Copyright © 1996, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

RAPID PROTOTYPING AND DEPLOYMENT OF USER-TO-USER NETWORKED APPLICATIONS

by

Wan-teh Chang

Memorandum No. UCB/ERL M96/95

18 December 1996

RAPID PROTOTYPING AND DEPLOYMENT OF USER-TO-USER NETWORKED APPLICATIONS

Copyright © 1996

by

Wan-teh Chang

Memorandum No. UCB/ERL M96/95

18 December 1996

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Rapid Prototyping and Deployment of User-to-User Networked Applications

by

Wan-teh Chang

Doctor of Philosophy in Engineering – Electrical Engineering
and Computer Sciences

University of California at Berkeley

Professor David G. Messerschmitt, Chair

User-to-user networked applications provide shared functionality to two or more human users distributed across a network. Examples include general telecommunications applications (voice telephony, video conferencing, etc.) and collaborative applications (shared whiteboards, shared editors, etc.). Compared with stand-alone or user-to-server applications, there exist relatively few user-to-user applications today. I argue that this is due to the unique difficulties in their *design* and *deployment*. To encourage a proliferation of user-to-user applications, this dissertation addresses these two difficulties.

I propose a heterogeneous approach for the rapid prototyping of user-to-user applications. This approach uses the most suitable models of computation to design different portions of an application, and combine them. User-to-user applications typically require intricate distributed control to handle the user interface, networking, and user-to-user interaction, which adds significant complexity to the design task. Therefore, I focus on the design of control. I study formal models with hierarchical finite state-machine semantics

for specifying complex control, and the semantic issues in mixing the control models with concurrency models such as dataflow. I describe an implementation in the Ptolemy simulation environment with examples.

After new user-to-user applications have been developed, their deployment is still hindered by what the economists call the *network externality* problem, which means that the utility of an application to one user increases with the number of other users who also have that application. Network externality makes an application of little value to early adopters, and hence is a major obstacle to developing and deploying such applications. With high-speed networking, software-defined applications can be transferred quickly from a repository to programmable terminals at session establishment and during the session, thus bypassing network externality. I study this *dynamic network deployment* approach, and describe a design of the system architecture and session establishment protocols based on the standard Java and World Wide Web framework.

David G. Messerschmitt 12/16/96

Professor David G. Messerschmitt, Chair Date

Contents

Preface

This dissertation marks the completion of my graduate study at the University of California at Berkeley, a period of my life filled with the joy of learning in the best academic institution, with the best minds, under the blue, sunny Californian sky. I have been fortunate enough to conduct my Ph.D. research under the guidance of Prof. David Messerschmitt. A Bell Labs researcher before joining the Berkeley faculty, he suggested that I study the design and architecture issues of telecommunications control software. Indeed, the telecommunications software system is the largest distributed software system in the world and is enormously complex. It definitely needs a disciplined design method and a carefully crafted architecture to allow it to evolve and progress rapidly.

Network control, i.e. signaling and call processing, is the coordination between terminals and networks for setting up and managing network connections. Network control software has always been very complex, and in the future we would only expect it to become even more complex as terminals, networks, and applications are becoming more and more heterogeneous. How should the network-terminal coordination proceed in order to enable terminals, networks, and applications to configure themselves, interoperate efficiently, and adapt themselves to changing conditions? To achieve these new objectives, new functions must be added to the network control software. How should these network control functions be organized? In the first chapter of this dissertation, I express my opinions on these issues. Although these are far from complete and final, and are bound to be controversial, they serve as a framework for the two specific issues I am going to delve into in depth in my dissertation.

The two issues I have addressed are the design and deployment of user-to-user networked applications. User-to-user networked applications include the general telecommunications applications (voice telephony, video conferencing, etc.) and collaborative applications (shared whiteboards, shared editors, etc.). A user-to-user application provides shared functionality to two or more human users distributed across a network. There are not that many user-to-user networked applications today, compared with stand-alone applications or user-to-information-server applications. I contend that this is because the design and deployment of user-to-user applications face difficulties that do not exist or are far less severe for stand-alone and user-to-information-server applications. One especially serious obstacle to the deployment of user-to-user applications is network externality. Network externality is the economist's term for the phenomenon that the utility of an application to one user increases with the number of other users who also have that application. Network externality makes an application of little use to early adopters. The failure of AT&T's attempt to deploy Picturephone in the 1970s was exactly due to this. Who would buy a Picturephone when there were not many other owners of Picturephones with whom to communicate? Network externality has been a great obstacle to innovations in telecommunications.

To encourage a proliferation of user-to-user applications, we should first minimize their development effort, and then make their deployment easy. This dissertation proposes a rapid prototyping methodology for user-to-user applications and a rapid deployment approach called dynamic network deployment.

Chapter Organization and Overview

The dissertation consists of five chapters. Chapter 1 is the foundation for the subsequent chapters. It first standardizes the somewhat divergent terminology used by the telecommu-

nications and computer networking communities, and then lays out a framework for terminal-network coordination for networked applications to deal with heterogeneity and exploit flexibility. Then it motivates the two issues addressed by this dissertation – a rapid prototyping methodology and a dynamic network deployment approach for user-to-user applications.

Chapter 2 describes a heterogeneous approach to rapid prototyping based on superimposing models of computation on general-purpose programming languages and mixing models of computation. It focuses on the design support for complex control functionality, which is common in application and service configuration in telecommunications networks and terminals. I describe specialized models of computation for specifying control, in particular one based on hierarchical finite state machine semantics. The control model can be mixed with other concurrency models such as dataflow, synchronous/reactive systems, and discrete event. I discuss the semantic issues on the interaction between models.

In Chapter 3, I describe the implementation of the results of Chapter 2 in the Ptolemy software environment. I also express opinions drawn from the experience with using this approach to design example applications.

Chapter 4 describes the dynamic network deployment approach, which transfers software-defined user-to-user applications over the network to programmable terminals dynamically at session setup. This approach bypasses the network externality problem, a major obstacle to the rapid deployment of user-to-user applications, and should encourage a proliferation of this class of networked applications. I first investigate this approach in general, and then work out a design, including the system architecture and protocols, based on the standard, widely disseminated framework of Java and World Wide Web HTTP (HyperText Transfer Protocol) servers and browsers.

Chapter 5 presents the concluding remarks and points out open issues.

Note that the rapid prototyping methodology presented in Chapters 2 and 3 and the dynamic deployment approach presented in Chapter 4 can be read independently. Moreover, Chapter 2 is in fact applicable to other application domains as well, for example embedded system design.

Acknowledgments

I would like to thank my research advisor, Prof. David Messerschmitt, for his guidance of my research. I have learned a lot from his experience, visions, and unique perspectives. Some ideas in this dissertation were originally inspired by him. I would also like to thank Prof. Edward Lee, who is practically a second advisor to me, for his guidance of my work on the rapid prototyping methodology. I would like to thank Prof. John Rhodes of the Mathematics Department for serving on my dissertation committee and reading my dissertation. I would like to thank Stephen Edwards, Brian Evans, Alain Girault, Weiyi Li, and José Pino for their careful review and honest critique of my dissertation. The work presented in this dissertation has benefited greatly from discussions with Joe Buck, Stephen Edwards, Alain Girault, Bilung Lee, Weiyi Li, and Kennard White. I would like to thank everyone in Dave's and Edward's research groups for the collaboration. I would like to thank my parents and the Semiconductor Research Corporation (SRC) for their financial support for my graduate study.

Berkeley, California *Wan-teh Chang*

December 1996

Vita

Wan-teh Chang (張萬德) was born on August 30, 1966 in Tainan, Taiwan, Republic of China and spent his boyhood in the suburb of Kaohsiung. He received a B.S. in electrical engineering from National Taiwan University in 1987. After two years' military service in the army, he came to the US for graduate study at the University of California at Berkeley. His intellectual interests are mathematics, computer programming, and linguistics.

1

Framework

1.1 Introduction

User-to-user networked applications provide audio-visual communication or some other shared functionality between two or more human users distributed across a network. Examples of user-to-user applications include general telecommunications applications (e.g., voice telephony, videophone, video conferencing, facsimile, and modem) and collaborative applications (e.g., shared whiteboards and shared editors) [1][2]. The number of user-to-user networked applications available today is small compared with stand-alone or user-to-information-server applications. Although the telecommunications industry has existed for over 120 years, we still have relatively few telecommunications applications. (Nevertheless, these telecommunications applications are of broad interest and very successful.) In sharp contrast, anyone walking into a computer software store will be amazed by the large number and wide variety of stand-alone applications available for personal computers. In the past decade we have seen some networked applications emerging on the Internet, such as electronic mail, newsgroups, file transfer, gopher, World Wide Web browsing [3][4][5], *talk*, *chat* [6], and groupware [7], but their number is still small compared with the number of stand-alone applications, and most of them are user-to-informa-

tion-server, rather than user-to-user applications. (Chapter 7 in Tanenbaum's text is a good introduction to many of these Internet applications [8].)

Why are there so few user-to-user applications? Perhaps inherently there are just fewer compelling user-to-user applications. However, I argue that this is because user-to-user applications face unique design and deployment difficulties that do not exist for stand-alone or user-to-information-server applications, so that they have been avoided by application developers. To rectify this situation, I propose a rapid prototyping methodology for user-to-user applications, addressing what is particularly difficult about their design, and suggest a new method to deploy a user-to-user application that avoids the obstacles to the traditional way of deploying them. These points will be elaborated in Section 1.6. As a preparation, I first standardize the terminology and express my opinion on what should happen during the execution of a networked application. This will serve as a framework for subsequent discussion.

1.2 Taxonomy of Networked Applications

I have used the terms *networked application*, *user-to-user*, and *user-to-information-server* without formally defining them. What exactly is a networked application, and how is a user-to-user application different from a user-to-information-server application? In this section I would like to classify networked applications and standardize the usage of terminology in this dissertation. Although the fields of telecommunications and computing are converging, the distinct terminologies used by the two communities for the same or similar ideas are often a barrier to the communication and healthy cross-fertilization between these two fields. Therefore, it is instructive to establish a standard terminology. I follow the terminology and taxonomy of networked applications proposed by Messerschmitt in

his paper on the convergence of telecommunications and computing [9]. (There is an earlier, condensed version [10].) The taxonomy is similar to that proposed by Fluckiger [11].

An *application* is a collection of functionality that provides value to a human user. A *networked* (as opposed to *stand-alone*) application has its execution distributed across a distributed networking and computing environment. Because this dissertation is concerned with networked applications exclusively, the word "networked" is often omitted where there is no possibility of confusion. I use the layered network model proposed by Messerschmitt, illustrated in Figure 1.1 [9]. (This network model is a simplified version of that proposed by the National Research Council [12].) In this layered model, applications are at the top level. Examples of networked applications are telephony, video conferencing, electronic mail, database access, file transfer, and World Wide Web (WWW or Web) browsing [3][4][5]. Below applications are the *services*, which are the more generic functions provided by terminals and networks to the applications. Services include the transport of various types of data (audio, video, data, etc.) by protocols such as Transmission Control Protocol (TCP) [13][14] and Real-time Transport Protocol (RTP) [15], and the *middleware* services [16] such as directory service and encryption. At the bottom layer, the *bitways* are the underlying conduits of bits, including network technologies such as

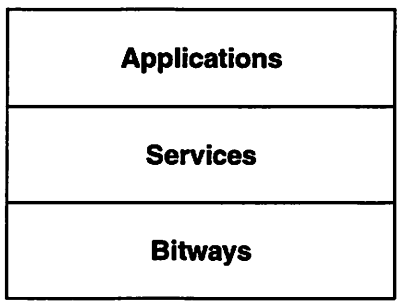


Figure 1.1 The layered network model.

Asynchronous Transfer Mode (ATM) [17][18][19], wireless access [20][21][22], Ethernet [23], etc. and internetworking protocols such as Internet Protocol (IP) [13][14].

Networked applications can be divided into two classes according to the number of users involved (see Figure 1.2).

- *User-to-user* applications, which involve two or more users. Typical user-to-user applications are general telecommunications applications, such as voice telephony, video conferencing, electronic mail, and voice mail, or collaboration of users in some shared task, such as the computer-aided design of a system or generation of a proposal or paper.
- *User-to-information-server* applications, in which a human user interacts with an information server computer. Examples of user-to-information-server applications include video on demand, WWW browsing, and file transfer.

In the temporal dimension, networked applications can be divided into *immediate* applications, whose user participation is at the same moment, and *deferred* applications, whose user participation can be delayed to a later time. For example, voice telephony is an immediate application. The two parties in a phone call must participate simultaneously. On the other hand, voice mail is a deferred application. The sender leaves a message in the voice mail server (usually when the receiver is not available to answer a phone call), and the receiver plays back the message at a later time.

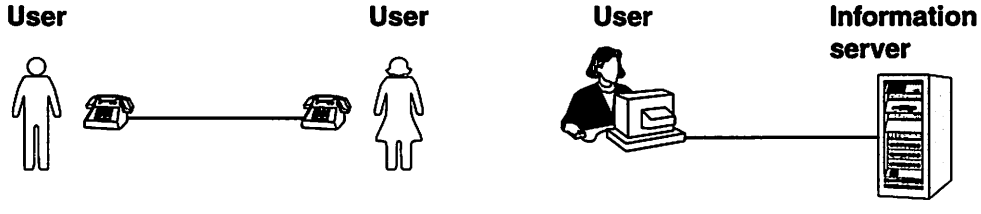


Figure 1.2 User-to-user and user-to-information-server applications.

Networked applications have the following two implementation architectures, shown in Figure 1.3:

- *Peer-to-peer* architecture, in which two users each interact with *peer computers* or *terminals*, which in turn communicate over the network to provide shared functionality to the two users. There are no central servers in this architecture. This architecture matches user-to-user applications naturally. For example, the *talk* program on Unix workstations uses the peer-to-peer architecture to implement a user-to-user text-based communication application.
- *Client-server* architecture, in which a user interacts with a *client computer* or *terminal*, which in turn communicates over the network with a *server computer* [24][25]. The server is not directly associated with another user, but rather provides services or functionality to the remote user. Usually the server implements the main functionality, and the client is simple, primarily implementing the user interface. The centralized server can usually handle multiple clients simultaneously. Clearly, a user-to-information-server application must be implemented with a client-server architecture, but a user-to-user application can also be implemented with a client-server architecture. For exam-

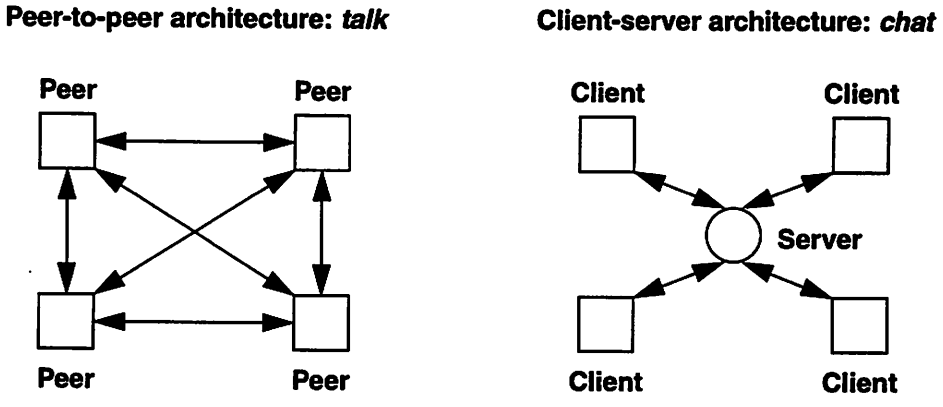


Figure 1.3 Peer-to-peer and client-server architectures.

ple, Internet Relay Chat (IRC) uses the client-server architecture to implement a user-to-user text-based communication application [6]. Another example is CU-SeeMe for video communication on the Internet [26][27][28].

Whereas the primary distinction between peer-to-peer and client-server architectures is whether a central server is involved, the detailed implementation characteristics typically differ as follows:

- Peer-to-peer architecture is typically fairly symmetric between the two peers with respect to network traffic, while client-server architecture is typically asymmetric. A server usually generates more network traffic than the client does.
- A server does not originate session establishment requests, but must be willing to receive a request at any time, and thus needs to be running at all times. A client only originates establishment requests at the initiative of the user, and thus only needs to be running at the behest of the user. Peers are an intermediate case: each peer must respond to establishment requests as well as initiate them, and must be running anytime the user is available to participate in an application. Therefore, a peer can be viewed as a hybrid of a client and a server.

1.3 Execution of Networked Applications

Having defined the terminology, I now discuss the issues in the execution of networked applications in a heterogeneous environment. A networked application may need to run on different kinds of terminals, such as telephones, desktop computers, and hand-held mobile devices. The terminals may send and receive different kinds of traffic (audio, video, data, etc.) over a concatenation of different kinds of network transport (broadband ATM networks, wireless access, etc.). The network architecture and protocols should be able to

handle the heterogeneity and exploit the flexibility in applications, terminals, and network transport. In this section I survey the current work in this active area of research.

1.3.1 Increasing Heterogeneity and Flexibility

The traditional telecommunications network is an infrastructure dedicated to the provision of only one application, namely voice telephony, which is functionally simple and of universal interest. There is some heterogeneity in that the network elements are manufactured by multiple equipment vendors and operated by multiple service providers, but it is a relatively homogeneous environment. In the future we will see increasing heterogeneity in applications, network transport, and terminals. The applications will have more complex functions, and typically will combine multiple media such as voice, images, video, and data that have different traffic characteristics and quality-of-service requirements. Conference and collaborative applications will have multipoint participants, giving rise to more complex connections. The terminals vary widely in their display and processing capabilities, ranging from powerful desktop computers to more restricted hand-held mobile devices. The network transport media vary widely in bandwidth, delay, and error mechanisms, ranging from the high bandwidth and low error rate of ATM to the more scarce bandwidth and high bit error rate in wireless access.

Expected to operate in such a heterogeneous environment, applications, terminals and networks must be increasingly flexible [29]. They should be configurable in order to handle the differences in the coding formats, capabilities, and available bandwidth and processing resources of the components. The components should also dynamically adapt themselves to changing conditions. All these goals should be achieved with low cost and high subjective quality.

1.3.2 A Carefully-Crafted Architecture to Manage Complexity

It is important that the future networks, applications, and terminals be flexible in order to operate in a heterogeneous and dynamic environment. New objectives, which are mutually dependent and sometimes even mutually conflicting, need to be achieved. As pointed out by Yun and Messerschmitt [30], a possible set of objectives are:

- handling the heterogeneity in applications, terminals, and network transport;
- high subjective quality and low perceived delay;
- high traffic capacity;
- multicast, multi-source connections;
- mobility;
- privacy by end-to-end encryption.

To achieve these objectives requires a complex coordination between the applications, terminals, and networks. How should we organize this complicated coordination? This is a complexity management problem. The first step to manage complexity is to have a well-crafted architecture, which divides the system into interacting modules with well-defined interfaces that minimize and control the dependencies [31]. Before I describe a possible architecture, I will first define some terminology.

- *Session*: A session is an execution of an application. For example, a session of voice telephony is a phone call.
- *Signaling*: It is a common practice to divide networking functions into *data delivery* and *control*. Control functions include coordination and configuration. In telecommunications terminology, the exchange of information between applications, terminals, and networks for the purpose of realizing control functions is called *signaling* [32][33].

In Section 1.4, I first review a proposed data delivery approach. Then in Section 1.5, I describe the control of such data delivery.

1.4 Data Delivery

Data should be delivered via *substreams* with optional *quality of service* guarantees. I now explain these two notions.

1.4.1 Quality of Service

Audio-visual data, such as voice, still images, and video, are different from computer data in many aspects.

- Audio-visual data are sampled and quantized versions of analog signals. The digital representation of these analog signals may have different resolutions (e.g., image size, number of bits per pixel, etc.), which represent different temporal-spatial sampling rates and quantization steps.
- Audio-visual data can be mixed and superimposed. This is especially useful in conference applications where there are multiple sources. Mixing the traffic from the multiple sources can cut down the incoming bandwidth to a receiver. These operations do not make sense for general computer data.

When transported over the network, audio-visual data are also different from computer data.

- The transfer of computer data, such as computer programs, cannot tolerate any error, while audio-visual data can tolerate moderate errors (except in some safety-critical applications like biomedical imaging), which correspond to distortion in the equivalent analog representation.

- Audio and video are also called *continuous media* [34][35]. Continuous media are streams of temporal samples, and hence their live reconstruction at the receiver imposes hard real-time requirements on the throughput and latency of the network transport. On the other hand, for computer data and still images, low throughput or long delay in general does no harm other than annoying impatient users.

These points are summarized in Table 1.1.

Table 1.1. Quality of service requirements for different media.

Media type	Error	Delay
Computer data	none	tolerant
Continuous media (audio and video)	tolerant	strict if interactive
Still images	tolerant	tolerant

Audio-visual data have *quality of service* (QoS) objectives that need to be satisfied by the transport. There are many proposed ways to describe QoS [36][37], but in general, bit rate (throughput), latency, and reliability/error (loss and corruption) are the three dimensions of QoS. Each of these dimensions can be described by some parameters. For example, source bit rate can be characterized by simple numbers like peak and average rates, or a more dynamic temporal characterization like leaky buckets [38][39], which are used in policing functions. In latency, the parameters would be maximum delay, average delay, delay jitter (variance in delay), etc. In the error dimension, loss and corruption should be distinguished for audio-visual data, where corruption (inaccurate information) is often better than loss (no information). Corrupted audio-visual data can often be utilized and should not be simply discarded. Moreover, if the protocol uses retransmission for error control, a corrupted version can be displayed first to let the user have something to look at very quickly, and it is successively improved as retransmitted packets arrive. This way we

can achieve asymptotic reliability while reducing the perceived delay. This approach has been applied to still images and graphics in a wireless computing environment [40].

All the above QoS parameters are concerned with just one stream of data. Inter-stream QoS parameters can also be defined, for example, the synchronization of audio and video streams.

