLowPassFilter` applies a first-order lowpass filtering algorithm on its input samples. It has one input port, one output port, one parameter port, and one state port. It implements a difference function shown below, with a time-constant coefficient a ($0 \leq a \leq 1.0$).

$$y(k) = au(k) + (1.0 - a)y(k-1)$$

The exported state variable holds the most recent output value. The parameter a is exported through the parameter port. Its initial value is set by the constructor, with a default value of 1.0.

B.2 Regulators

`Abs()`

`Abs` generates the absolute values of its input samples. It has one input port and one output port, and implements a difference function shown below.

$$y(k) = \begin{cases} u(k) & \text{if } u(k) \geq 0 \\ -u(k) & \text{if } u(k) < 0 \end{cases}$$

`Biaser(double B = 0.0)`

`Biaser` adds a constant, B, to its input sequence to generate its output sequence. It has one input port, one output port, and one parameter port. It implements a difference function shown below.

$$y(k) = u(k) + B$$

The parameter B is exported through the parameter port. Its initial value is set by the constructor, with a default value of 0.

`Delay(int D = 0)`

`Delay` buffers its input samples a certain units of time before outputting them. The units of time to delay is specified by the integer part of a parameter D ($\lfloor D \rfloor$), where $D \geq 0$. `Delay` implements a difference function shown below.

$$y(k) = u(k - \lfloor D \rfloor)$$

The parameter D is exported through the parameter port. Its initial value is set by the constructor, with a default value of 0.

`Gain(double G = 1.0)`

`Gain` multiplies each of its input samples with a gain, G, to generate its output samples. It has one input port, one output port, and one parameter port. It implements a difference function shown below.

$$y(k) = Gu(k)$$

The parameter G is exported through the parameter port. Its initial value is set by the constructor, with a default value of 1.0.

`Inverter()`

`Inverter` inverts each of its input samples to generate its output sequence. It has one input port and one output port. It implements a difference function shown below.

$$y(k) = \begin{cases} \frac{1}{u(k)} & \text{if } u(k) \neq 0 \\ \infty & \text{if } u(k) = 0 \end{cases}$$

`Limiter(double L = MAXDOUBLE)`

`Limiter` limits the magnitude (absolute value) of the samples it passes to be no more than L, where $L \geq 0$. It has one input port, one output port, and one parameter port. It implements a difference function shown below.

$$y(k) = \begin{cases} -L & \text{if } u(k) < -L \\ u(k) & \text{if } -L \leq u(k) \leq L \\ L & \text{if } u(k) > L \end{cases}$$

The parameter L is exported through the parameter port. Its initial value is set by the constructor, with a default value of `MAXDOUBLE` (∞).

`Quantizer(double S = 0.0)`

`Quantizer` quantizes its input samples with a step-width of S, where $S \geq 0$. It has one input port, one output port, and one parameter port. It implements a difference function shown below.

$$y(k) = \begin{cases} u(k) & \text{if } S = 0 \\ \lfloor \frac{u(k)}{S} \rfloor * S & \text{if } S > 0 \end{cases}$$

The parameter S is exported through the parameter port. Its initial value is set by the constructor, with a default value of 0.0.

B.3 Connectors

`Merger(int N = 2, char * P = "+-")`

`Merger` merges all its input sequences into a single output sequence. It has N input ports and one output port. The number of input ports, N, is specified by the first parameter in its constructor, with a default value of 2. The second parameter of the constructor, P, specifies the way the input sequences are merged. P is a character streaming containing N '+'s or '−'s. Thus each input port corresponds to a sign '+' or '−'. P has a default value of "+−". The most recent samples presented to input ports other than the first one (input port 0) are latched in input array `input[]`. Whenever input port 0 receives a sample, it also latches the sample. Besides, `Merger` sums up all the latched samples for all input ports into an output sample. If an input port corresponds to a '−' in P, then the corresponding input value has its sign changed before being summed. Following equation shows the operations `Merger` performs upon a input from its input port 0.

$$\texttt{output[0]} = \sum_{i=0}^{i<N} (\pm)\texttt{input[i]}$$

`Merger` with the default parameters can be used to connect components into feedback loops. Figure 3.10 in Section 3.5 shows an application of `Merger` in a PLL simulator.

`Trigger(int N = 1)`

`Trigger` is an analogue to a latch circuit, which latches all its input signals until a trigger happens. It has $N + 1$ input ports and N output ports. The parameter N ($N \geq 1$) is specified by the constructor, with a default value of 1. Among the input ports, port 0 through $N - 1$ are data ports, and port N is a trigger port . The samples presented to input ports 0 through $N - 1$ are latched. When a signal is received the trigger port (input port N), all the latched samples on input port 0 through $N - 1$ are sent to output port 0 through $N - 1$ respectively.

`Trigger` is useful in implementing feedback simulators. It has been used in the flow control feedback simulator shown in Fig. 3.15 of Section 3.6.

Biographical Note

Shanwei Cen is from Hunan, China. From 1982 to 1989, he attended Tsinghua University in Beijing, China, and received his B.S. and M.S in 1987 and 1989 respectively, both in Computer Science. After that he spent three years as a member of faculty in the same University. In this position, he did research on the implementation of declarative programming languages on distributed systems, supervised undergraduate and graduate students, and taught courses. In 1992, he joined a start-up company in Beijing, where he led a group of software engineers to develop networked Chinese paging systems, telephone switchboard accounting systems, and Chinese word processors. In search of deeper understanding of computer science, in 1993 he enrolled in the PhD program in the Department of Computer Science and Engineering at Oregon Graduate Institute of Science and Technology. While continuing working hard towards his thesis, in 1995 he did an internship at Tektronix, Inc. in Portland, Oregon, and since then he has been working part-time at Tektronix as a software engineer. His current research interests include adaptive multimedia systems, distributed and real-time operating systems, and networking.