1.4.2 Substreams for Variable Quality of Service

I have explained why streams of different types of data should receive different QoS treatment from network transport. But even within a stream, some portions may still be more important than others. This notion of *variable QoS* is obvious for reliability/error. For example, in uncompressed, pulse code modulation (PCM) samples, the most significant bits of each sample are less tolerant of bit errors than the least significant bits. After compression, some bits may become almost intolerant of bit errors (e.g., the motion vectors in motion compensation). These differences in error susceptibility benefit from different qualities of service from the network transport. Error protection mechanism, such as redundancy coding and power control, can be applied using this information to allocate the resources optimally; we do not need to protect a bit more than is necessary.

In addition to variable loss and corruption, we can also exploit variable delay. For example, *Asynchronous video* does not use the traditional synchronous, frame-by-frame reconstruction. It allows components of a frame to have different delays. The visually important component would have a low delay, while the visually less important component would have a higher delay. By relaxing the worst-case delay and hence smoothing traffic, asynchronous video achieves a perceived delay less than the worst-case delay and a higher traffic capacity [41][42].

Thus a stream could be further subdivided into *substreams*. The data in each substream should be treated identically in terms of QoS. (Messerschmitt uses the term *substream* to

denote a subset of a stream that should receive the same QoS [43]. In Internet Protocol version 6 (IPv6), this notion is called a *flow* [44][45].) As a simple example, a stream of PCM samples can be divided into two substreams, one carrying the most significant bits and the other carrying the least significant bits. The transport should endeavor to deliver the most significant bit substream with a lower error rate.

Data transport should be performed as a set of end-to-end substreams. The difference between streams and substreams is that substreams coming from the same stream have additional information available on their joint behavior (e.g., joint rate) that the network transport might be able to exploit. Note that variable QoS control and the reconstruction of streams from substreams at the receiver potentially have high overhead.

The substream structure is motivated by the interaction between signal processing and networking for continuous media [43]. In the above I have given a motivation: variable QoS. A second motivation for substreams is to transport scalable, layered coding of audio [46] and video [47][48][49]. Each layer is transported as a substream. Using this approach, we can easily vary the quality of video, such as the resolution or rate, especially in a multicast context where the receivers have different capabilities (access bandwidth or display resolution). Transporting layered coding in substreams also helps the network to adapt to changing network traffic conditions dynamically by selectively throwing away packets in less important substreams (this can be used in both point-to-point and multicast connections).

1.4.3 Remarks: Quality of Service Guarantee vs. Best Effort

There are two schools of thought on real-time networked applications. One school proposes guaranteed QoS, usually achieved by reserving resources such as bandwidth, buffer space in routers and switches, processing, and display resolution [50]. This results in simpler applications but more complicated networks, because the networks must realize QoS

guarantees. The other school advocates best effort network transport and (bandwidth) adaptive applications. The applications observe the current network conditions and dynamically adapt themselves accordingly. Network transport would do their best effort at light load, and enforce fair sharing of resources at heavy load [51][52]. The philosophy of this school is that resources like bandwidth probably will not be scarce in the future, while the signaling complexity to achieve QoS guarantees may offset the benefits gained. Indeed, the processing overhead to achieve the QoS guarantees may be extremely high, as the scheduling problems involved are often intractable.

Despite its high complexity, a compelling reason for QoS control is interactive delay, which experiments show should be less than 400 ms in critical interactive applications. (Asynchronous video [41][42] and asymptotic reliability [40] are two approaches to reduce the perceived interactive delay.) Another compelling reason is the traffic capacity on wireless access links. The limited bandwidth on wireless links is a lasting bottleneck, which will not go away with improving technology.

In the next section, I will discuss the session control for the guaranteed QoS approach, which has a more complicated session establishment procedure than the best effort approach.

1.5 Session Control

I now discuss the control a session, i.e., the application-terminal-network coordination at session establishment and during a session.

1.5.1 Session Establishment

How should a session be established? An application needs to coordinate with the network and terminals to configure the service they provide. The network and terminals also need to coordinate among themselves. I will describe a general model that works for both tele-

communications [53][54][55][56][57] and computer networking [58][59][60]. If certain assumptions are made about the network or terminals in the model, simplifications ensue. They can explain the differences in the cultures of the telecommunications and computer networking communities.

In general, a session establishment would consist of the following mutually dependent steps. (The order of these steps is not definite. Also, some steps should ideally be performed jointly to reach a truly optimal solution.) These steps may take place as follows.

- *Dynamic deployment*: Usually applications are built into the terminals, for example, telephone sets and videophones. If an application is implemented in software, application functionality does not have to be built into the terminals. Application descriptions can be downloaded from some repository to programmable terminals at startup time. This optional *dynamic deployment* step is elaborated in Section 1.6.2.2.
- *Presenting a session model*: The application starts execution and sends a session setup request to the network. In the session setup request, the application presents to the network a session model, which is a collection of abstract connections. An abstract connection is a high-level logical model of the connection with only minimal information such as media type and terminal descriptions. For example, the video connection in a multipoint video conference would be described by its media type (video), network addresses of the participants, and their video coding formats and display capabilities. How this multipoint video connection can be realized is determined in the next step.
- *Realizing the abstract connections*: The terminals and networks jointly determine what resources are necessary to realize the abstract connections in the session. The resources include physical network connections (both point-to-point and multicast), so this involves solving the routing problem. The resources may also include processing,

such as conversion (to match the different access bandwidths and processing/display capabilities of the terminals) and composing (in multi-source connections) functions.

- *Negotiation, resource allocation, and admission control*: The application data will be delivered over the network connections as substreams with QoS requirements. The application, terminals, and networks negotiate the end-to-end QoS for each substream, subject to cost constraints. They may also negotiate where (in the terminal or network) the conversion and composing functions should be performed. A concatenated network needs to allocate the end-to-end QoS to the constituent links. If an agreement cannot be reached, the connection is not admitted.
- *Configuration*: Finally, each terminal and network link configures itself according to the negotiation agreement.

If conditions in the network or terminals change during the session, this process may be repeated dynamically to adapt to the changing conditions. In the following subsections, I elaborate on some of the steps.

1.5.1.1 User Involvement: Initiating and Responding Users

In immediate user-to-user applications, responding users are invited to join a session. So session establishment has an additional *alerting* step (for example, the phone rings) to notify the user of an incoming call.

1.5.1.2 Allocation of Quality of Service and Processing

In a concatenated network, we need to allocate the given end-to-end QoS objective onto the constituent subnetworks operated by multiple service providers. This is a *disaggregation* of the QoS model, as shown in Figure 1.4. We also need to perform the inverse operation, the *aggregation* of QoS models for the constituent subnetworks into a single end-to-

end QoS model. Therefore, the QoS models must be able to be aggregated and disaggregated, in addition to describing the needs of the applications.

Some processing functions, such as composing (e.g., conference bridges [61]) and conversion functions (e.g., transcoders or the *proxy* architecture of Fox and Brewer for mobile clients [62]) can be performed either by the terminals or by the network, or even by some third-party service providers. Where such processing should be allocated depends on the routing and the relative costs for these entities to perform the processing. Shih-Fu Chang has done some work on the allocation of video composing functions in his Ph.D. dissertation [63].

1.5.1.3 Negotiation of Quality of Service and Processing

In a heterogeneous network environment, an end-to-end connection consists of concatenated links with different QoS characteristics, possibly operated by different service providers. At session establishment, there must be a complex negotiation among endpoint terminals and the network, as well as among network entities, to arrive at a mutually acceptable set of QoS (delay, loss, corruption, rate) guarantees that also meet user cost objectives. This is a form of terminal-network signaling.

One possible model to carry out the QoS negotiation is to use *negotiating agents*. In the negotiation, a component, which can be an application, terminal, or network link, is

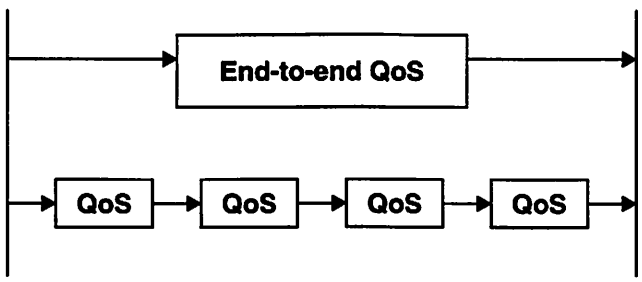


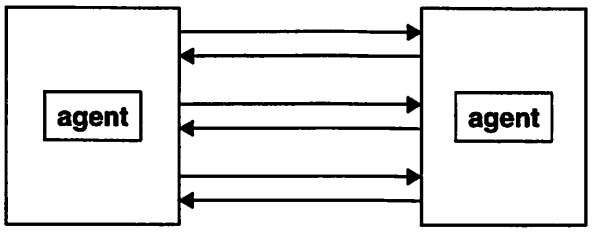
Figure 1.4 QoS disaggregation.

represented by an autonomous entity called an *agent* [64][65]. An agent encapsulates relevant information about a component, such as its cost and performance models (as functions of resources), resource coupling, current network conditions, and negotiation strategy. The negotiation agents jointly seek to optimize some metric, for example, to maximize performance subject to cost constraints, or to minimize cost subject to performance constraints. This optimization is logically distributed, represented by the negotiation strategy of each autonomous agent. In general it needs several rounds of information exchange because of the lack of a centralized entity, the inherent complexity of the optimization problem, and the "give and take" nature of negotiation. Although agent negotiation has a valid goal, its overhead must be minimal for it to be practical.

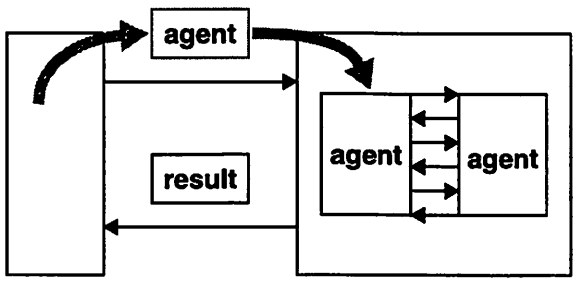
If the agents are stationary, they have to negotiate with each other across the network using message passing or remote procedure calls (RPC). With many rounds of information exchange, the end-to-end propagation delay and the signaling traffic overhead can become expensive, as shown in the top half of Figure 1.5. In particular, propagation delay is fundamentally constrained by the speed of light. It cannot be decreased by progresses in networking technology.

A viable approach to avoid the propagation delay and reduce signaling traffic overhead is to apply techniques from *transportable computation*, where *itinerant agents* can move around the network to accomplish their tasks [66][67]. A component would send an agent to the site of another component, as shown in the bottom half of Figure 1.5. Then the two agents' interaction becomes local and does not go through the network. Only when the negotiation is done is the result sent back across the network to the component. If the negotiation involves multiple parties, the agents would all physically move to a centralized place (the *negotiation engine*) and conduct the negotiation there locally, as shown in Figure 1.6.

Note that the itinerant agent approach does not change the functionality of the negotiation; the same functionality is realized whether the agents are stationary or itinerant. However, the itinerant agent approach offers two potential advantages. First, it is a new programming abstraction, a new way of thinking about network programming. In some cases the itinerant agent abstraction may be more natural than the message passing or RPC abstraction. Second, the itinerant agent approach may reduce the networking overhead in negotiation. The overall network latency is the sum of *propagation delay*, which is the geographical distance divided by the speed of light, and the *transmission delay*, which is the number of bits to transmit divided by the transmission rate. Since an agent contains complicated program code (the negotiation strategy, optimization algorithms, etc.) and data (the cost and performance models, traffic conditions, etc.), it takes longer to transmit an agent than simple messages. But the itinerant agent approach incurs propagation delay



Negotiation by message passing or remote procedure calls requires many signaling message exchanges, each adding a round-trip delay.



If an agent moves to the premises of the other entity and negotiates locally with the agent there, the interaction does not go through the network.

Figure 1.5 Message passing vs. itinerant agents in service negotiation.

less often, only when the agent and the result are transferred. As transmission rate increases, eventually the propagation delay will dominate the overall latency. Therefore, the itinerant agent approach has a potentially superior performance to message passing or RPC in situations with high transmission rate, long geographical distance, and extended rounds of interaction.

Because itinerant agents are executable programs coming from remote sources, security is an important issue. Programming languages with built-in security mechanisms, such as Telescript [68][69], Safe-Tcl [70][71][72], and Java [73][74][75][76], must be used to write the agent programs. More details on applying agents to QoS negotiation and telecommunications signaling can be found in Weiyi Li's Ph.D. dissertation [77].

1.5.2 During Session

When a session is in progress, two actions can be taken to respond to changing network conditions.

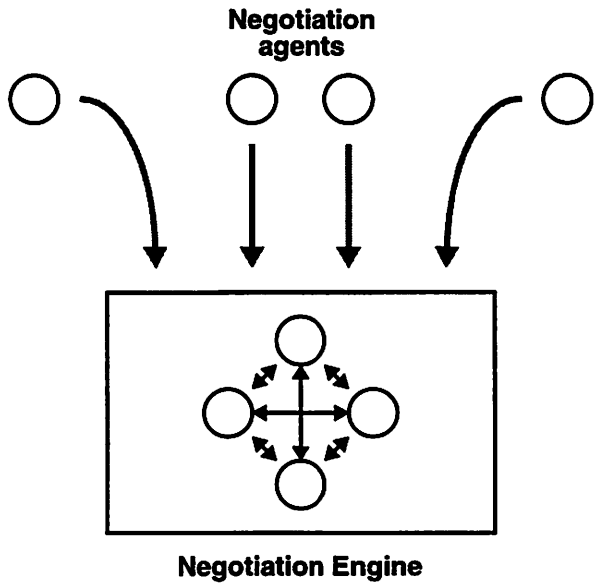


Figure 1.6 Negotiation agents all move to a centralized place, the negotiation engine, and perform the negotiation locally.

- *Renegotiation*: In principle, when the network promises a QoS guarantee to an application, the network should obey its contract. However, a session may span a long period of time. It is likely that in the middle of a session the network condition has changed significantly from the time the session was established. The network may want to request the application to lower its QoS objective so that more users can be admitted. A renegotiation of QoS would be required. Another reason for QoS renegotiation is that the application's QoS objective may change during a session. For example, some video compression algorithms, such as MPEG [78][79], produce variable bit rate output streams. The bandwidth requirement should be renegotiated during a session, tracking the output rate of the video coder, to make efficient utilization of the network bandwidth resource [80][81][82].
- *Dynamic adaptation*: If the network offers best effort service with no QoS guarantees, an application can be written to be aware of the current traffic condition and adapt itself dynamically in response to changing network conditions. For example, a voice telephony application on the Internet may switch to a more aggressive (and hence lower quality) compression algorithm when it detects that the network has become congested.

1.5.3 Remarks: Intelligent Network vs. Intelligent Terminal

Telecommunications has taken the network-centric, *intelligent network* view [83][84]. The terminals are minimal (e.g., the telephone sets), and the network performs some of the application functions, such as mixing (by conference bridges) and transcoding [61]. There are economic reasons behind this: network service providers would like to offer bundled services.

On the other hand, computer networking (the Internet) has taken the *intelligent terminal* view. The terminals are intelligent, such as personal computers and workstations, and network is minimal, acting as a conduit of bits. All the conversion and mixing functions, if deemed necessary, are performed by the terminals. The network provides best effort transport, which is considerably simpler, and the applications run adaptively on the terminals.

The computer networking community has begun to take some of the intelligent network flavor. For example, Fox and Brewer's *proxy* architecture [62] enables a mobile client terminal, which typically has low access bandwidth, low processing power, and low display resolution, to access a Web server by transparently converting the full-resolution data to a lower-quality version that can be handled by the mobile client. The proxy is a service provided by the network for the client terminals.

Even in the realm of functions that must be performed by network nodes, such as network control and management (routing, charging, billing, etc.), recently there have been proposals for exposing an open interface for end terminals to access and program these network functions. I give two representative examples here. The *xbind* project, leaded by Lazar at Columbia University, defines an object-oriented programming model and an application programming interface to ATM networks for the creation, deployment, and management of multimedia applications [85]. Tennenhouse and Wetherall of the Laboratory for Computer Science, MIT advocated an *active network* architecture that allows users to inject customized programs into network nodes to perform computation (e.g., compression) on the user data flowing through them or to tailor the node processing to the user application [86]. Although the two approaches use different technologies (*xbind* uses remote object method calls; the active network uses transportable computation), they realize the same goal — allowing users to program the network.

1.6 Scope of Dissertation

In this section, I identify the scope of my dissertation in the framework that I have laid out. Now we revisit the question asked at the beginning of this chapter: why are there so few user-to-user applications? To answer this question, I examine their *design* and *deployment*, two important phases in bringing new applications from ideas to reality. My dissertation addresses the issues in these two phases. The contributions are:

- a design methodology that is tailored to the design of networked applications, in particular the control functionality, which is a dominant component of network protocols;
- a dynamic deployment approach that avoids the standardization and network externality barriers faced by new user-to-user applications in their deployment.

Applications developed using the design methodology can be deployed to the hands of the users using the dynamic deployment approach at session establishment. The goal is that by reducing the design effort and avoiding the deployment obstacles, we can encourage a proliferation of compelling user-to-user applications. The issues and proposed approaches are described below.

1.6.1 Design

1.6.1.1 Design Challenges

I first argue that networked applications, in particular user-to-user applications, are inherently more difficult to design than stand-alone applications. Networking protocols usually have intricate control functionality, which is hard to specify and verify the correctness [87]. Therefore, the networking component in networked applications presents additional complexity to the design task. In addition, as I point out in Section 1.3.1, terminals and network transport are becoming increasingly heterogeneous and flexible. It is especially challenging to design networked applications that can deal with the heterogeneity in termi-

nals and networks while simultaneously exploiting their flexibility. The next generation of networked applications should be able to negotiate, coordinate, and reconfigure in order to interoperate, use resources efficiently, and achieve the best subjective quality. All this complicated interaction between applications, terminals, and networks adds complexity to the control functionality in applications as well as in terminals and networks. Therefore, the design of sophisticated control is key in the design process of networked applications. Control in user-to-user applications is especially hard to design. User-to-user applications implemented in the peer-to-peer architecture must use distributed control, which is usually symmetric interaction and complicated. They cannot use the centralized control implemented by the client-server architecture, where the interaction is often of the simpler request-response type.

1.6.1.2 Rapid Prototyping Methodology

To minimize the development effort, I propose a design methodology for user-to-user applications. The design methodology combines specification, simulation, verification, and synthesis all within an integrated environment. The design environment, based on Ptolemy [88], has special design support for the sophisticated distributed control in user-to-user applications. It also has design support for the other parts of user-to-user applications, for example signal processing.

To model and design a system composed of diverse subsystems, I advocate a heterogeneous approach. The approach superimposes *models of computation* such as dataflow and finite-state machines on standard programming language such as C++ [89], Tcl [90], and Java [73][74][75]. Systems are constructed by interconnecting modules. A model of computation coordinates the interaction between modules, which can be primitives in the model of computation or programs written in a standard programming language. Domain-specific models of computation can be mixed and combined to achieve generality.

In the Ptolemy design environment, a design is specified using a mixture of visual and textual descriptions. A design can be simulated to validate its functionality and critical properties can be checked at compile time. Finally, an implementation in software or a hardware description language (e.g., VHDL) can be automatically synthesized.

1.6.2 Deployment

1.6.2.1 Deployment Obstacles

Once new networked applications have been developed, they are still difficult to deploy quickly. Certain *closed* network architectures prevent easy introduction of new applications. If applications are implemented by the network, then to introduce a new application or modify an existing application requires upgrading some or even all of the network nodes. What is worse, this upgrade has to be done by the network operators. On the other hand, if an application is implemented in the endpoint terminals, then only those users who want the new application need to upgrade their terminals, and the users can upgrade independently of the network. In fact, this fragmentation of application space and autonomy of users are characteristic of the computer industry and are one main reason behind its booming growth.

For example, the voice telephony application uses the usual POTS ("plain old telephone service") telephone sets as terminal equipment. All the application functionality and features, such as digital voice coding, call forwarding, and call waiting, are implemented by the network. To deploy new functionality or features, it is necessary to modify the network nodes. This not only can be a large-scale, comprehensive operation but also requires the cooperation of the network operators. The users have no choice but to wait for the network operators' action.

An opposite example is the voiceband modem, whose functionality is implemented in the terminals (the modems). In recent years we have seen the data transmission rate of

modems rising from 300 bits per second to 28.8 kilobits per second. For the same reason, we have seen faster progress in the networked applications on the Internet. Most of the Internet applications are implemented in the terminals, which are computers or workstations with a lot of processing power. The network merely provides end-to-end transport. When new applications come out, it is only necessary to upgrade the endpoint computers, and only those who want the new applications need to upgrade.

The trend has clearly moved towards implementing application functionality in the endpoint terminals. But even with this architecture, the progress and innovation in networked applications has long suffered from two major obstacles to the rapid deployment of new networked applications. The first is the perceived need to standardize individual applications to achieve interoperability. Standardization is a lengthy process and adds a significant delay before the development can even start. The second is the *community of interest*, or what the economists call the *network externality* problem, where a sufficient number of specialized terminals must be deployed before a sufficient community of application participants exists [91][92][93]. The network externality problem is an especially serious obstacle to the deployment of user-to-user applications implemented in the peer-to-peer architecture. It has slowed down the progress and innovations in user-to-user applications significantly.

1.6.2.2 Dynamic Network Deployment

With the high-performance microprocessors available today, many multimedia networked applications can be realized in software on programmable terminals such as desktop computers and workstations. With high-speed networking, software-defined application functionality can be downloaded from a central repository to the terminals very quickly at session establishment. This is called the *dynamic network deployment* of applications.

Java applets downloaded from Web servers to Web browsers are an example of dynamic deployment [73][74][75].

Dynamic network deployment largely avoids the two obstacles to the deployment of networked applications – standardization at application level and network externality. Since the application software comes from the same source, interoperability is ensured without need of standardization. Moreover, with dynamic deployment, the network externality problem becomes less severe. The success of a new application would now depend on a critical mass of standardized programmable terminals, which is easier to achieve than a critical mass of specialized application terminals.

Dynamic deployment has the potential to save user-to-user applications from being "second-class applications" and encourage a proliferation of innovative user-to-user applications. Therefore, I propose to incorporate dynamic network deployment into the session establishment procedure (see Section 1.5.1).

1.6.3 Relationships Between Chapters

Although the design methodology and dynamic network deployment approach can be used independently, we gain the most leverage when they are integrated, as shown in Figure 1.7. A new application is first prototyped in the design environment, using intuitive high-level computational models such as dataflow, finite state machine (FSM) control, discrete event, and synchronous reactive systems. After the design is fully validated and tested in the design environment, application descriptions in a language suitable for remote execution such as Tcl or Java are automatically synthesized from the high-level specifications. The application descriptions are then stored in a repository and deployed dynamically to the programmable terminals at session establishment.

We have previously demonstrated this combined design and deployment process using Ptolemy as both the prototyping and run-time environment (the programmable terminals

were Unix workstations running Ptolemy), the Ptolemy interpreted language (based on Tcl/Tk) as the application description language, and a custom dynamic deployment protocol based on TCP [94]. In Ptolemy, we prototyped a software emulation of a telephone set,[1] using the synchronous dataflow model for real-time audio processing and finite state

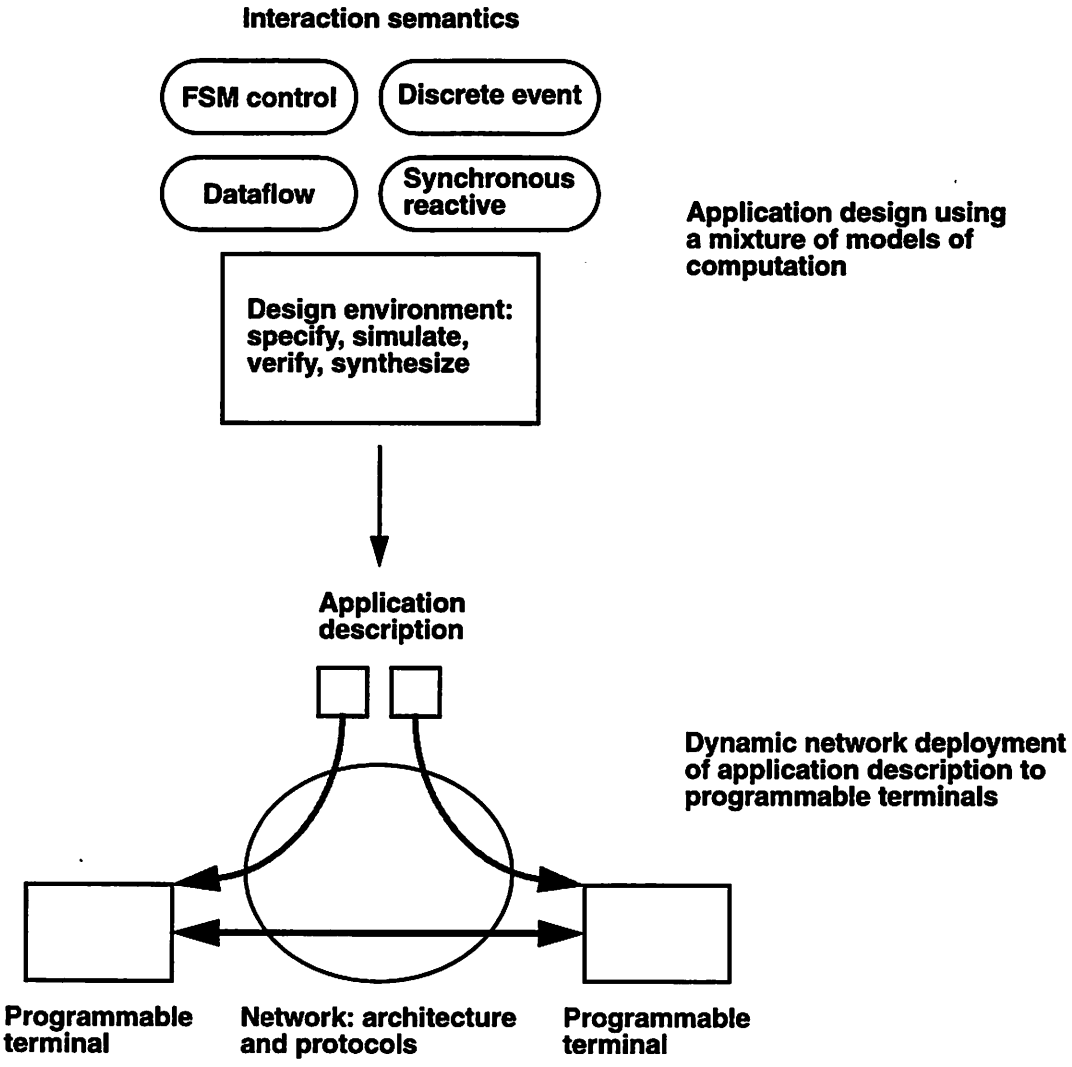


Figure 1.7 Relationship between the design environment and the dynamic network deployment.

1. A shared whiteboard and a shared text editor were also prototyped, though not in the Ptolemy environment.

machines for control. The application description (a Ptolemy interpreter file) is relatively small because it uses Ptolemy's software library modules installed on the terminals.

Since Java was announced in early 1995, we have adopted Java as the application description language for its better security mechanism. Our second implementation of the dynamic deployment approach uses a Java-enabled Web browser such as Netscape Navigator as the run-time environment and the HyperText Transfer Protocol (HTTP) as the dynamic deployment protocol [95]. We have demonstrated the dynamic deployment of a text-based communication application (similar to *talk* on Unix workstations) and a shared whiteboard on this standard, widely available framework.[1]

The remainder of the dissertation delves into the design and deployment issues in depth. Chapter 2 describes the rapid prototyping methodology. The general philosophy of the heterogeneous approach of Ptolemy and its application to heterogeneous design and simulation have been published [88][96][97]. Therefore, Chapter 2 focuses on a computational model based on hierarchical state machine semantics for the design of complicated control. Chapter 3 describes the implementation of this methodology in the Ptolemy software environment and some example applications. Chapter 4 describes the dynamic deployment approach. Chapter 5 gives concluding remarks and points out open issues.

1. Java code generation capability in Ptolemy is not finished yet at the time of this writing.

2

Rapid Prototyping Methodology

2.1 Introduction

In this chapter I propose a rapid prototyping methodology for networked applications (both user-to-user and user-to-information-server applications). The methodology is an application of the *heterogeneous design approach* of the Ptolemy design environment, developed at the University of California at Berkeley, to networked applications. The heterogeneous approach encourages designers to use domain-specific models of computation to design different parts of the system. In Ptolemy, subsystem descriptions in different models of computation are mixed by *hierarchical nesting*. A subsystem in one model can be hierarchically nested in a block in another model. Key to the heterogeneous approach is to define the semantics of the interaction between hierarchically nested models.

Models of computation describe the following aspects of system functionality:

- concurrency and communication (or *concurrency* in short): the interaction (via events) between concurrent components;
- control flow (or *control* in short): the sequencing of operations of an individual component.

I first review Ptolemy's interaction semantics for mixing concurrency models. Then I go on to the main theme of this chapter: mixing control models with concurrency models. Because networked multimedia applications typically have a layer of sophisticated control functionality (e.g., network protocols and user interface) overlaid on signal processing operations (e.g., compression, error protection, and encryption), I examine hierarchical state machine models for describing control, and study how to mix hierarchical state machine control with concurrency models such as dataflow that describe numeric computation.

In digital systems, control flow and numeric computation are usually structured into separate *control* (usually a finite state machine) and *datapath* modules. I propose an alternative scheme, called **charts* (pronounced *starcharts*), for mixing state machine control and numeric computation. In *charts, a subsystem in some concurrency model can be hierarchically nested within a control state, with the intended interpretation that the internal subsystem describes the active portion of the datapath when the control is in that state. At any time instant, the subsystem inside the current control state is active. The active subsystem is switched as the control makes a state transition. The *charts model is flexible. Various concurrency models can be mixed with hierarchical state machines to obtain essentially variants of Statecharts, the most popular hierarchical state machine model. I describe the syntactic features of *charts and define its semantics by giving an execution algorithm.

2.2 The Heterogeneous Design Approach

A networked application is a reactive system [98][99]. Reactive systems interact continuously with their environment at the speed of the environment. Reactive systems have real-time constraints, and are frequently concurrent systems.

The heterogeneous approach to system design seeks to combine a variety of design methodologies and implementation technologies. Design methodologies are encapsulated in models of computation, which define the semantics of the interaction between system components. Examples of models of computation include dataflow (the dataflow model, which originated from Dennis's seminal work [100], has many variants such as synchronous dataflow [101][102][103], Boolean dataflow [104][105], cyclo-static dataflow [106], and dataflow process networks [107][108]), discrete-event, synchronous reactive systems [109][110][111], and finite state machines [112]. Instead of using a monolithic model of computation to describe the entire system, the heterogeneous design approach uses multiple models of computation to describe different parts of the system. The heterogeneous approach is based on the belief that a collection of small, domain-specific models of computation, when seamlessly combined, is better than a big, universal model of computation for the specification of system functionality, synthesis of efficient implementation, and verification of critical properties. The Ptolemy design environment, developed at the University of California at Berkeley, supports the heterogeneous design methodology.

2.2.1 The Ptolemy Design Environment

Ptolemy is a system-level design environment for signal processing and reactive, real-time systems [88]. A system component is called a *block* in Ptolemy. (What Ptolemy calls a block is called a *process* in many concurrency models.) Blocks can be composed to form a subsystem, and they interact according to some model of computation. A block can be hierarchical, containing a subsystem of other blocks. Ptolemy overlays high-level, domain-specific models of computation (called the *coordination languages*) on top of general programming languages (called the *host languages*) such as C++ [89], C [113], Tcl [90], and Java [73][74][75]. An atomic block can be a primitive in a model of computation, or its function can be specified by a piece of code in a host language.

Ptolemy mixes models of computation by *hierarchical nesting*, as shown in Figure 2.1. A block in one model of computation may hierarchically contain a subsystem of blocks in another model of computation. Such a hierarchical block, called a *wormhole*, is treated as an atomic block by the model of computation on the outside.

2.2.2 Partitioning of Functionality

Networked multimedia applications frequently combine intensive numerical computations with sophisticated control. Coding and decoding the audio and video signals require sophisticated signal processing. But a sizable portion of the application functionality is in control operations such as initialization, configuration, dynamic reconfiguration, adaptation, network protocols, and user interaction. A layer of control must be overlaid on top of the signal processing.

In general, the functionality of an application can be partitioned into control (decision making and sequencing of operations), numeric computation, and data manipulation. Each of these aspects of application functionality has a distinct style, and there are domain-specific design methods that work especially well. For example, dataflow models are typically used for signal processing and numeric computation [101][102]. Hierarchical finite-state machines, such as the Statecharts model [114] and at least 20 variants [115][116] including Argos [117][118], are typically used for control. Data structures, such as linked

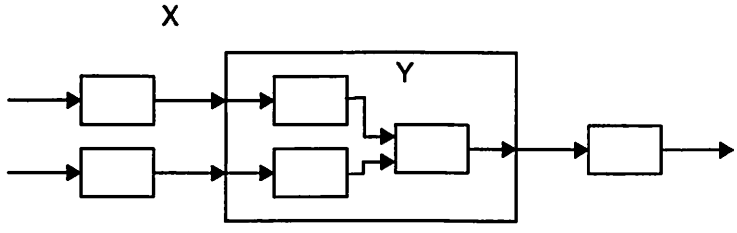


Figure 2.1 Mixing models of computations using hierarchy: shown here is a subsystem of model of computation Y embedded in model of computation X as a hierarchical block.

lists, stacks, and queues, are used for data manipulation. (Note that data structures are not models of computation.)

2.2.3 Domain of Applicability

Ptolemy overlays models of computation on programming languages. The concurrency (such as dataflow and synchronous reactive systems) and control (such as hierarchical state machines) models augment the sequential control flow constructs available in imperative languages. The concurrency models also impose structures on the interaction between concurrent components. The benefit of structured interaction between concurrent components is analogous to the benefit of the previous stages of *structured programming*: structured programs are easier to understand and maintain.[1] Moreover, if we use semantically small models of computation to describe the interaction between high-level components, the verification of some critical properties (at the level of the models of computation), such as determinacy and freedom of deadlocks, remains decidable and can be automatically performed. Lower level details, such as the manipulation of data structures, are done using the underlying programming languages.

2.3 Mixing Concurrency Models in Ptolemy

In Ptolemy, the execution of a block is divided into a sequence of discrete *firings*. A firing is a quantum of computation. Ptolemy further requires that the execution of a subsystem of concurrent blocks be divided into a totally ordered sequence of *rounds*. (A reason for this requirement is to mix concurrency models, which is elaborated in Section 2.3.1.) The concurrency model defines what a round of execution is. Within each round, individual blocks

1. The first stage of structured programming (Dijkstra) is concerned with sequential programming. Control flow structures such as *if-then-else*, *for* loops, and *while* loops are preferred over unstructured *goto* and conditional branch statements. The second stage of structured programming is concerned with mutual exclusion and synchronization in concurrent programming. Constructs such as conditional variables, critical regions, and monitors are preferred over the primitive semaphores.

are executed, one firing at a time, in an order determined by the semantics of the concurrency model. Firings within a round are in general only partially ordered. A firing of a hierarchical block is defined to be a round in the execution of its constituent blocks.

In some concurrency models, such as synchronous reactive languages and discrete-event models, the execution of a subsystem of blocks naturally has the structure required by Ptolemy: a totally ordered sequence of rounds. Other concurrency models, such as dataflow, may not impose sufficient ordering relations on the block firings to have the required structure, so additional order relations, not intrinsic in the model, must be imposed on the execution to turn it into a sequence of rounds. The additional order relations in effect restrict the set of legal executions originally allowed by the execution policy of the concurrency model.

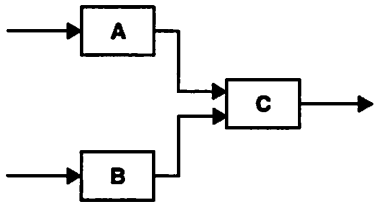
In dataflow, firings are only partially ordered by data dependency. A natural definition of a round in the execution of a dataflow graph would be a *complete cycle*. A complete cycle is a minimal finite execution that returns a dataflow graph to its original state, where the state of a dataflow graph is defined to be the number of tokens on each arc. An example is shown in Figure 2.2. For synchronous dataflow, a complete cycle, if it exists, can always be found [101][102]. For the more general Boolean or dynamic dataflow, determining whether a complete cycle exists is an undecidable problem [105], so there is no obvious definition for a round of execution. A programming environment may leave the definition of a round to the user.

2.3.1 Synchronized Interaction Semantics

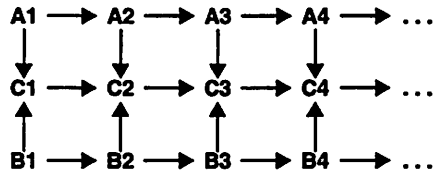
Ptolemy uses a *synchronized interaction semantics* to mix concurrency models [96]. Definition of a round of execution is crucial for Ptolemy to mix concurrency models this way. The algorithm is outlined below.

In Ptolemy, blocks communicate with each other by exchanging events. An *event* denotes the intuitive notion of an instantaneous happening. At any time instant, an event is either present or absent, and it may carry a value. A model of computation is *timed* if the events in the model carry real-valued time stamps. For example, the discrete-event model is timed. A model is *untimed* if its events do not have time stamps. For example, the synchronous dataflow model is untimed.

In order to mix timed models with untimed models, we need to impose time stamps on untimed models. In Ptolemy, each round is assigned a time stamp whether the model is timed or untimed. An untimed model ignores the assigned time stamps for its internal execution, but use them for interacting with a neighboring model, which is possibly timed. Time is stopped during a round. This means that the events within the same round, although possibly causally related, are considered simultaneous. Time is advanced from one round to another (see Figure 2.3).

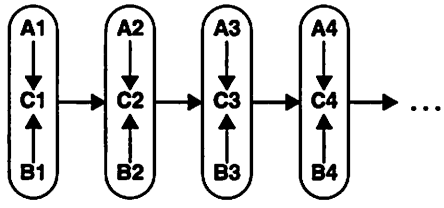


(a) An SDF subsystem where every actor consumes one token on each input and produces one token on each output.



(b) Partial order of firings imposed by the dataflow model of computation.

An arc from a group of firings (enclosed in a bubble) to another is a shorthand for saying that every firing in the "source" group precedes every firing in the "destination" group.



(c) Partial order of firings after grouping the firings into rounds.

Figure 2.2 Effect on partial orders of the quantum of computation.

A round begins with receiving external input events from the environment of the subsystem. Then the enabled blocks fire and generate intermediate events, which in turn trigger more block firings. When all the firings have finished, external output events are delivered to the environment. In the following, I briefly describe some issues in resolving the differences in models.

2.3.2 Generating Events

Events are usually generated by blocks in response to triggering input events. Then where do the first input events that get a round going come from? Such an initial input event may come from the external environment, or it may come from the simulator (in response to a future event notice), or it may be generated without any triggering, e.g., by a source block (a block with no input), a latch, or a Moore finite state machine[1].

The output events generated by a block become input events to other blocks. Models differ in when the newly-generated events become available for other blocks waiting on these events. There are three possibilities (the first two are synchronous models, the third simulates asynchronous models):

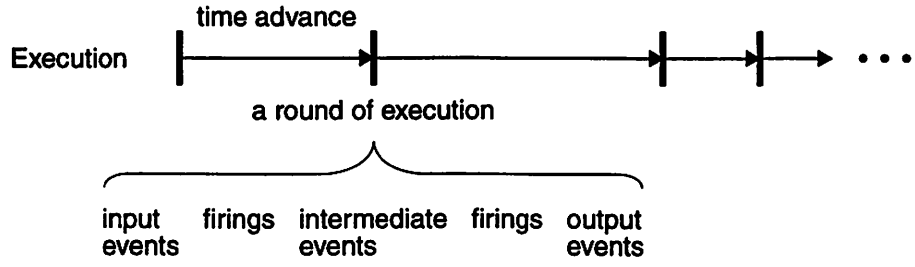


Figure 2.3 In Ptolemy, the execution of a subsystem of blocks is structured as a totally ordered sequence of rounds. Firings and events within a round are partially ordered.

1. The outputs of a latch or a Moore finite state machine depend only on its current state. Therefore, its outputs can be generated without waiting for input events.

- Generated events can be used immediately within the current round. This amounts to a "zero-delay" block. The initial events trigger intermediate events, which trigger other intermediate events and eventually output events. To ensure that there exists at least one behavior, we may either require that there be no causality loops (i.e., the firings within a round are partially ordered, and hence there are no deadlocks) or use a fix-point semantics to give meaning to cyclic dependency (e.g., the synchronous reactive programming languages [110]). Furthermore, it is desirable that there is at most one behavior, i.e., there is no nondeterminism in the computation results due to the execution schedule actually used. If a block in a model must wait until an event becomes present, then it can be shown that there is no nondeterminism due to different execution schedules used (e.g., in dataflow and Kahn process networks) [119][120][121]. If a block in a model may react to the presence or absence of an event differently, then there is potential nondeterminism due to the actual execution order used in a round (e.g., in some discrete event models). To eliminate such nondeterminism, blocks in Ptolemy's discrete event model provide hints to the simulator so that it can constrain the execution schedule and ensure that the emitter of an event is fired before the consumer of the event. Some models have a notion of consuming an event. An unconsumed event may persist to the next round (e.g., via the FIFO arcs in dataflow). If not, a latch block must be used to hold the event over to the next round.
- Generated events can only be used in the next round, e.g., systolic arrays and the δ-cycles in VHDL simulators.
- *Event notices*, rather than events, are generated for a future time, e.g., in discrete event simulators. In Ptolemy *simulation*, a block when fired must return immediately. This implementation restriction is not a problem for synchronous models in the previous

two cases. To simulate asynchronous models, where enabled blocks may take an arbitrary delay to react and generate outputs, Ptolemy uses the *event-scheduling* paradigm [122][123][124]. In this paradigm, an enabled block (e.g., triggered by an event) fires (and returns) immediately and generates *future event notices* with the times the events are supposed to occur. An event scheduler maintains the event notice list and generates the future events on behalf of the blocks at the right times in chronological order.

Remark: A block in such a model may be unable to generate future event notices, e.g., it is a wormhole containing another concurrency model. If we want to model a wormhole block with a nonzero delay, the trick is to have the wormhole block generate a real output event (which would have the same time stamp as the triggering input event) and feed the output event to a delay block, which generates a future event notice for the wormhole block.

2.3.3 Time Advance and Round Initiation

After a round of execution has finished, the next step in simulation is to advance the time and initiate a new round. A round may be initiated by a triggering event, e.g., in discrete event models. In this case, the time stamp of the round is set to be equal to the time stamp of the triggering event. A round may also be initiated automatically without trigger. For example, a synchronous dataflow system with its own source blocks and no external inputs is always enabled for firing. In this case, the user must specify how the time is advanced between rounds because timing information is irrelevant to and ignored by the synchronous dataflow model. In Ptolemy, a synchronous dataflow system with no external inputs advances its time between rounds by a fixed increment (called the *schedule period* in Ptolemy) provided by the user.

Sometimes a subsystem that sits within another model (i.e., it is inside a wormhole block) can initiate a round of execution without any trigger from the outside model. It either does not need any trigger at all (e.g., a synchronous dataflow subsystem with no external inputs, as mentioned in the paragraph above) or has its own internal triggering (e.g., a discrete event subsystem with internal future event notices). In such cases, the internal subsystem of a wormhole block informs the outside model of the future time at which it should be invoked. Where it does not violate the execution policy of the outside model, the outside model fires the wormhole block at the requested time. (Note that this mechanism for a wormhole block to schedule itself for future execution is not yet fully implemented in Ptolemy.)

2.4 Formal Models for Specifying Control

Control is studied by many fields. (A common structure of a control system is shown in Figure 2.4.) However, the word *control* is used in many senses, and the styles of control studied by different communities can be quite different. Therefore the models or languages suitable for describing each style of control functionality may be different.

In programming languages and digital electronic systems, control (short for *control flow*) means the sequence in which operations are performed. Control (also called *control-*

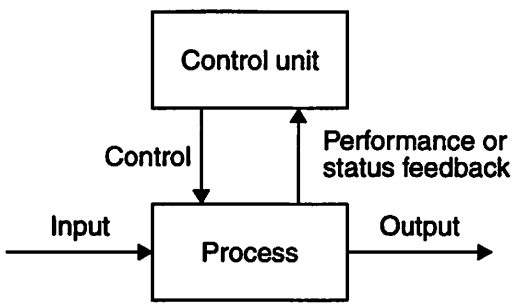


Figure 2.4 Common structure of a control system.

ler, control unit, etc.) also refers to the system component in charge of sequencing the operations performed by other system components. The states of such a control module typically correspond to the steps to perform a task or the operation modes. Therefore, the control module's states usually have no algebraic structure and must be enumerated. Similarly, the control signals that the control module sends to the controlled process typically have no structure (usually denote various control actions such as *select, switch, enable*, etc.) and must be enumerated. The state transition relation is best represented as a *state machine.*

This style of control is distinct from the control most often studied by the control theory community, where the state variables are typically numbers and the control functionality seeks to ensure that the numeric state variables stay close to desired values or evolve close to desired trajectories by giving numeric control parameters to the controlled processes. Because state variables are numbers, state transitions are best represented by differential (in continuous-time systems) or difference (in discrete-time systems) equations. Note that this style of control actually performs intensive numeric computation.

Here I focus on the former style of control, i.e., sequencing of operations. The main objective of this chapter is to develop specialized models of computation (both their formal semantics and visual syntax) for describing complex control functionality and mix them with concurrency models such as dataflow. In the rest of this section, I first review the most common models for specifying control flow in reactive systems.

2.4.1 Finite State Machines

A number of modern programming methodologies aimed at the domain of control-dominated applications are based on finite-state machines (FSMs) [112]. FSMs are based on the intuitive notions of *states* and *events*, and have a well-developed mathematical theory. FSMs can be represented visually by state transition diagrams. A generic state transition

diagram consists of nodes that stand for states and arcs that stand for transitions. There are extended versions of state transition diagrams that have state nodes but replace the outgoing transition arcs from each state by an acyclic decision diagram (flowchart), for example, the ASM (Algorithmic State Machine) chart in logic design [125], the OC (object code) automata format generated by some synchronous language compilers, and the international telecommunication standard SDL (Specification and Description Language) [126][127].

2.4.2 Hierarchical Finite State Machines

Simple FSMs have a major weakness: nontrivial systems have a very large number of states. Modern solutions add *hierarchy* and *concurrency* to the basic FSM model. Hierarchical state machines, such as Statecharts and its variants, are representative [114][115]. In a Statechart, a state can be either *atomic*, as in a basic FSM, or *hierarchical*. There are two kinds of hierarchical states: *sequential* and *concurrent*. A sequential hierarchical state contains another (sequential) FSM. A concurrent hierarchical state contains multiple FSMs operating simultaneously and communicating by event broadcast.

Statecharts retain the inherent concurrency in the control functionality to avoid the state explosion in forming the Cartesian product of the state spaces of the concurrent machines. Hierarchy in Statecharts organizes the state space and amounts to *behavior refinement and abstraction*. The behavior during a hierarchical state can be refined into the FSMs nested inside, and the hierarchical state represents an abstraction of the more complicated behavior. Hierarchy allows us to look at the behavior at various levels of abstraction and is a tool for containing the complexity of the control functionality.

Hierarchy is also handy for describing interrupt-like behavior. If an outgoing transition arc from a hierarchical state is triggered, the hierarchical state is exited no matter what state the internal FSM is in. Without hierarchy, we would need to replicate the outgoing

transition arc for every state of the internal FSM. The use of hierarchy reduces the number of transition arcs.

2.5 Mixing Control and Dataflow

Control and data computation are two complementary aspects of system functionality. Dataflow models excel at describing numeric computation. A dataflow system is a graph, where the nodes, called *actors*, represent computation, and the arcs are first-in-first-out channels that carry data tokens flowing between the actors. The order of executing the actors is solely determined by data dependencies among the actors. An actor can be either atomic or hierarchical (containing another dataflow graph). A subset of dataflow called *synchronous dataflow* (SDF) requires that an actor consume fixed numbers of tokens on input arcs and produce fixed numbers of tokens on output arcs in each firing [101][102]. With this constraint imposed, the order of executing the actors can be determined at compile time. SDF is very suitable for describing multirate signal processing algorithms.

Mixing control and dataflow is an important issue in system-level design. In this section, I examine the various styles in which control flow and dataflow are combined in models of computation and specification languages.

2.5.1 Control/Data Separation Style

The traditional approach to mixing control and data is the control/data separation style. This style reflects the common structure of a control system shown in Figure 2.4. In this approach, there are a control entity and a separate data computation subsystem. Two examples are given below.

1. *Digital circuits.* A digital circuit is often divided into the control and the datapath [125][128]. The control sends control signals to the datapath and receives status signals from the datapath. The control is usually implemented as a finite-state machine, although

other architectures such as microcoded controllers are also possible. The datapath consists of functional units (e.g., adders, multipliers, and arithmetic logic units), registers, and interconnections (e.g., multiplexers and buses). The control uses the control signals to select the function of a multi-function unit and change datapath configurations (interconnects), and changes its state in response to external events or internal status signals. Such control/datapath systems can be built hierarchically, that is, some of the datapath units may in turn be composed of control and datapath.

2. *STATEMATE.* The STATEMATE design environment takes a similar control/data separation approach [129]. A system is built hierarchically. At each level of the hierarchy, there are some interconnected *activities* (processes) specified using an *activity chart.* One of the activities is special: the *control activity.* For convenience I call the other activities *data activities*, although this is not standard STATEMATE terminology. The control activity is internally specified using a *Statechart* [114]. In addition to sending control signals to data activities, the control activity in STATEMATE is equipped with some coarse-grained, asynchronous control primitives — it can start, stop, suspend, and resume a data activity. These activity control primitives are similar to those provided by an operating system for controlling software processes.

2.5.2 Control/Data Nesting Style

Another style of mixing control and data is hierarchical nesting. A graph that describes control flow, such as a flowchart or state transition diagram, can be nested inside a node in a dataflow graph, and conversely, a dataflow graph can be nested within a node in a control flow graph.

The control/data flow graph (CDFG) (e.g., the one described by Gajski *et al.* [130][131]) used in high-level synthesis tools combines flowcharts with dataflow graphs. It is the intermediate format obtained by analyzing a program written in a high-level spec-

ification language. The control flow and data dependency (from dataflow analysis) are used to schedule the operations for execution on the allocated datapath elements in the control steps [132]. CDFG is also used by optimizing compilers for imperative programming languages for similar purposes.

My preferred style of mixing control and dataflow consists of hierarchical nesting of dataflow graphs with FSMs, as shown in Figure 2.5 [97][133]. The depth and ordering of the nesting is arbitrary. In that figure, I have schematically illustrated dataflow semantics with rectangular boxes and FSM semantics with round nodes. Any dataflow actor can have its functionality specified by an FSM. The FSM in turn can have actions associated with either states or transitions, and these actions can be specified by a dataflow graph. Either model can form the top level of a design.

What are the motivations for combining control and dataflow this way? And how do we interpret such a mixture? The interpretation is *mode switching*. A system may have several *operation modes*, in each mode performing a different input-output transformation. Modes naturally correspond to the states of the control, and the input-output transforma-

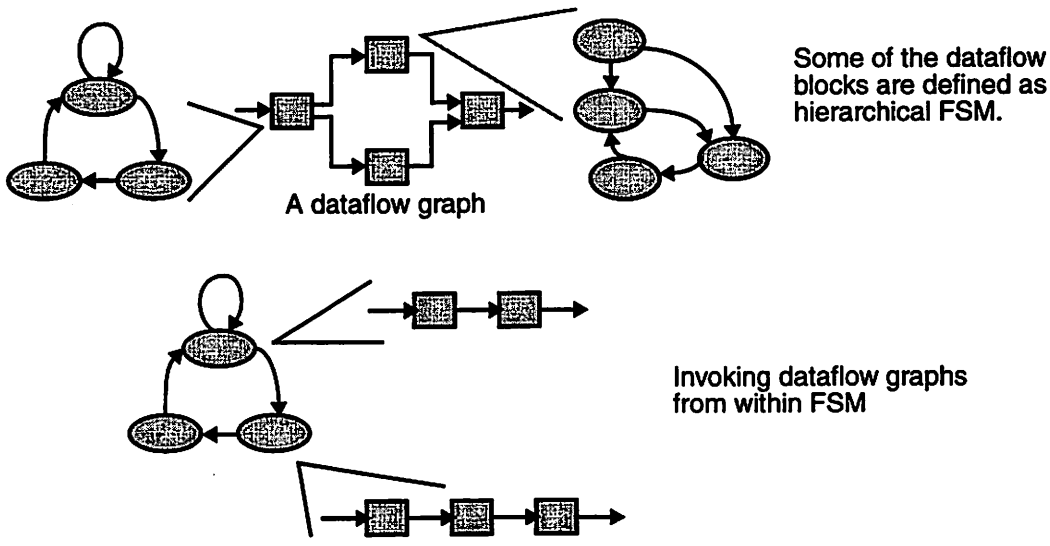


Figure 2.5 Hierarchical nesting of FSM controllers with dataflow graphs.

tions in the various modes can usually be described by dataflow graphs. To highlight the association between an operation mode and its data transformation, it is natural to embed the dataflow graphs into their respective control states, as shown in the bottom half of Figure 2.5. A dataflow graph is active when the control is in the state it is embedded in. Then state transition amounts to switching the operation modes and the input-output transformations of a system.

Pankert *et al.* of Aachen University of Technology, Germany independently proposed the same scheme [134][135][136]. A related scheme was proposed by Jourdan *et al.* of IMAG, France [137], where a textual dataflow synchronous language Lustre [138] and a visual hierarchical FSM language Argos [117][118] (a synchronous variant of Statecharts) are nested.

Side Remark: The control theory community uses *hybrid automata* to describe *hybrid systems* [139][140]. A hybrid system has a discrete-event controller and a continuous-time controlled process. The discrete-event control is specified as a FSM. The real-valued state variables of the continuous-time process evolve continuously according to a different set of differential equations while the discrete-event control is in a different state. Each set of differential equations is nested in the corresponding control state. Note that the differential equations inside different states govern the evolution of the same set of state variables. When a hybrid automaton makes a state transition, the values of the state variables can be carried over (by default) or changed abruptly. The scheme of Pankert *et al.* achieves a similar effect by allowing the dataflow graphs inside different states to pass data with *mailboxes* and by allowing dataflow graphs to overlap (thus the actors in the overlapping portion of the dataflow graphs are carried over when a state transition is made).

2.5.3 Dynamic or Boolean Dataflow

The only control flow in a dataflow model is data dependency. An actor cannot fire until all the data tokens it is waiting on have arrived. Dynamic dataflow (DDF) and Boolean dataflow (BDF) allow the number of tokens consumed or produced by an actor to depend on data values [104][105]. Using value-dependent token consumption/production, one can simulate common control flow constructs such as *if-then-else* (with a matched pair of *switch* and *select* actors) and *while* loops with DDF or BDF. However, expressing control flow this way is awkward and can easily turn into a brain-teaser once the control flow gets a bit more complicated than the simple *if-then-else*. Another problem with BDF or DDF is that it is undecidable to determine whether a complete cycle exists, so there is no obvious definition for a round of execution (see Section 2.3).

2.6 The *charts Model: Hierarchical Nesting of Control and Concurrency

In this section, I propose a flexible scheme, called **charts* (pronounced *starcharts*), for mixing control and concurrency models. The *charts model hierarchically nests state machine control and concurrency models. An FSM can be nested within a block in a concurrency model, with the meaning that the FSM describes the behavior of the block. Conversely, a subsystem of some concurrency model can be nested within a state of an FSM, with the meaning that the subsystem is active if and only if the FSM is in that state.

2.6.1 Mode Switching

There are two motivations for *charts. The first motivation is a generalization of the *mode switching* interpretation, shown in Figure 2.5, that I discussed in Section 2.5.2. The *charts scheme generalizes that model by allowing concurrency models other than dataflow to be nested with state machine control.

2.6.2 Generalizing Statecharts

The second motivation is that *charts are actually generalization of Statecharts. A problem with Statecharts is that it is monolithic — the FSM semantics and the concurrency models for the parallel component FSMs are built-in and fixed. People have their favorite FSM semantics and concurrency models, thus giving rise to at least 21 variants of Statecharts [115]. I will show in three steps that *charts generalize Statecharts and subsume all variants. First, I make Statecharts modular by removing noncompositional constructs. After this preparation step, FSM, hierarchy, and concurrency become orthogonal semantic properties and can be freely mixed and matched. (This is another example of the concept of *modular semantics* in Ptolemy.) Then I generalize Statecharts by allowing the concurrency model to be replaced and not requiring all concurrent components to be FSMs. By substituting various concurrency models, we can obtain models that are essentially variants of Statecharts, hence the name *charts.[1] These three steps are detailed below.

2.6.2.1 Modular Semantics of Statecharts

Some Statecharts constructs make the semantics of Statecharts noncompositional. For example, Statecharts allow transition arcs to cross hierarchy levels. Inter-level transition arcs can be removed from the Statechart model without losing expressiveness, i.e., they are just "syntactic sugar." Outgoing inter-level transitions can be easily simulated by *self-termination*, as shown in Figure 2.6 [110]. Incoming inter-level transitions are more cumbersome, yet possible, to simulate. A procedure that translates a hierarchical FSM with inter-level entry transition arcs to one without is shown in Figure 2.7. Inter-level entry transition arcs are like the *goto* statements in structured programming languages: they may

1. In the Unix operating system, * is the wildcard character, which stands for an arbitrary number of arbitrary characters. Many variants of Statecharts have "charts" in their names, e.g., SpecCharts [141][142][143].

be useful, but more often obscure the control flow, and can usually be avoided by a well-structured program.

Another noncompositional Statechart construct is *state reference*. A component state machine in Statecharts can test whether a concurrent machine is in a particular state by naming that state. I replace state references by what I call *Moore outputs*[1], which are outputs that do not depend on the inputs and depend only on the current state.

With inter-level arcs and state references removed, Statecharts become modular. The modular semantics of Statecharts consists of the following three cleanly separated components:

- FSM: There are different kinds of FSMs, e.g., Moore machines, Mealy machines, and ASM.
- Hierarchy: This semantic property is concerned with the interaction between an FSM and the internal subsystems nested within its states. In general, this means *selective activation* of the nested internal subsystems. But there are different nuances, e.g., strong or weak preemption, immediate or next-round start, etc. (Jourdan and Maraninchi's paper gave an example of using immediate start of internal subsystems in Argos [144].)

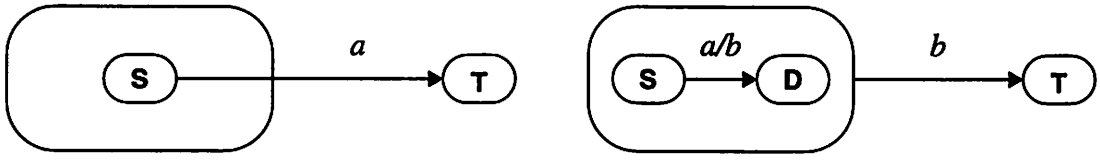
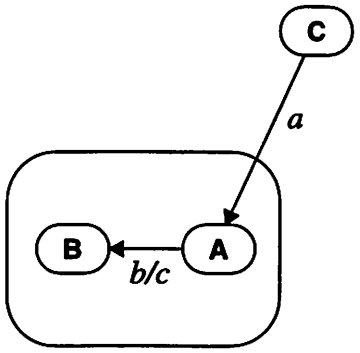


Figure 2.6 Self termination simulates an outgoing inter-level transition (exit from an arbitrary state).

1. The term *Moore output* is suggested by a Moore FSM, whose output is a function of the current state only.

- Concurrency: This semantic property is concerned with the interaction between the concurrent component FSMs.



Original FSM

A new initial state, Init, is added to the original FSM. Let A be a state in the original FSM with a cross-level entry arc. The cross-level entry arc now ends on the hierarchical state, and emits a new event *enter_A'*. For each arc leaving state A, such as the one going into state B, there is an arc from Init to the same destination state, with a new *in_A'* conjunctive term in the arc's input condition.

A new Moore machine is put in parallel with the original FSM. Let A be a state in the original FSM with a cross-level entry arc. A state A' is created in this new Moore machine, which emits event *in_A'* when it is in state A'. There is an arc triggered by *enter_A'* coming from every other state. I.e., no matter what the present state is, if *enter_A'* occurs, the next state is state A'.

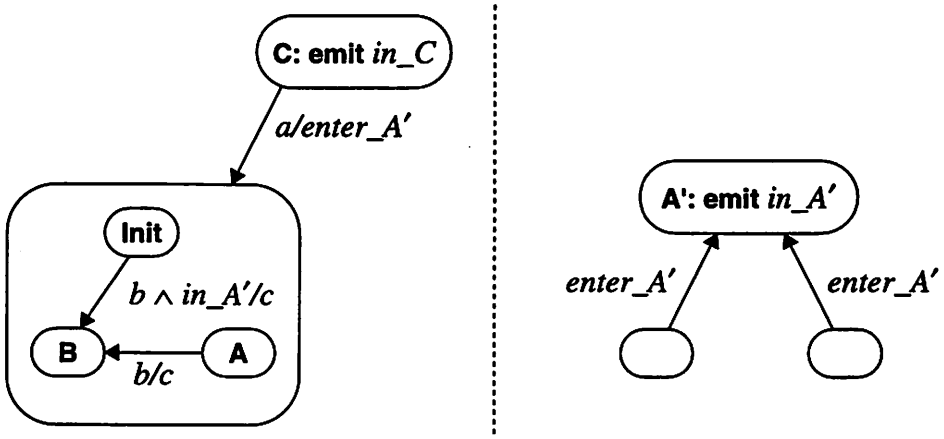


Figure 2.7 Translating a hierarchical FSM with inter-level entry arcs to one without.

2.6.2.2 Concurrency Models Can Be Replaced

Statecharts hierarchically nest FSMs with a built-in concurrency model, which varies among the Statechart variants. The concurrent hierarchical state in Statecharts is actually a syntactic shorthand for an interconnection of state machines operating concurrently, as shown in Figure 2.8. We can replace the built-in concurrency model by other concurrency models to obtain essentially variants of Statecharts. For example, if we use the synchronous/reactive communication (SR) model, the concurrency and communication semantics of the synchronous programming languages [109][110][111], the resulting hierarchical FSM model is essentially Argos[1]. We may also use the dataflow models as the concurrency and communication mechanism for the interconnected state machines, as shown in the top half of Figure 2.5.

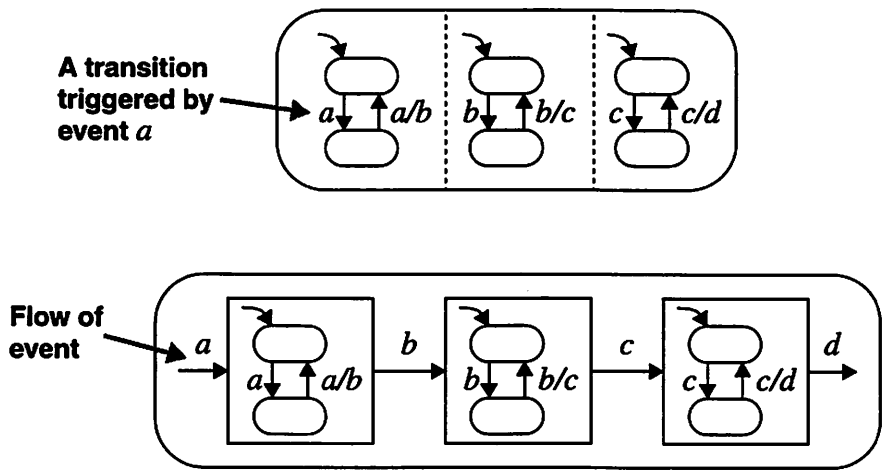


Figure 2.8 The concurrent states in Statecharts is a shorthand for an interconnection of finite state machines.

1. Mixing hierarchical FSMs with the SR model requires *nonstrict execution* of FSMs, which is not addressed by this dissertation (the reasons are given in Section 2.8.3, Item 2 on page 59).

2.6.2.3 Concurrent Components Do Not Have to Be FSMs

In fact, we can go one step further. The concurrent components inside a hierarchical state do not all have to be state machines. For example, they may be a network of dataflow processes. Sometimes a state evolution may be more naturally described by a dataflow process network than by a finite state machine. An example is shown in Figure 2.9. The state machine in this example (Figure 2.9(a)) may start to count until there have been five occurrences of event *e* at some point of its behavior. Then we may have a state named *count_5_e* that is exited when event *e* has occurred five times. The details of state

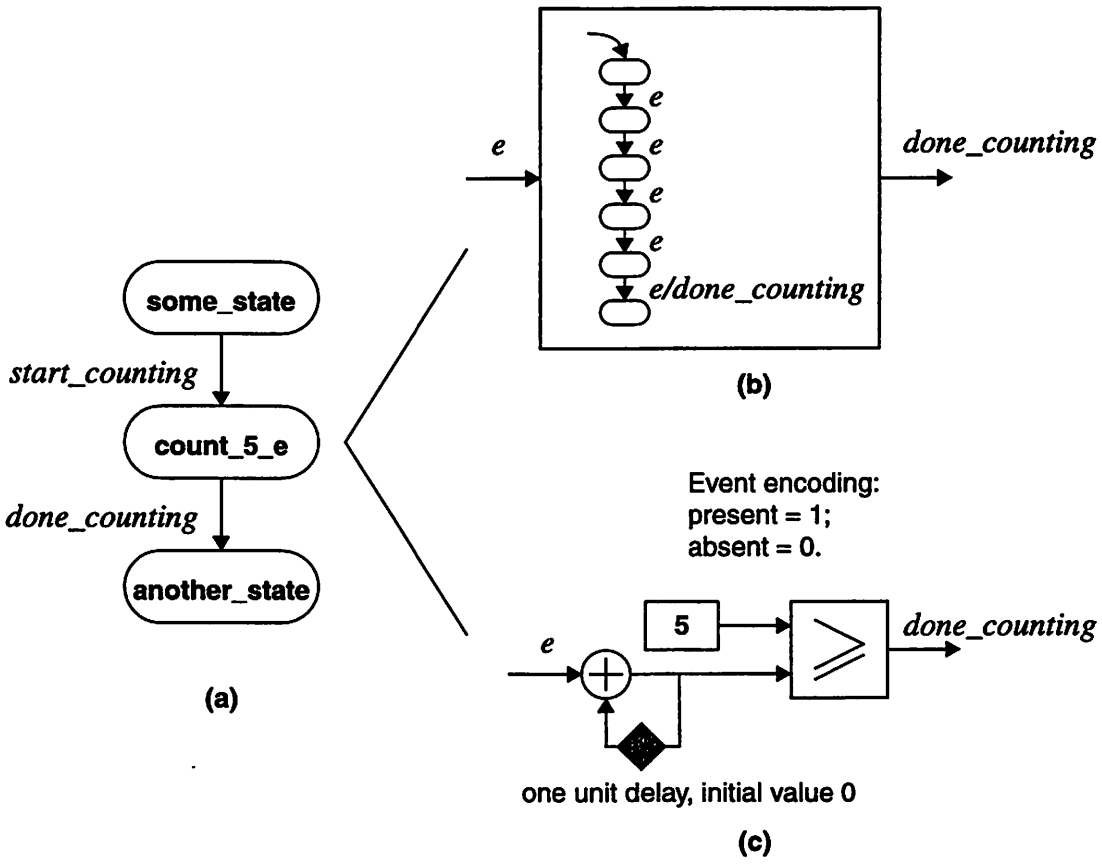


Figure 2.9 The behavior of a module (a) when it is in a hierarchical state can be refined by either a FSM (b) or a dataflow process network (c).

count_5_e can be described equally well by an internal FSM (Figure 2.9(b)) or an internal dataflow graph (Figure 2.9(c)).

In conclusion, I have shown that we can generalize the Statecharts model in two ways. First, we can substitute a different concurrency and communication semantics for the concurrent machines. Second, the concurrent machines do not need to be specified as FSMs. In particular, the concurrent machines can be a network of dataflow processes. As a result, *charts' style of freely nesting FSM with concurrency models is more flexible than Statecharts.

2.7 A Hierarchical Finite State Machine Model

In this section, I propose a hierarchical FSM model. From the conclusion of the previous section, only its FSM and hierarchy semantics need to be defined; its concurrency semantics is provided by mixing it with another model such as SDF or SR. I define its operational semantics by giving a simulation algorithm.

2.7.1 Syntax

The design of my hierarchical state machine model is mainly inspired by Argos. Its main syntactic constructs are outlined below (see Figure 2.10). We assume that the blocks in a concurrency model communicate by events, i.e., not by shared variables.

1. The behavior of a block can be specified as a hierarchical FSM, called the *master control FSM*, depicted visually as a state transition diagram. One of the states is designated as the initial state, denoted by an entry arrow with no source. A state can be hierarchical, which contains another block, called a *slave process*. The slave process inside the current state, called the *current slave process*, is active and runs concurrently with the master control FSM. A slave process may be another hierarchical state machine, thus achieving the same effect as a sequential hierarchical state in State-

charts. A slave process may also be a hierarchical block containing a subsystem of some concurrency model, thus achieving the same effect as a concurrent hierarchical state in Statecharts.

2. The master control FSM has external inputs, internal status inputs (coming from slave processes), internal control outputs (going to slave processes), and external output events. It may have private variables (not shared with other concurrent blocks or the slave processes), each with an initial value specified. Note that the private variables are also components of the state space of the master control FSM, in addition to the state component represented by the state boxes in the state transition diagram.

3. The inputs to an internal slave process are a subset of the external inputs and internal control outputs of the master control FSM. The outputs of an internal slave process are a subset of the internal status inputs and external outputs of the master control FSM. In

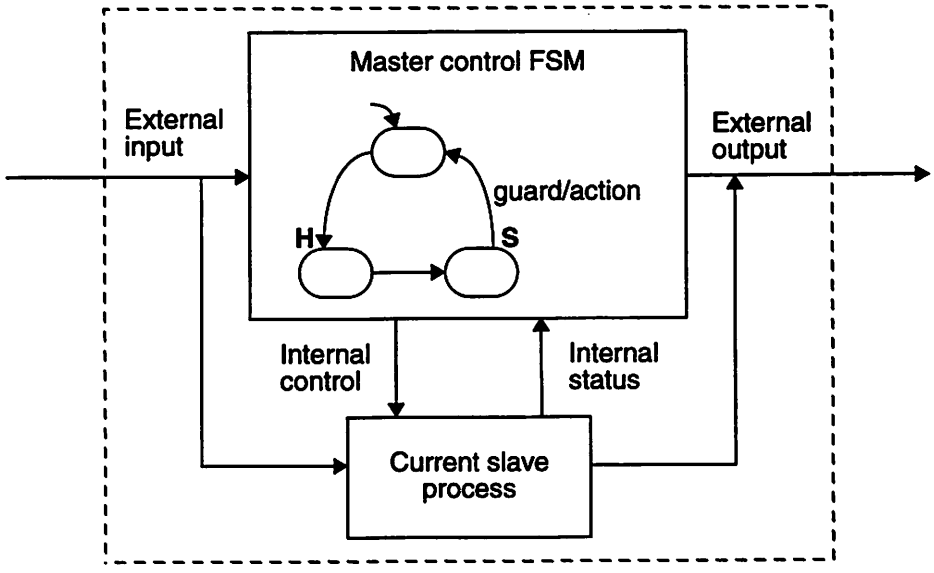


Figure 2.10 Main syntactic and semantic features of the proposed hierarchical FSM model. For the convenience of explanation, the current slave process is extracted from the current state and shown below the master control FSM.

each firing, an external output event may be emitted by either the master control FSM or the current slave process, but not by both (this is the *one-emitter rule*).

4. Output events may be emitted in Moore machine style (the outputs of a Moore FSM depend only on the current state). *Moore output* events (and for valued events, their values) are specified with a state. In each firing, Moore outputs of the current state can be emitted right away without knowledge of the external inputs. Note that the internal control outputs must be Moore outputs (to break cyclic dependency between the master control FSM and current slave process).

5. A transition arc is labeled with a *guard* and an *action*, separated by a slash /. The guard component of a label is a predicate (logical expression) in the presence or absence of events and the values of events and private variables. We allow the usual arithmetic, logical, and comparison operators on numeric and boolean types available in most programming languages. The action component (a la Mealy machine) is a piece of code that computes the values of the output events to be emitted and the next values for the private variables as a function of the input events and the current values of private variables. The action code can be in an imperative language or a graphical language such as a dataflow graph (see Figure 2.11). Note that a dataflow graph, when viewed as a dataflow process network, can also be embedded within a state as a slave process (see Figure 2.9). Either guard or action may be empty and omitted (an empty guard is equivalent to the Boolean constant `TRUE`).

6. Only explicit nondeterminism, such as that shown in Figure 2.12, is allowed. The compiler can check for such nondeterminism. No nondeterminism is introduced by our hierarchy operator. Note that the parallel composition operator in some asynchronous concurrency models such as I/O Automata [145][146] may introduce nondeterminism.

7. Hierarchical state entry: A transition arc ending on a hierarchical state can be either an *initial entry* or a *history entry.* An initial entry arc enters the initial state of the internal slave process. A history entry arc, labeled with an H at the arrowhead, enters the previous state of the slave process, i.e., the slave process picks up where it left off last time. Initial entry corresponds to a *start* call on a thread or software process. History entry corresponds to the *resume* call on a thread or software process after it was previously suspended.

8. Hierarchical state exit: A transition arc leaving a hierarchical state can be either *strong* or *weak preemption.* If a strong preemption arc (labeled with an S at the base of the

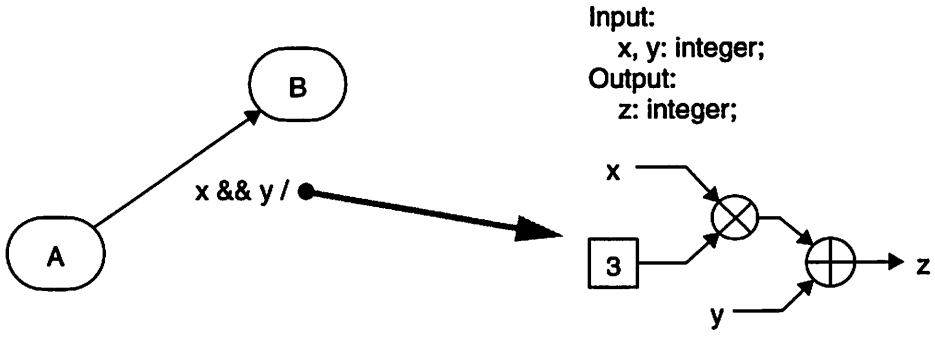


Figure 2.11 An SDF graph can be used to specify the action part of an arc label. In this example, the dataflow graph is equivalent to the statement z = 3 * x + y.

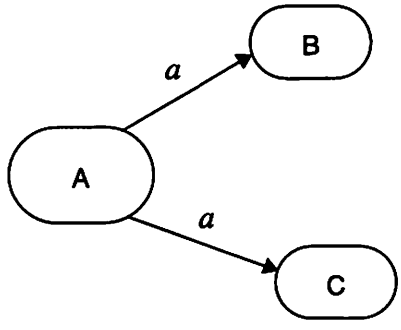


Figure 2.12 Explicit non-determinism is allowed.

arc) is triggered, the internal slave process is not executed in that firing. Therefore, the guard on a strong preemption arc must be a predicate in external input events only (the guard cannot depend on internal status events, which are generated by the slave). If a weak preemption arc is triggered, the internal slave process is executed before the master control FSM makes the transition. Exit arcs correspond to either the *stop* or *suspend* call on a thread or software process.

2.7.2 Operational Semantics

The operational semantics of this hierarchical state machine model is defined by the following simulation algorithm. We assume that a slave process can be *initialized* and *executed for a firing*, which is true of all the models of computation in Ptolemy. A firing of a hierarchical FSM consists of:

1. Emit Moore output events. Note that some of these output events may be internal control outputs for the slave process.
2. Examine strong preemption arcs of the current state. If any of them are triggered by the current input events (i.e., the guards of the arcs are true), choose one nondeterministically and go to Step 5.
3. Execute a firing of the current slave process, which may generate internal status and external output events.
4. Examine the weak preemption arcs of the current state. If any of them are triggered, choose one nondeterministically.
5. Execute the action of the chosen transition arc.
6. Check one-emitter rule: if an output event is emitted by both the master control FSM and the current slave process, flag a run-time error.

7. Enter the next state of the chosen transition arc. If it is an initial entry arc, initialize the internal slave process.

Note that in this simulation algorithm, the executions of the master control and slaves are tightly synchronized: the ratio of the firings of the master control FSM and the current slave process is 1:1. Also, the algorithm favors strong preemption arcs because strong preemption arcs are examined (and taken) before weak preemption arcs.

2.8 Discussion

In this section, I discuss the strengths, weaknesses (and how to work around them), and possible extensions to *charts and the proposed hierarchical FSM model.

2.8.1 Strengths

1. The *charts scheme is "syntactic sugar" for expressing the fact that a process controlled by an FSM is active if and only if the control FSM is in a particular state. In *charts, the process is nested in that control state. Equivalent "selective activation" behavior can be achieved in the control/process separation structure by adding an *enable* input signal to the controlled process. The control FSM asserts the *enable* signal when it is in the particular state. However, the visual syntax of *charts conveys the correspondence between modes (control states) and internal slave processes directly. One does not need to gather this information by analyzing the system description.
2. The slave processes of a master control FSM run in mutual exclusion, i.e., at any time, at most one slave process is running. This information, readily available from the *charts visual syntax, can be exploited by a synthesis tool for resource allocation because resources such as functional units (adders, multipliers, etc.) can be shared by the slave processes.

3. The *charts scheme is general because any model of computation whose processes use *events* for inter-process communication and can *initialize* and *execute a firing* can be nested within a control state.

2.8.2 Weaknesses

1. Because a process can only be nested in one state in the control FSM, *charts cannot easily express the situation where a process *P* is active in more than one state of the control FSM. A standard trick to work around this problem is to add a concurrent FSM with two states, *P_on* and *P_off*, and put the process inside the *P_on* state. The state transitions of this new FSM are synchronized with the main control FSM — it is in state *P_on* whenever the main control FSM is in any of the states the process *P* should be active, and in state *P_off* otherwise.

   The *charts scheme does not preclude the use of the control/process separation structure. The *charts scheme is most natural when the period of time in which a process is active can be identified with a particular state in a control FSM. One can always fall back to the control/process separation structure when *charts are not the most natural syntax to express the behavior.

2. Completion/termination of a process: The notion of completion is most useful when combined with concurrency – a subsystem of concurrent processes is defined to complete if all the component processes have completed. Completion of a parallel statement is a common control flow structure (often called *fork-join*). With a notion of completion, many algorithms can be specified more compactly. So some mixed concurrency and control models support the notion of completion, for example, the completion points and transition-on-completion arcs in SpecCharts [141][142][143]. The notion of completion requires a close coupling between the hierarchy and concurrency semantic components.

3. The amount of concurrency, i.e., the number of concurrent processes, is bounded, although variable.

2.8.3 Possible Extensions

1. It is possible to perform a simple one-emitter rule check (see Step 6 of our simulation algorithm) at compile time. We can require that each slave process declare all the external outputs it may possibly emit. Then for each state, we check that the master control FSM does not emit any of these external outputs. This is somewhat overly conservative because the master control FSM and current slave process may turn out to emit the same output at different times.

2. The operational semantics in Section 2.7.2 defines a *strict* execution of the master control FSM, i.e., no outputs are emitted until the status of every input event (both external and internal status) is known. Zero delay loops such as instantaneous dialogs are not possible with this strict FSM semantics. Nonstrict execution of FSMs is possible, but the syntax, such as that of Argos, can be difficult to understand. (An example taken from an Argos paper [118] is shown in Figure 2.13.) Nonstrict execution of FSMs is

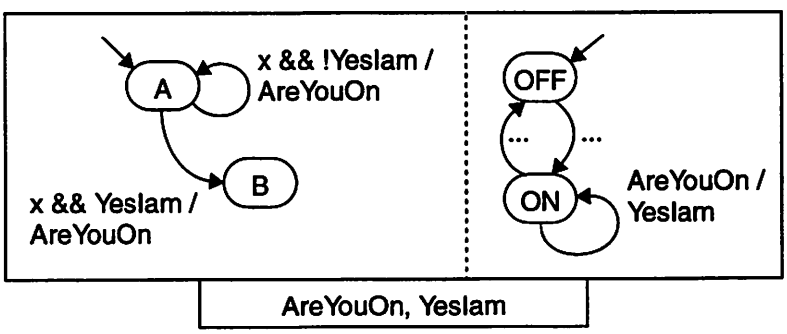


Figure 2.13 An example of an Argos program that requires partial evaluation of nonstrict functions to reach a fixpoint. If the program is in state (A, ON) and event x occurs, the program goes to state (B, ON). Note that the FSM on the left has arc labels that seem to causally reversed, i.e., the question "AreYouOn" appears after the answer "Yeslam".

also computationally expensive (an algorithm for the nonstrict execution of FSMs is given in Girault's draft paper [147]). Therefore, I do not address general nonstrict execution of FSMs here. However, the hierarchical FSM model that I propose does have some limited nonstrictness: the Moore outputs in Step 1 of the simulation algorithm can be emitted right away without knowledge of any input event.

3. Instead of having events as the communication primitive, we may alternatively have *variables* as the fundamental primitive, for both communication and representing state, and define events as changes in the values of Boolean variables. The reactive modules of Alur and Henzinger take this approach [148].

4. We may want to allow the slave processes to share state (or blocks with state) so that one slave can "pass the baton" to another slave. Hybrid automata and some other models have this feature (see the side remark in Section 2.5.2).

2.9 Conclusions

In this chapter, I described a heterogeneous approach to the design of networked multimedia applications. The functionality of networked multimedia applications can be partitioned into digital signal processing, control, user interface, etc. The heterogeneous approach combines domain-specific design styles for these diverse application components. I proposed a hierarchical state machine model called *charts, which is suitable for describing complex control functionality and is a new way to mix control and numeric computation. The *charts model hierarchically nests finite state machines with concurrency models. By mixing with different concurrency models, *charts essentially subsume all variants of Statecharts, yet remain modular. I defined the operational semantics of *charts by giving a simulation algorithm. A visual editor and simulator for *charts have been implemented in Ptolemy and Tycho. In the next chapter, I will describe the program-

ming interface of the Ptolemy implementation of *charts with a programming example of a digital watch.

3

Ptolemy Implementation and Examples

In this chapter, I describe an implementation of *charts and the hierarchical FSM model proposed in the previous chapter in the Ptolemy design environment. The implementation consists of a visual editor for the state transition diagrams of hierarchical FSMs and a technique to execute hierarchical FSMs.

Then I describe a programming example of a simple digital watch using *charts, nesting hierarchical FSMs with synchronous dataflow (SDF). I comment on the control flow structure of the digital watch and the effectiveness of *charts for describing the digital watch's behavior.

3.1 Ptolemy Implementation

3.1.1 Visual Editor for State Transition Diagrams

For schematic capture, Ptolemy has used a visual editor called Vem, developed by the Computer-Aided Design group at the University of California at Berkeley. Vem was designed to be a visual editor for VLSI layouts. Used for purposes not originally intended for, Vem is a reasonable tool for drawing block diagrams in Ptolemy, but is inconvenient for drawing the oval state boxes and curvy transition arcs common in state transition dia-

grams. Therefore, I implemented a new visual editor for state transition diagrams. The FSM visual editor is written in Itcl [149], an objected-oriented extension to the Tcl/Tk language [90], and uses the Itcl class library of Tycho [150], an extensible user-interface toolkit designed as part of the Ptolemy project.

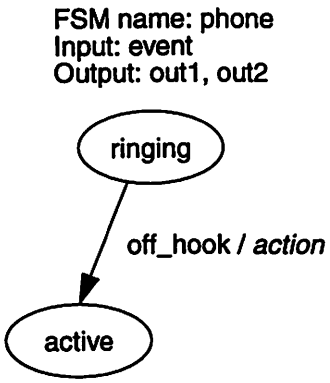
In the FSM visual editor, an FSM is intended to be overlaid on top of a programming language, such as C++, C, Java, and Tcl. Using this editor, one can draw ovals that represent states and arcs that represent transitions. For a state, one can specify a label, entry action, exit action, and an internal slave process. The entry action and exit action can be arbitrary code in the underlying programming language, which is executed when the state is entered or exited, respectively. For a transition arc, one can specify a label, guard, and action. The guard is a Boolean expression in the underlying language. The transition is taken when the guard evaluates to TRUE. The action is a piece of code in the underlying language, which is executed when the transition is taken.

For execution, an FSM is translated into an implementation in the underlying language by stitching together the code fragments specified with the states and transition arcs. A straightforward translation represents the state of the FSM with a variable and the state transitions with a function containing a *switch-case* statement, as shown in Figure 3.1. Using Tcl as the underlying language, I have applied this technique to execute hierarchical FSMs in Ptolemy, which is described below.

3.1.2 Hierarchical FSMs as Dynamic High-Order Functions

The strategy for implementing hierarchical FSMs in Ptolemy is depicted in Figure 3.2. A hierarchical FSM block is implemented for each concurrency model, e.g., SDF. A hierarchical FSM block consists of a master control block (typically a basic FSM obtained by taking the top level of the hierarchical FSM) and a number of *replacement blocks* (typically obtained by extracting the internal slave processes from the top-level states of the

hierarchical FSM). The master control *selects* one of the replacement blocks, and may optionally *initialize* them. The master control does not have to be an FSM. Other forms of control, such as knowledge-based control [151], can be used in place of the FSM to interact with the replacement blocks through the same interface. This implementation strategy is called a *dynamic high-order function* (HOF).[1]



(a)

```
proc fsm_phone_react {event} {
     global fsm_phone_state
     switch $fsm_phone_state {
       ...
       ringing {
          ...
          if {$event == off_hook} {
             action
             set fsm_phone_state active
          }
          ...
       }
       ...
     }
     return [list $out1 $out2]
}
```

(b)

Figure 3.1 Translation of an FSM, shown in (a), into an implementation in the underlying language (e.g., Tcl), shown in (b).

1. A function whose return value is also a function is called a *high-order function*. The hierarchical FSM block dynamically selects a replacement block, which can be viewed as a function mapping external inputs to external outputs, hence the name dynamic high-order function.

I have implemented the hierarchical FSM block for the SDF domain (a model of computation is called a *domain* in Ptolemy). The idea illustrated in Figure 3.2 is realized as a number of stars (a C++ class implementing an atomic block in Ptolemy is called a *star*) in the SDF domain related by the class inheritance hierarchy in Figure 3.3. The base class, the `SDFDynamicHOF` star, provides the C++ interface for the master control to switch in, initialize, and execute a replacement block. The replacement blocks, external inputs, exter-

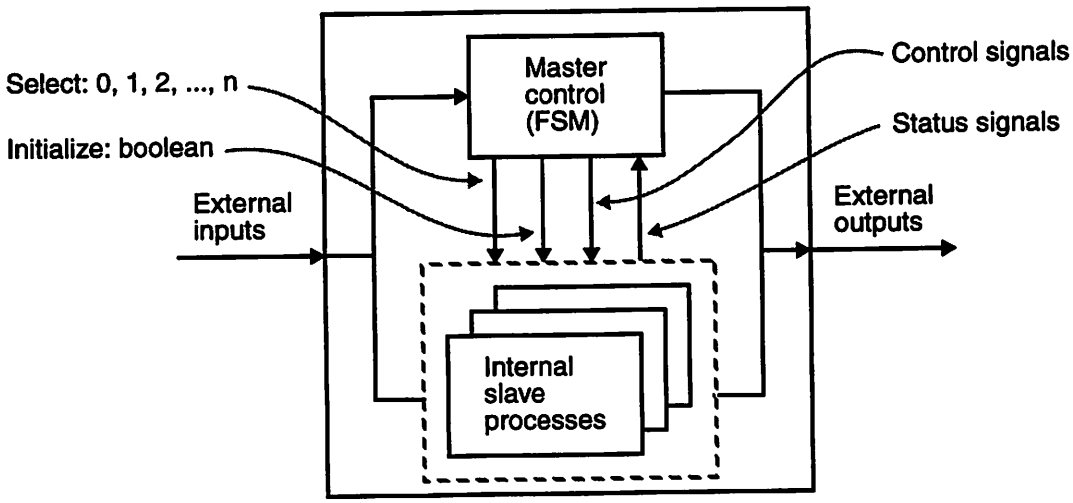


Figure 3.2 Implementation of hierarchical FSMs in Ptolemy. Select is the key primitive. An internal slave process is selected dynamically by the master control, optionally initialized. Select = 0 means strong preemption.

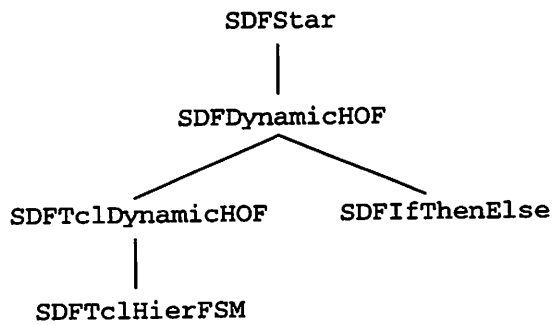


Figure 3.3 Class hierarchy of the hierarchical FSM block in the SDF domain.

nal outputs, and internal events are specified as star parameters. The public method

```
void switchBlock(const char* blockName)
```

switches in the replacement block by the given name. If the argument `blockName` is an empty string, no replacement block is switched in. (This can be used when the current state does not have an internal slave process or to implement strong preemption.) The public method

```
void initCurrentBlock()
```

initializes the current replacement block. The public method

```
void fireCurrentBlock()
```

executes a firing of the current replacement block. The protected method

```
Particle* getInternalEvent(const char* eventName)
```

returns the internal event by the given name. (An event is called a *particle* in Ptolemy.)

As an example of the dynamic HOF concept, I implemented the *if-then-else* control structure using `SDFDynamicHOF`. The `SDFIfThenElse` star, derived from `SDFDynamicHOF`, has a Boolean input named `condition`. Two blocks, `thenBlock` and `elseBlock`, are specified as star parameters. If `condition` is `TRUE`, `thenBlock` is fired. If `condition` is `FALSE`, `elseBlock` is fired. A star that implements the *switch-case* control structure can be written similarly using `SDFDynamicHOF`.

The `SDFTclDynamicHOF` star, derived from `SDFDynamicHOF`, makes the C++ methods of `SDFDynamicHOF` available in Tcl. In Tcl, every Ptolemy star is assigned a unique ID, denoted `$starID` below. The Tcl command

`switchBlock_$starID` *`blockName`*

switches in the replacement block by the given name for star `$starID`. The Tcl command

```
initCurrentBlock_$starID
```

initializes the current replacement block of star `$starID`. The Tcl command

```
fireCurrentBlock_$starID
```

executes a firing of the current replacement block of star `$starID`. The Tcl command

```
grabInternalEvents_$starID
```

returns a list of the values of the internal events. Note that the absence of an event is denoted by the value 0 (`FALSE`). This is called the `TRUE/FALSE` encoding of the presence/absence of events. If an event is absent, a null event, with value `FALSE`, is generated in its place.

Finally, the `SDFTclHierFSM` star, derived from `SDFTclDynamicHOF`, is the hierarchical FSM block overlaid on top of Tcl. A state transition diagram is specified as a star parameter. The state transition diagram is translated into a Tcl procedure that implements the state transition function, as shown in Figure 3.1. The state transition function invokes the Tcl commands of `SDFTclDynamicHOF` to switch in a replacement block (slave process), initialize it, or fire it.

3.2 Examples

3.2.1 Digital Watch

The digital watch is perhaps the most common programming example cited by almost every specialized language or model for specifying control functionality, such as Statecharts [114], Esterel [152][153][154], and Argos [144]. The digital watch has complicated mode switching but simple data computation.

I have programmed a simplified version of the digital watch using the *charts implementation in Ptolemy. The function of the watch is simplified, but the essential characteristics of the control flow are kept. The user interface of the watch consists of four buttons, labeled UL (upper left), UR (upper right), LL (lower left), and LR (lower right), and a graphic display, as shown in Figure 3.4. The digital watch has five modes: watch mode,

set-watch mode, stopwatch mode, alarm mode, and set-alarm mode. The four buttons and graphic display are shared by the five modes. The watch has a clock signal source that generates an event every second.

Shown in Figure 3.5 is the top-level Ptolemy system for testing the digital watch. The top-level system is in the discrete-event (DE) domain. The four buttons and the clock signal are generated by Tcl/Tk code. The button pushes and clock events are the input events

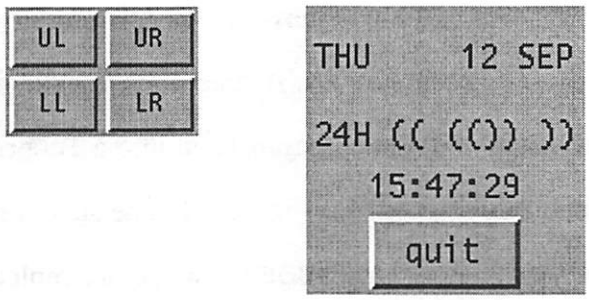


Figure 3.4 The four buttons and graphic display of the digital watch.

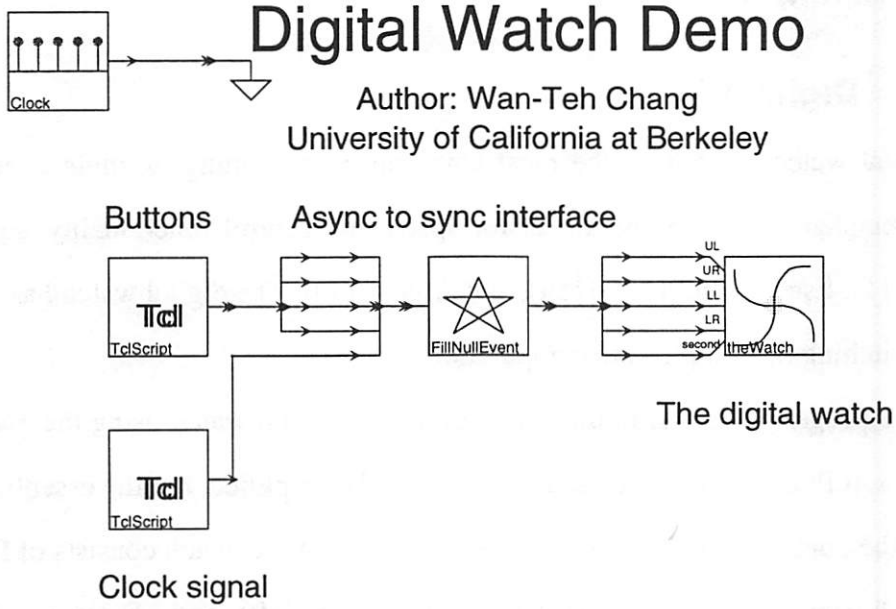


Figure 3.5 The top-level Ptolemy system for the digital watch.

for the watch. Before going into the watch proper, which is programmed in the SDF domain, the events first go through an asynchronous-to-synchronous interface, which performs TRUE/FALSE encoding of the presence/absence of events.

The watch proper consists of a number of blocks in the SDF domain, as shown in Figure 3.6. Three of the blocks are hierarchical FSMs: the control kernel (button interpreter), clock status, and alarm status. The control kernel of the digital watch keeps track of mode switching and interprets the meaning of the buttons according to the current mode. The top-level FSM of the control kernel is shown in Figure 3.7. It consists of the five modes of the watch. Button LL cycles through the three main modes: watch, stopwatch, and alarm. Button UL, if pressed in watch or alarm mode, enters the corresponding *set* mode, and if pressed again, leaves the corresponding *set* mode.

Inside the set-watch mode is another FSM, shown in Figure 3.8. Various components of the date and time can be set. Button LL cycles through second, hour, minute, month, day

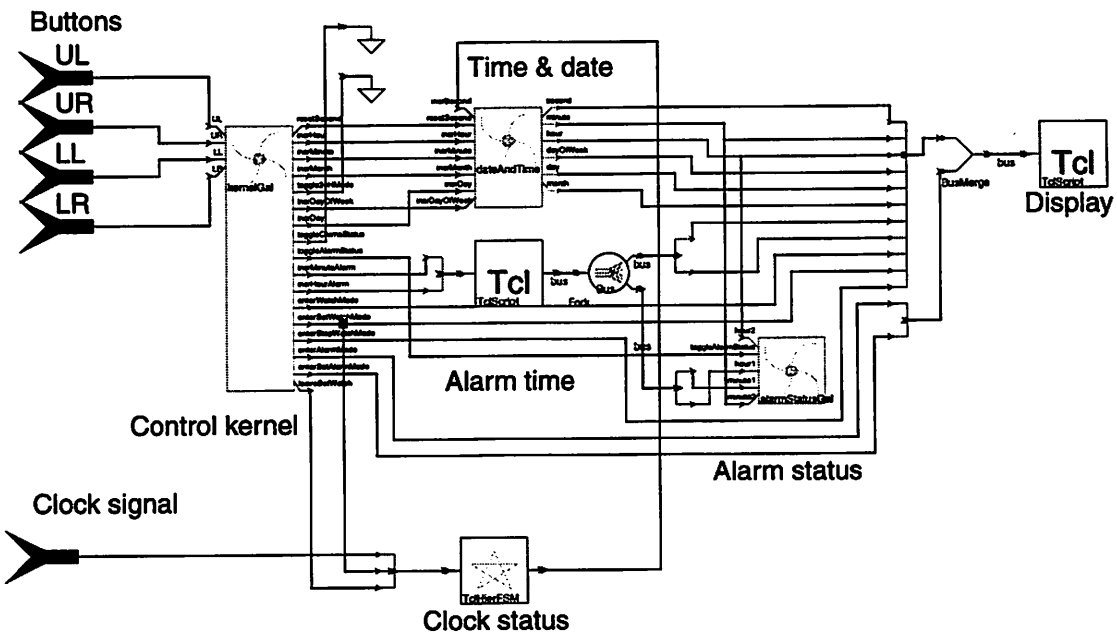


Figure 3.6 The watch proper is a number of interacting blocks in the SDF domain.

(of the month), and day of the week. Button LR increments the current component of the date and time, except for second, which is reset to 0. Note that if button UL is pressed in the set-watch mode, the watch switches back to the watch mode.

There are three counters: a date-and-time counter for the watch mode, a counter for the alarm time, and a counter for the stopwatch. The date-and-time counter for the watch mode is an SDF galaxy (a subsystem is called a *galaxy* in Ptolemy), shown in Figure 3.9. A stage of the counter, recording one component (second, minute, hour, etc.) of the date and time, is defined first. Then the counter is implemented as a series of stages. When not in the set-watch mode, the counter is incremented on each *second* event in the clock sig-

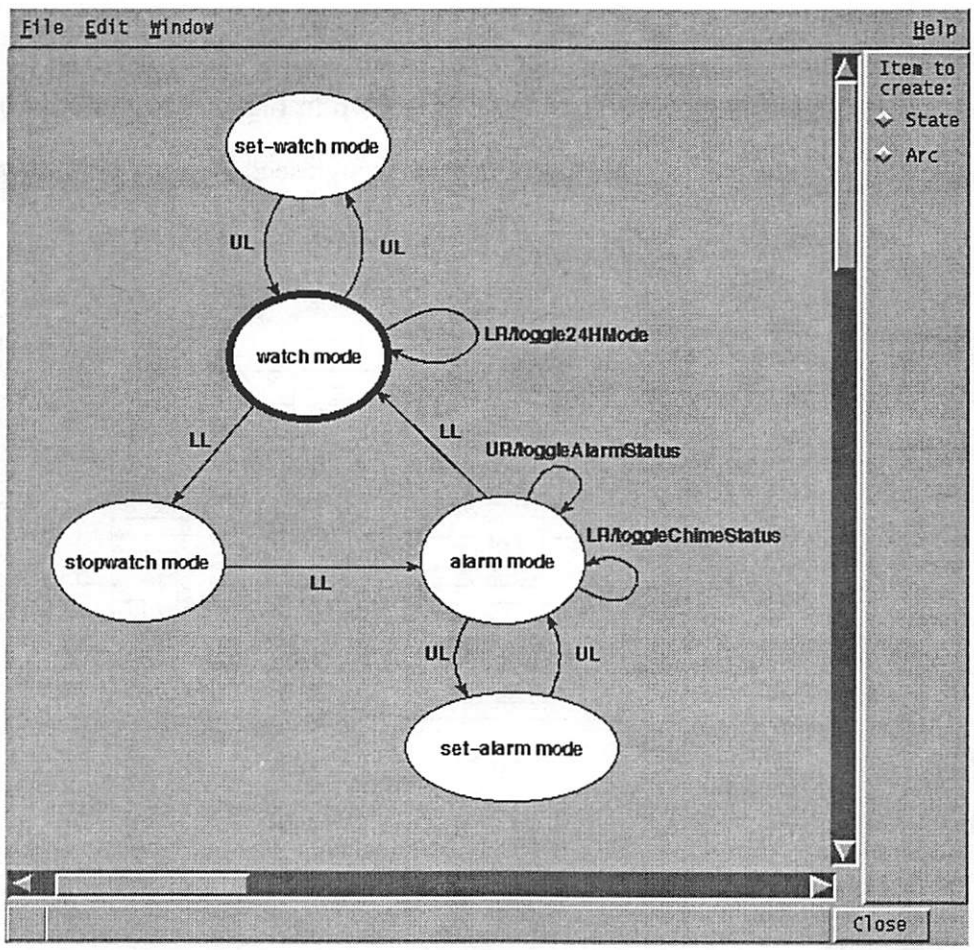


Figure 3.7 The top-level FSM of the control kernel of the digital watch.

nal. When in the set-watch mode, the counter is incremented by the control commands from the control kernel.

Remark: The main complexity of control in the digital watch is in the control kernel, and the complexity of the control kernel results from using the same four buttons to invoke different operations when in different modes. If the digital watch could have a graphical user interface, which could create and destroy new buttons (and displays) on demand, the complexity due to the control kernel's interpreting the buttons would be gone. Also, the communication between the control kernel FSM and the other two concurrent FSMs (clock status and alarm status) is unidirectional: the button commands interpreted by the control

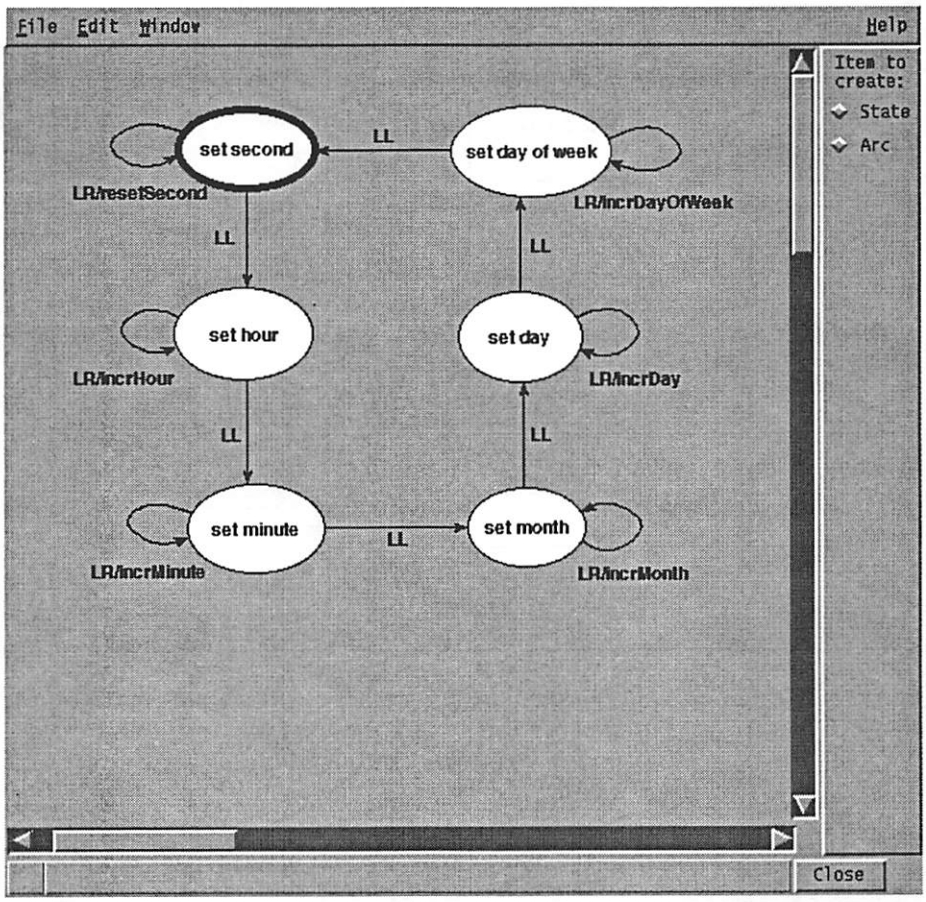


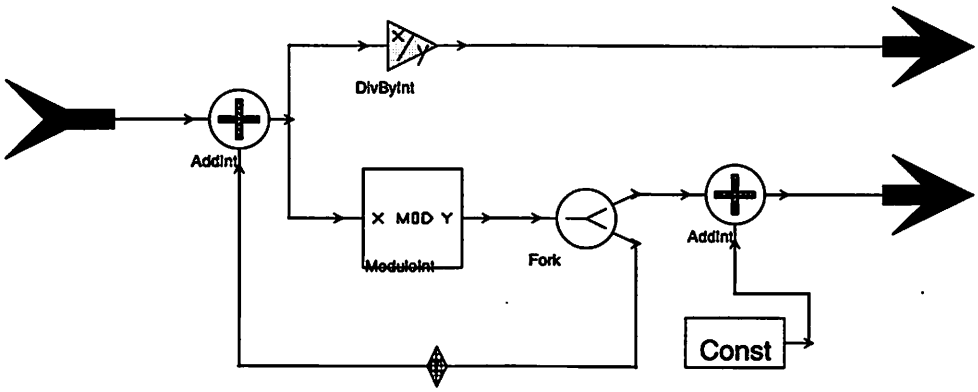
Figure 3.8 The set-watch mode of the digital watch.

kernel are sent to the other concurrent blocks. Therefore, the concurrency model of synchronous dataflow, which forbids delay-free loops, is sufficient.

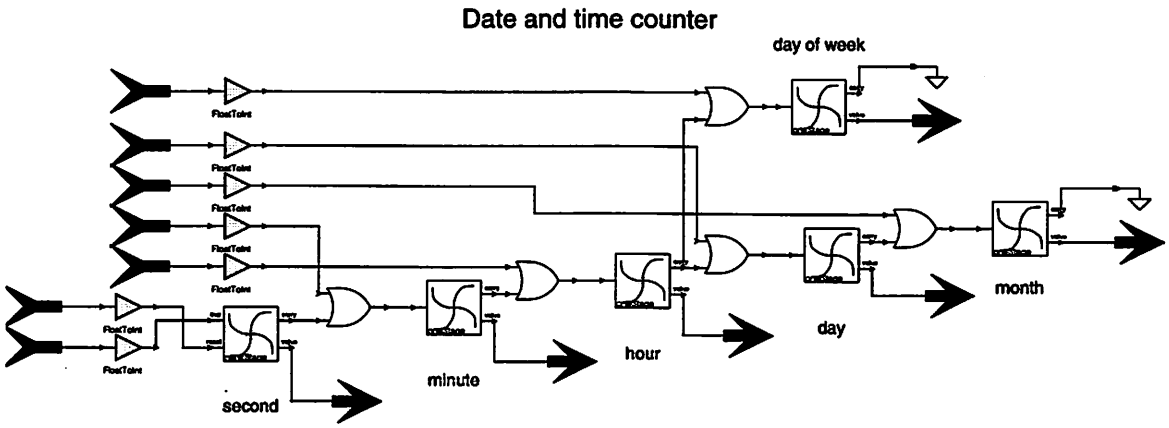
The next two examples, although not implemented, have been studied regarding the complexity of control and control/dataflow interaction.

3.2.2 Telephone Answering Machine

The telephone answering machine is the example used in a SpecCharts paper [141]. Similar to the digital watch, the control flow in an answering machine is complicated. There is



(a)



(b)

Figure 3.9 (a) One stage of the date-and-time counter (b) The date and time counter as a series of stages.

some amount of signal processing, but the control actions on the signal processing tasks are coarse-grained: start and stop recording a message, start and stop playing a recorded message, etc. In the reverse direction, DTMF (dual tone multiple frequency, i.e., touch-tone) digit detection is a signal processing task that generates events to the control.

3.2.3 Video Encoder: MPEG and H.261

A video encoder is an interesting example of signal processing algorithms. The MPEG encoder has three coding modes for the three kinds of video frames in MPEG: I, P, and B [78][79]. The I, P, and B frames are encoded differently. H.261, an international standard for videophone and video conferencing, has two coding modes: *interframe* and *intraframe* [155]. There is a significant amount of signal processing, e.g., motion compensation, discrete cosine transform, quantizer, and run length coding. The control of the signal processing tasks is more fine grained, at the *block parameterization* (e.g., the step size of the quantizer can be adjusted) and *system reconfiguration* (e.g., different ways to encode I, P, and B frames) level.

The selection of the coding modes usually follows a fixed sequence: IPPPBPPPI. But it is also possible to change that fixed sequence in response to the results of a pre-processing stage. In H.261, two *switch* controls are used to reconfigure the encoder for interframe and intraframe coding modes.

Feedback control is used on the output buffer to maintain a fixed-rate output. The buffer control monitors the status of the buffer, such as how full it is, and generates control parameters such as the quantizer step size.

In conclusion, simple *select/switch* control together with control parameters should be sufficient for describing the control/data interaction in these video compression algorithms.

4

Dynamic Network Deployment

4.1 Introduction

Networked applications (in both peer-to-peer and client-server architectures) inherently require that their constituent networked components be interoperable. Peer-to-peer applications are far less diverse than client-server applications. I contend that this is primarily due to the greater hurdle faced by new peer-to-peer applications in achieving interoperability. In a nutshell, in a client-server application, the server is persistent, and a single user who wants to participate in the application merely has to obtain the interoperable client software. In a peer-to-peer application, on the other hand, two or more users who wish to participate in the application have to anticipate that need and install the software. This makes opportunistic or casual participation in such applications unlikely.

The difference is particularly obvious from an economic perspective. When a server is established, any client (or rather its associated user) immediately derives full benefit from that server. On the other hand, a company wanting to establish a peer-to-peer application faces the obstacle that the first user derives zero benefit; more generally, the benefit increases with the number of users. Economists call this the problem of *network externality* [91][92][93].

The network externality problem can be solved by a speedy mechanism to distribute an application to a large number of users simultaneously. For software-defined applications, this is technically feasible, since software can be distributed over the network. In the Internet, developers of client-server applications, such as World Wide Web browsers, document viewers, and audio and video players, are distributing new versions of those applications over the network. By bypassing traditional slow distribution channels, the velocity of innovation has been increased dramatically.

The network distribution of applications has the potential to make a much bigger impact on peer-to-peer applications than client-server ones, since getting those applications to many users simultaneously is the key to overcoming network externality. Nevertheless, the current approach in the Internet, in which the user has to anticipate the need for an application and execute the relatively sophisticated and manual file transfer and installation, remains a barrier. Other obstacles are multiple instruction sets and operating systems (collectively referred to as *platforms*) and the security problems associated with downloading binary executables from untrusted sources.

4.2 Dynamic Deployment of Peer-to-Peer Applications

With the ever-increasing speed of microprocessors, applications, including real-time applications like voice and video, are increasingly implemented in software on high-performance programmable user terminals, such as desktop or portable computers and the *network computers*[1] that Oracle and Sun Microsystems plan to introduce in the near future. With high-speed networking, application descriptions can be downloaded quickly

1. A network computer, also called an Internet appliance, Web PC, etc., is an inexpensive diskless computer with minimal built-in software (e.g., a Web browser, a Java interpreter, and a word processor) and gets most other software from the network [156].

from some central repository to the terminals as part of session establishment, transparently to the user. This is called the *dynamic deployment* of applications.

There are three elements in the infrastructure for dynamic deployment, as illustrated in Figure 4.1:

- A standardized *virtual machine* in the programmable terminal. The virtual machine hides the differences in the underlying platforms and provides a layer of protection.
- Standardized *application description languages.*
- A standardized *protocol for transferring application descriptions* from some central repository to the terminals as part of session establishment. Alternatively, one of the terminals, such as the terminal on the left in Figure 4.1, can serve as its own repository, and the application description could be transferred to the other terminal.

While standardization of some aspects of the infrastructure is presumed, we hope to avoid standardization of any part of the application itself. Also, interoperability is guaranteed

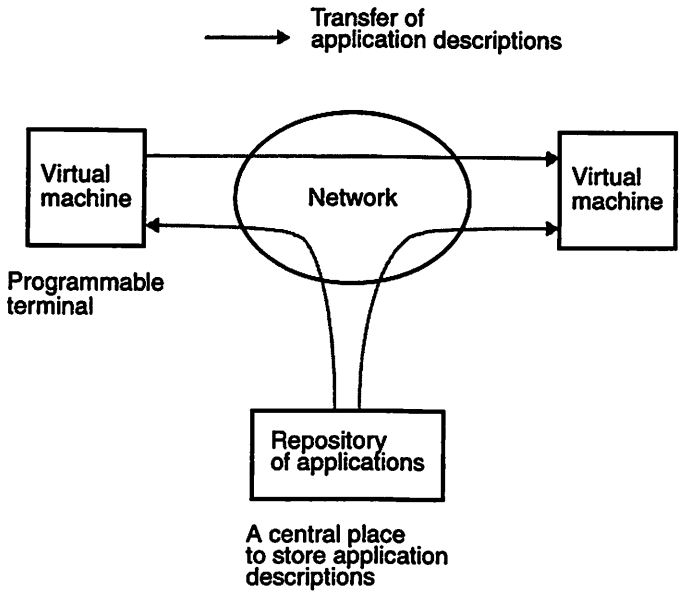


Figure 4.1 Dynamic deployment of peer-to-peer applications.

because interoperable peer descriptions are downloaded to the terminals from a common source.

So far, dynamic deployment has been applied primarily to client-server applications, such as adding functionality to a WWW browser. It should have a much greater impact on peer-to-peer applications, since it bypasses the obstacle of network externality. Peer-to-peer applications interoperable over the network can be established, without prior standardization or even the need for users to obtain the requisite software in advance, to a community of interest consisting of all networked implementations of the virtual machine.

A peer-to-peer application is not symmetric at session establishment. Rather, one peer is the *originating* peer, whose task is to initiate the session, and the other peer is the *responding* peer, which participates in the application in response to a session establishment request. Roughly, the steps to set up a dynamically-deployed application are as follows:

- The originating peer places a session establishment request.
- The responding peer is alerted of an incoming establishment request.
- If the responding peer accepts the establishment request, the application description in the form of two peer application programs is transferred to the peer terminals.
- The two peer programs initiate a peer-to-peer session.

The dynamic deployment infrastructure helps the peer programs establish their first connection (through a procedure called *session coordination*, described in Section 4.6.4). From that point on, they are on their own to define any application functionality they choose.

4.3 Issues

Despite its many advantages, dynamic deployment has some limitations and critical issues that must be addressed:

- For network distribution, the applications must be defined in software. Hence the application functionality is constrained by what can be realized in software.
- High-speed networking is required to make application download time (which contributes to the session establishment delay) reasonably short.
- The security risk due to executing software downloaded from untrusted sources is a critical issue. Therefore, the downloaded application software should not be native binary executables, but rather written in application description languages that are interpreted or compiled by the terminals.
- The same program should run on terminals with different processors, operating systems, and windowing systems. This also requires that the application software be written in portable application description languages.
- There is run-time or establishment-time performance penalty due to the interpretation or compilation of application description languages.
- Dynamic deployment is a form of on-line software distribution. Therefore, there must be protection mechanisms (e.g., using cryptographic techniques) for copyrighted material. The licensing and pricing of software distributed in this model are also important issues. Messerschmitt's white paper has some discussion on these emerging issues [9].

4.4 Dynamic Deployment Based on Java and World Wide Web

My work on the dynamic deployment of networked applications was first inspired by an unpublished manuscript of Messerschmitt's, in which he proposed *intelligent signals* (object-oriented signals that contain not only the data streams but also the method code to manipulate the data) as a means to manage the complexity of setting up flexible connections [157]. We previously published the general idea of the dynamic deployment of application functionality to programmable terminals as a means to deploy telecommunications applications rapidly [94]. In that work, however, a proprietary environment, Ptolemy, has to reside within each terminal as the application run-time environment, which still presents a problem of network externality.

Since our goal is to establish a framework within which arbitrary peer-to-peer applications can be dynamically deployed, ideally this framework should be a standard commercial environment that is widely available to potential participants in peer-to-peer applications. To this end, I have designed and implemented a prototype with the WWW environment, using Java as the description language for peer-to-peer applications. (An extended abstract on this prototype was published earlier this year [95].) However, the current WWW/Java environment has been designed for client-server applications. I will show below that by some clever reorientation of the existing Java-enabled WWW browser framework, it can serve as an environment for peer-to-peer applications.

Java, developed by Sun Microsystems, is a programming language specially designed for writing programs that are to be executed by a remote computer [73][74][75]. Java uses the virtual machine execution model. Java programs are compiled to the instructions (called *bytecode*) for the Java virtual machine to achieve platform independence. Moreover, Java has built-in support for security. Therefore, Java is well-suited to serve as the first two elements of the dynamic deployment infrastructure described in Section 4.2, the

virtual machine and the *application definition language*. It is the third element, a *session establishment protocol* incorporating the distribution of application descriptions to terminals, that is missing. My work adds this missing piece to the standard Java/WWW framework.

Java's primary application at present is writing executable programs (called *applets*) that are embedded in HyperText Markup Language (HTML) files and executed by Java-enabled WWW browsers elsewhere on the network. The present WWW browsers use a "hyperlink following, file pulling" browsing mechanism. This simple browsing mechanism can download and run Java applets on just a single WWW browser; it can set up Java applets that are stand-alone applications or clients in some client-server networked applications. The current WWW browsing mechanism, however, cannot set up peer-to-peer applications. In the establishment procedure for peer-to-peer applications, some user interaction, such as alerting the responding user of an incoming session request, is needed. Moreover, if the functionality of the two peers are both defined by Java applets, it is necessary to download the Java applets to their respective WWW browsers, execute them, and help them connect to each other. The session establishment procedure requires coordination among the two peers' WWW browsers and Java applets.

I have designed a scheme to deploy Java applets dynamically to two WWW browsers that together implement a peer-to-peer application. (It would be fairly straightforward to extend to three or more peer WWW browsers.) I add two helper programs running in conjunction with WWW browsers and a coordination protocol built on top of TCP/IP and HTTP (HyperText Transfer Protocol). Only standard commercial software is used, supplemented by code written in Java. Thus, I have demonstrated that standard WWW browser applications and server software can support peer-to-peer applications, even though they were originally defined for a client-server environment. The scheme and a prototype implementation are outlined next.

4.4.1 System Architecture

The system organization is illustrated in Figure 4.2. There are two peers: the originating (local) peer and the responding (remote) peer.

4.4.1.1 Local, Originating Peer

The local, originating peer has the following environment:

- It should have an HTTP server running.
- The local peer may have some Java applets defining the two halves of peer-to-peer applications (one applet for the local peer and the other for the remote peer) and the HTML template files in which the applets are embedded. (These HTML files are called *template files* because *session files* are created by inserting some additional applet parameter tags into the template files. See the explanation on session files in

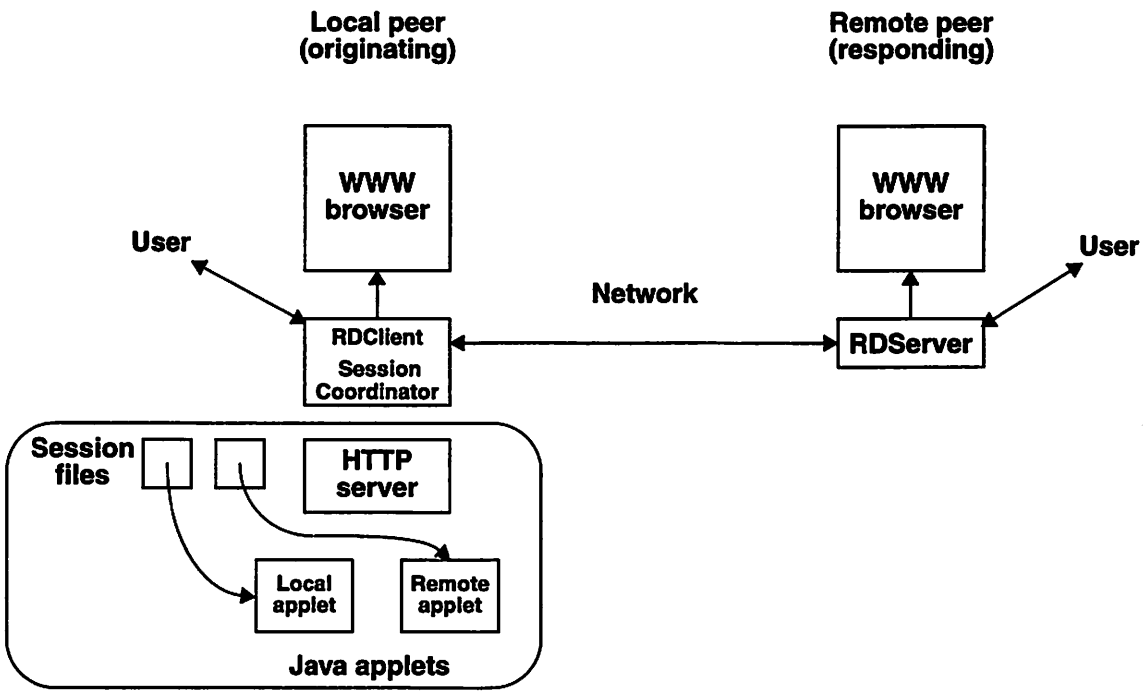


Figure 4.2 System architecture of the dynamic deployment prototype based on WWW and Java.

Section 4.4.2.) In fact, in the prototype described in Section 4.6 I assume that all the peer-to-peer applets are stored at the local peer. This is the case where the local peer serves as its own repository of applications, as shown in Figure 4.1.

- The local peer should run a Java-enabled WWW browser such as Netscape Navigator and a companion helper program `RDClient`. The local peer should have a startup file for `RDClient` called `.rdclientrc`, which is a list of the available peer-to-peer applications and the Uniform Resource Locators (URLs) of their HTML template files, and a `sessions` directory that is exported by its HTTP server.

Remark: The prototype described in Section 4.6 does not implement the central repository case in Figure 4.1 because that case is not allowed by Netscape's stringent applet security policy. A remote applet running inside Netscape Navigator can only open network connections back to its originating host (the *applet host*). This security precaution is meant to prevent the applet from impersonating the host it is running on and attacking other hosts on the network. On the other hand, there is no restriction on the networking capabilities of a local applet. A local applet can open a listening socket or open a network connection to any host on the network. In a peer-to-peer application, the two peers' applets must connect to each other. To work around Netscape's applet-host security policy, I therefore require that the local (originating) peer have an HTTP server and the Java applets that define the peer-to-peer application. This way, the local peer's applet can open a listening socket, and the remote peer's applet, coming from the local peer's HTTP server, can connect to the local peer's applet. In the future, when a *signed applet* (an applet digitally signed with cryptographic means by a trusted party) is allowed to open network connections to any host, my scheme should work for the central repository case too. More information on Java applet security can be found in Sun's Applet Security Web page [158].

4.4.1.2 Remote, Responding Peer

The remote peer should run a Java-enabled WWW browser and a companion helper program `RDServer`, which listens at a well-known TCP port for connections.

4.4.2 Session Establishment Procedure

Session establishment takes place in three stages:

- Stage 1: The local, originating user places a call and the remote, responding user is invited to participate (see Figure 4.3).
- Stage 2: If the responding user accepts the call, session files are generated and downloaded to the two WWW browsers (see Figure 4.4).
- Stage 3: The two Java applets embedded in the session files start running and connect to each other through a session coordination procedure (see Figure 4.5).

Figure 4.3 shows the first stage in session establishment. When the local, originating user wants to initiate a peer-to-peer session (i.e., place a call), the user interacts with the

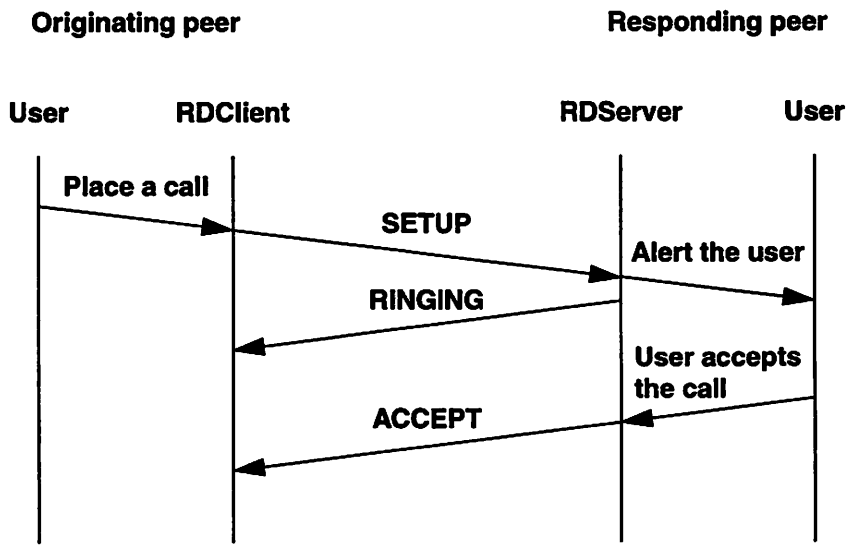


Figure 4.3 Stage 1 in session establishment: peer invitation.

RDClient to choose an application and specify the remote, responding peer. Then RDClient connects to the RDServer on the responding peer, passing it a SETUP message that contains the originating host name (which RDServer can also obtain from the TCP connection), user's name (which RDServer has no way to verify), application name, and optionally a short description of the application. RDServer alerts its user of an incoming establishment request with a pop-up dialog, giving the information contained in the SETUP message, and lets the responding user decide whether to accept or reject the establishment request. RDServer then tells the local RDClient whether the responding peer accepts or rejects the establishment request, or it timed out (no one answers the establishment request).

If the remote peer accepts the establishment request, RDClient picks a random (but unique) *session number* for this session, and generates two session files (one for the local peer and one for the remote peer) by inserting some applet parameters into the applets' HTML template files. The inserted applet parameters are used to pass information to the applets. The inserted applet parameters are:

- session (session number) and coordport (session coordinator's listening port), for both applets;
- orighost (the local, originating host), for the remote applet only.

The session files are created in the local peer's sessions directory with random file names so that intruders cannot easily guess the file names. The sessions directory is made unreadable so that intruders cannot list that directory.[1]

1. I tried to make the session establishment procedure secure by using random session numbers and session file names and making the sessions directory unreadable. However, the communication between RDClient, RDServer, and the peer applets is not encrypted. A malicious person could eavesdrop on the conversation and do a "denial-of-service" attack. For example, during session coordination, the TCP port number that the remote applet should connect to is passed in plain text over a TCP connection. Someone may intercept this TCP port number and connect to the port before the remote applet does, preventing it from establishing a connection. To be more secure, all conversation should be encrypted.

Then the second stage in session establishment begins (see Figure 4.4). `RDClient` tells its local companion WWW browser to retrieve (from the local HTTP server) and display the local session file. `RDClient` passes the URL of the remote session file to the remote `RDServer`, which in turn tells its companion WWW browser to retrieve and display the remote session file (therefore downloading and executing the remote Java applet).

Since the order in which the local and remote Java applets are executed is arbitrary, we need to help them synchronize and connect to each other. They need to synchronize because the local applet must open a listening socket before the remote applet connects to it. Moreover, we do not want to use a fixed, predefined port number for the peer-to-peer application. So the local applet picks an unused TCP port number, which must be passed to the remote applet. The local `RDClient` provides this synchronization and coordination service, called the *session coordinator*. The session coordinator allows the two applets to

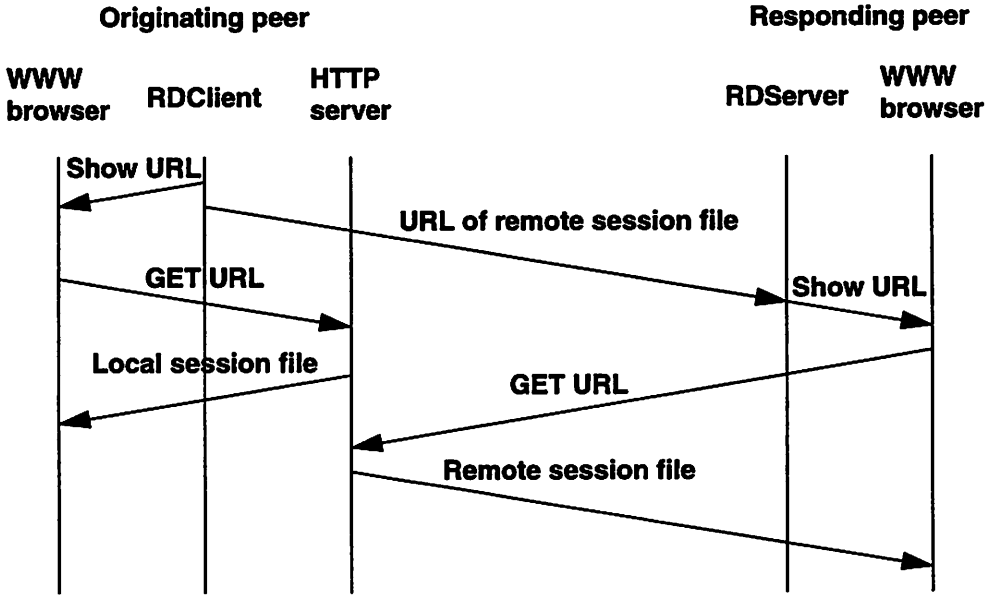


Figure 4.4 Stage 2 in session establishment: downloading the session files.

synchronize and exchange information, e.g., the TCP port number the local applet is listening on. The local and remote applets identify each other using the session number.

Figure 4.5 shows the session coordination procedure, which is the third stage in session establishment. After the local applet opens its listening socket, it connects to the session coordinator (by using the `coordport` parameter) and gives its TCP port number to the session coordinator. (The local applet is said to *check in*.) The remote applet (by using the `orighost` and `coordport` parameters) also connects to the session coordinator and waits until its local peer has connected to the session coordinator. Then the session coordinator passes the TCP port number to the remote applet. At this point, the remote applet knows that the local applet is ready and listening at that port number, so it opens a network connection to the local applet. The session is considered established from the point of view of `RDClient` and `RDServer`. They are only responsible for helping the local and remote applets establish this first network connection. If the local and remote applets need more network connections, they should use the first connection to coordinate and pass the necessary information.

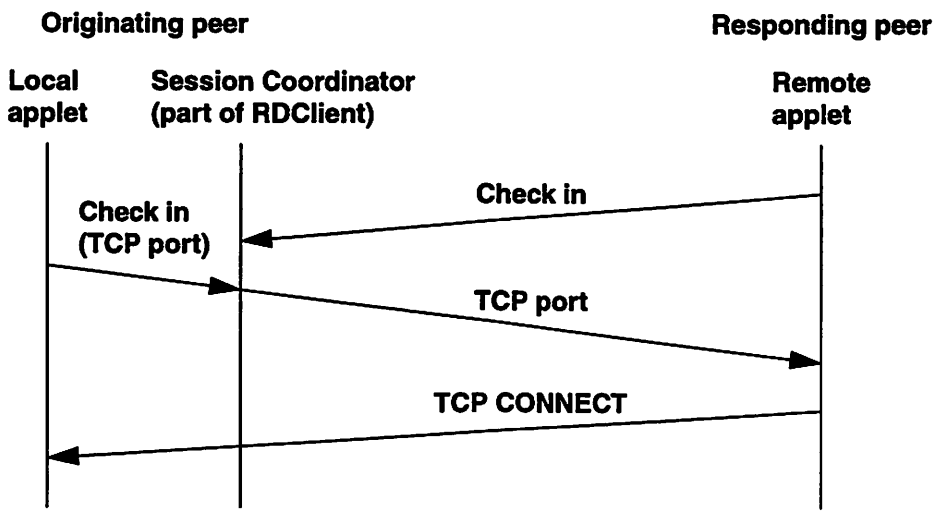


Figure 4.5 Stage 3 in session establishment: session coordination.

4.5 Related Work

The most closely related work is the *Promondia* project (formerly known as *Como*) at the University of Erlangen-Nürnberg, Germany [159]. Promondia is a system for dynamically deploy networked applications implemented in the *client-server* architecture. The Promondia servers, usually running in conjunction with HTTP servers, provide a directory of currently on-line users. A user connects to a Promondia server, looks in the on-line user directory, and picks a party to communicate with. Promondia servers also provide the *reflector* function, i.e., a user does not communicate with another user directly; users communicate via the Promondia servers. (The reflector function is obviously necessary to work around the applet-host security policy's constraints on an applet's networking capabilities. See the remark on page 82.)

The differences from my peer-to-peer approach are:

- In Promondia's client-server approach, users participate in sessions in the *meeting room* model. Users have to connect to the Promondia servers, and only currently connected users can be contacted. In my peer-to-peer approach, users participate in sessions in the *telephone* model. A user can directly contact any other user, provided that the called user's computer is on and running a Web browser and the `RDServer` helper program. (A possible solution to this constraint is to have a *daemon* program running to accept connection requests and start up the needed Web browser and helper.)
- In Promondia's client-server approach, there needs to be some service provider to run the Promondia servers, but users do not need any additional software other than the regular Web browsers. In my peer-to-peer approach, users need to run additional helper programs in conjunction with their Web browsers, but they do not need to rely on any service provider to run servers of any kind.

4.6 Implementation

I have built a prototype implementation of the dynamic deployment scheme using Sun SPARCstations running the Solaris 2.5 operating system, Netscape Navigator Version 3.0, NCSA HTTP/1.0 server (httpd), and Sun's Java Developer's Kit (JDK) Version 1.0.2.

Remark: I use HTTP for transferring applets because Netscape Navigator contains a built-in HTTP client. (So FTP would be an equally good choice.) This way we just need to run an HTTP server on the initiating peer. The responding peer does not need to run additional software; it can simply use the Netscape browser to download the applets. Note that only a small subset of HTTP is used, so one can write a stripped-down version of HTTP server for use in the prototype.

The prototype is divided into two helper programs, `RDServer` and `RDClient`, that run in conjunction with the Netscape Navigator. (RD stands for *rapid deployment*, the former name of this project.) `RDClient` lets a user place a call setup request and sees through the whole call setup procedure. `RDServer` waits for call setup requests from `RDClients` on other computers.

The prototype is implemented in the Java programming language and consists of multiple threads organized into thread groups. The system resources used by the implementation are listed below:

- Files: `RDClient` reads a text file `.rdclientrc` when it starts up. It creates temporary session HTML files in the `sessions` directory during call setup.
- TCP ports: `RDServer` listens on the fixed TCP port 9000. The session coordinator (part of `RDClient`) listens on an unused TCP port (called the `coordport`) chosen at run time.
- Threads: To set up an n-party call, `RDClient` on the initiating peer starts a `CallSetupThread` and $n-1$ `InvitePeerThreads` to talk to the `RDServers`

on the $n-1$ responding peers. For the common special case $n = 2$, a `TwoParty-CallSetup` thread may be used instead. During the session coordination phase, a `SessionCoordHandler` thread is started for each peer applet by the `Session-CoordDispatcher` thread listening on `coordport`. `RDServer` on a responding peer is itself a thread. It listens on TCP port 9000 and starts a `ClientHandler` thread to talk to the `InvitePeerThread` or `TwoPartyCallSetup` thread of the `RDClient` on an initiating peer.

4.6.1 The Startup File .rdclientrc

`RDClient` reads a text file named `.rdclientrc` when it starts up. An example of the `.rdclientrc` file is shown in Figure 4.6. The syntax of `.rdclientrc` is based on Tcl. A # character indicates the beginning of a comment, and a backslash \ at the end of a line continues the line. The `.rdclientrc` file specifies the following information:

- `sessionsDir`: the pathname of the `sessions` directory, in which temporary session HTML files are created. On Unix, I suggest using the `public_html/sessions` directory under the user's home directory.
- `sessionsURL`: the URL of the `sessionsDir` directory exported by the local peer's HTTP server. In the example, a user named `wtc` on a Unix workstation named `markov.eecs.berkeley.edu` running an HTTP server on TCP port 8080 uses the `sessionsDir` directory suggested above. Then `sessionsURL` is

  `http://markov.eecs.berkeley.edu:8080/~wtc/sessions`.

- Information about each available application starts with the keyword `application`, followed by the application name and the configuration information enclosed in curly braces {}. (The configuration information is used in the "PLACE CALL" dialog in `RDClient`'s graphical user interface. See Section 4.6.2 below.) Each type of party is

described. The keyword `origPeer` specifies the originating party; the `templateURL` keyword specifies the URL of its session template file (in which its applet is embedded). For each type of responding party, there is a `respPeer` block that gives a short description and specifies the number of instances (`one` or `multiple`, e.g., in a

```
# The startup file for RDClient.

sessionsDir /users/wtc/public_html/sessions
sessionsURL http://markov.eecs.berkeley.edu:8080/~wtc/sessions

application MiniApp {
    origPeer {
        templateURL \
http://markov.eecs.berkeley.edu:8080/~wtc/java/apps/MiniApp/origpeer.html
    }
    respPeer {
        description "Called party"
        numInstances one
        templateURL \
http://markov.eecs.berkeley.edu:8080/~wtc/java/apps/MiniApp/resppeer.html
    }
}

application JavaTalk {
    origPeer {
        templateURL \
http://markov.eecs.berkeley.edu:8080/~wtc/java/apps/JavaTalk/talkorig.html
    }
    respPeer {
        description "Called party"
        numInstances one
        templateURL \
http://markov.eecs.berkeley.edu:8080/~wtc/java/apps/JavaTalk/talkresp.html
    }
}

application WhiteBoard {
    ...
}

application ChalkBoard {
    ...
}
```

Figure 4.6 The .rdclientrc file for user wtc on Unix workstation markov.eecs.berkeley.edu running an HTTP server on TCP port 8080.

conference, there can be more than one participant who runs the same applet) and the URL of its session template file. For example, in Figure 4.6, "MiniApp" is the name of a minimal peer-to-peer application, which will be used as a programming example. The URL of the session template file for its originating party is given. It has one other type of party, with the description "Called party." There can be only one instance of "Called party," and the URL of its session template file is given.

4.6.2 Placing a Call

A user places a call through `RDClient`. `RDClient` opens a top-level window called `RDClientFrame` (in Java, a stand-alone top-level window is called a *frame*), consisting of four buttons (see Figure 4.7). When a user pushes the "PLACE CALL" button in `RDClientFrame`, a "PLACE CALL" dialog window pops up (see Figure 4.8). A list of available applications (obtained from the `.rdclientrc` file) is displayed. The user selects an application from the list. The configuration information of the selected application is used to construct the entry boxes for the user to fill in the addresses of the other parties. In the example, the user selects the application MiniApp. There is only one other party, with the description "Called party." After the user fills in the address of the called party and presses the "OK" button, a `CallSetupThread` is started to carry out the call setup procedure. Note that multiple sessions can be in progress simultaneously.

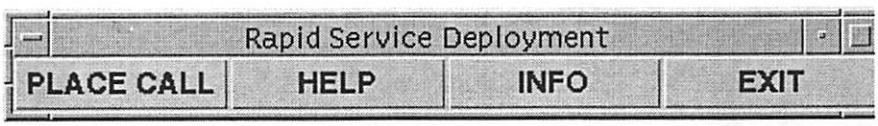


Figure 4.7 RDClientFrame, the top-level window of RDClient.

4.6.3 Call Setup Procedure

The call setup[1] procedure is illustrated in Figure 4.9. The `CallSetupThread` first creates a `CallSetupRecord` to record the progress of the call setup. Then it starts a number of `InvitePeerThreads`, which are collectively put in the "Peer invitation threads"

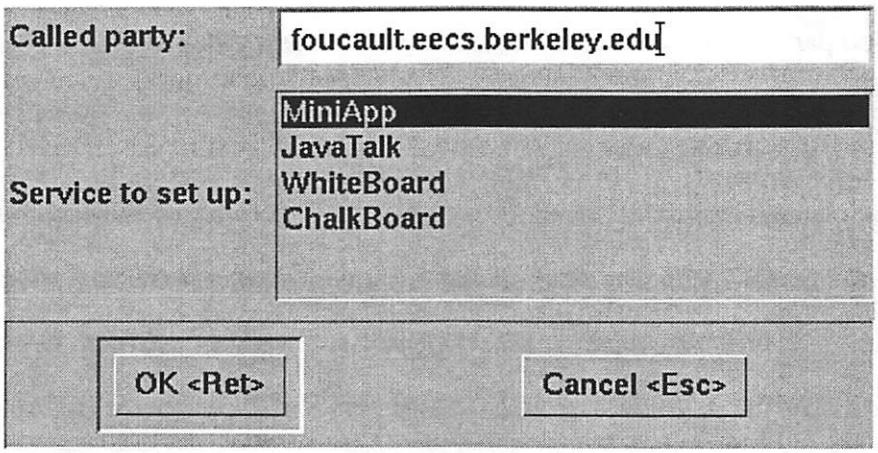


Figure 4.8 The "PLACE CALL" dialog. The user selects an application from the list and fills in the addresses of the other parties.

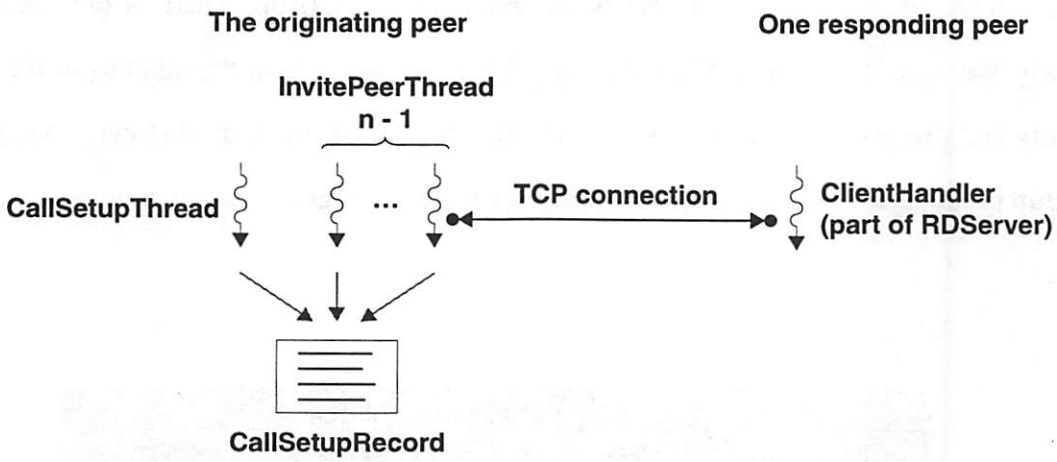


Figure 4.9 The threads in call setup procedure. The protocol over the TCP connection is shown in Figure 4.3 and part of Figure 4.4.

1.In the Java code I have used the shorter term *call setup* to refer to session establishment.

thread group, to invite the other peers to join the session. Each `InvitePeerThread` is responsible for talking to the `RDServer` on one responding peer. The `CallSetupRecord` is shared by all the threads involved (i.e., `CallSetupThread` and `InvitePeerThreads`) for communication and synchronization. In the `CallSetupRecord`, the status of each peer is recorded as one of *unknown*, *yes*, or *no* (the status *no* indicates that either the peer rejects the invitation or times out in responding). The call setup succeeds when all peers say yes. The call setup is aborted when any peer says no or an error occurs.

Each `InvitePeerThread` opens a TCP connection to the `RDServer` on one responding peer. In response, the `RDServer` starts a `ClientHandler` thread to talk to the `InvitePeerThread`. (See also Figure 4.3 for the protocol message exchange over this TCP connection.) In the SETUP message, `InvitePeerThread` writes the host name, user name, and application name to the `ClientHandler`. `ClientHandler` pops up a `RingingDialog`, shown in Figure 4.10, to alert the user of an incoming call and writes a RINGING message back to the `InvitePeerThread` to indicate that the called user is being alerted. The user pushes the "Yes" or "No" button to accept or reject

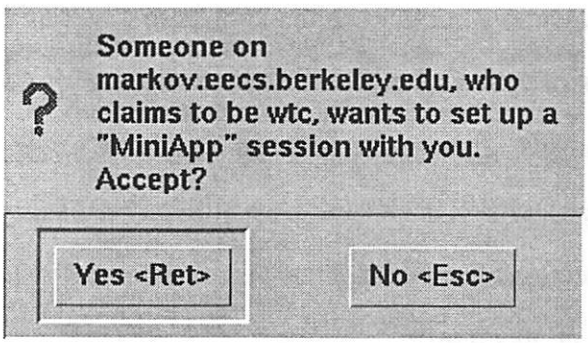


Figure 4.10 The RingingDialog dialog that alerts the user of an incoming call.

the call, and `ClientHandler` writes the user's answer back to `InvitePeerThread`. If the user does not respond within 30 seconds, a TIMEOUT message is written.

When every peer has said yes, the `CallSetupThread` picks a session number (in this implementation, the session number is the current time in milliseconds) and creates the session HTML files. Session HTML files are generated by inserting the following applet parameters into the HTML template files: `orighost`, `session`, and `coordport`. `CallSetupThread` instructs the local Netscape browser to download and display the local session file (and hence execute the embedded applet), and the `InvitePeerThreads` transfer the URLs of the remote session files to the `ClientHandlers` of the responding peers for them to download and display their session files. (The details of downloading the session files and the embedded applets are shown in Figure 4.4.) Then the *session coordination* phase starts, in which the peer applets exchange information, synchronize, and interconnect.

In the common special case of two-party calls, there is only one peer to invite, so a single thread is sufficient. A `TwoPartyCallSetup` thread is started instead.

4.6.4 Session Coordination Procedure

A thread group called the `SessionCoordinator` provides coordination (i.e., synchronization and communication) services to the applets that belong to the same session. The thread group consists of a `SessionCoordDispatcher` thread and some `SessionCoordHandler` threads. The `SessionCoordDispatcher` thread listens on a TCP port (the `coordport`), which is chosen when `RDClient` starts up. It sits in an infinite loop, accepting connections from the applets and dispatching `SessionCoordHandler` threads to handle the connections.

The session coordination protocol allows the peer applets in a session to exchange information and synchronize, *in a single round*, by reading and writing a shared array of

(*name*, *value*) pairs, where *names* and *values* are text strings. Each applet, when connected to the `SessionCoordinator` on the initiating peer (the applet is said to *check in*), first identifies the session number. Then it tells the `SessionCoordHandler` the number of (name, value) pairs it is going to write and proceeds to write those pairs. Then it tells the `SessionCoordHandler` the number of (name, value) pairs it wishes to read and gives the list of names. The `SessionCoordHandler` writes the requested (name, value) pairs to a peer applet as soon as the information becomes available (when written by another peer applet). Session coordination completes when every applet has checked in.

Example: In this example, I explain in detail how the two-party session coordination shown in Figure 4.5 is carried out. The two peer applets in a two-party call can use the session coordination service to establish their first TCP connection as follows. The applet on the initiating peer picks an unused TCP port and listens on the port for connection. The chosen TCP port number, say 3056, is written to the session coordinator as the pair `("listenPort", "3056")`. The applet on the responding peer needs to connect to this TCP port on the initiating peer. It obtains the port number by reading the value for `"listenPort"` from the session coordinator. In addition, when the TCP port number is available, the applet on the responding peer knows that the applet on the initiating peer is ready and listening on the port, so that it can connect to the port. This way, the session coordinator enables the two applets to exchange information (the TCP port number) and synchronize (the responding peer's applet connects after the initiating peer's applet has started to listen). Peer applets in an *n*-party call can use the session coordinator to establish their connections too.

4.6.5 A Peer-to-Peer Application Example

MiniApp is a minimal peer-to-peer application example. The files for MiniApp are placed under a directory named `MiniApp` (see Figure 4.11). The directory `MiniApp` corre-

sponds to the Java package `MiniApp`. Classes in Java are organized into *packages*. Packages give the global class name space a hierarchical structure to avoid name conflicts. Java class bytecode files must be placed in a directory hierarchy that reflects the package hierarchy. The root of this directory hierarchy is called the *codebase*. In the example, the codebase is the directory `/users/wtc/public_html/java/apps`, which is exported by the HTTP server as the URL `http://markov.eecs.berkeley.edu:8080/~wtc/java/apps` (see Figure 4.6). The applications MiniApp, JavaTalk, etc. each have their own directory under the codebase.

Under the directory `MiniApp` are the following files:

- Java source files: `OrigPeerApplet.java` (shown in Figure 4.12) defines the applet for the originating peer, and `RespPeerApplet.java` (shown in Figure 4.13) defines the applet for the responding peer.
- Java class bytecode files: `OrigPeerApplet.class` and `RespPeerApplet.class`, generated by a Java compiler from the above source files.
- Session template files: `origpeer.html` and `resppeer.html`, one for each peer (see Figure 4.14).

MiniApp can be used as a template for constructing more sophisticated applications. Most of the code in MiniApp is generic and can be reused by many applications. In particular,

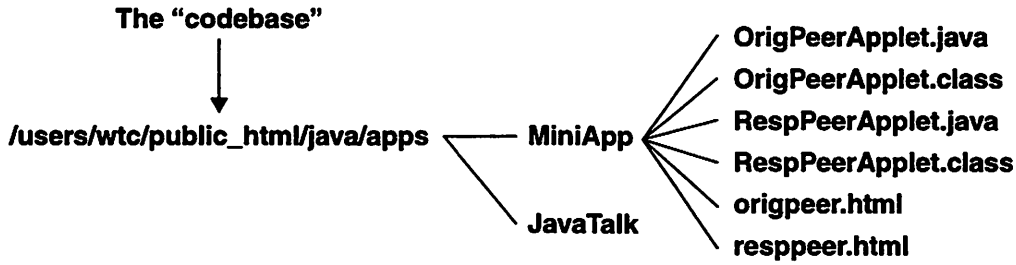


Figure 4.11 Files for the MiniApp application.

```
/*
 * The applet for the originating peer in MiniApp, a minimal
 * peer-to-peer application.
 */

package MiniApp;

import java.applet.Applet;
import java.awt.*;
import java.io.*;
import java.net.*;

public class OrigPeerApplet extends Applet implements Runnable {
    Socket peerSocket;

    TextArea textArea;

    public void init() {
        textArea = new TextArea(5,40);
        add(textArea);

        (new Thread(this)).start();
    }

    void connectToPeer() {
        try {
            // Open a listening server socket.
            ServerSocket serverSocket = new ServerSocket(0);

            // Connect to session coordinator.
            int coordport = Integer.parseInt(getParameter("coordport"));
            Socket socket = new Socket(InetAddress.getLocalHost(), coordport);
            DataOutputStream dos =
                    new DataOutputStream(socket.getOutputStream());

            // Session coordination protocol

            // Send session number.
            long session = Long.parseLong(getParameter("session"));
            dos.writeLong(session);

            // Will write one (name, value) pair, the listening port.
            dos.writeInt(1);
            dos.writeUTF("listenPort");
            dos.writeUTF(Integer.toString(serverSocket.getLocalPort()));

            // Will read nothing
            dos.writeInt(0);
            dos.flush();
```

Figure 4.12 OrigPeerApplet.java (continued on next page).

```
            // Done
            dos.close();
            socket.close();

            // Wait for responding peer's applet to connect.
            peerSocket = serverSocket.accept();

            // Done, no more connections.
            serverSocket.close();
        } catch (NumberFormatException e) {
            System.err.println(e.toString());
        } catch (UnknownHostException e) {
            System.err.println(e.toString());
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }

    public void run() {
        connectToPeer();
        if (peerSocket == null) {
            textArea.appendText("Cannot connect to peer.  Abort.\n");
            return;
        }

        try {
            DataInputStream dis =
                    new DataInputStream(peerSocket.getInputStream());
            DataOutputStream dos =
                    new DataOutputStream(peerSocket.getOutputStream());

            dos.writeUTF("Hello");
            dos.flush();

            String ack = dis.readUTF();
            textArea.appendText("The peer acknowledges, \"" + ack + ".\"\n");

            dis.close();
            dos.close();
            peerSocket.close();
        } catch (EOFException e) {
            System.err.println(e.toString());
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }
}
```

Figure 4.12, continued.

```
/*
 * The applet for the responding peer in MiniApp, a minimal
 * peer-to-peer application.
 */

package MiniApp;

import java.applet.Applet;
import java.awt.*;
import java.io.*;
import java.net.*;

public class RespPeerApplet extends Applet implements Runnable {
    Socket peerSocket = null;

    TextArea textArea;

    public void init() {
        textArea = new TextArea(5, 40);
        add(textArea);

        (new Thread(this)).start();
    }

    void connectToPeer() {
        try {
            // Connect to the session coordinator.
            String orighost  = getParameter("orighost");
            int coordport = Integer.parseInt(getParameter("coordport"));
            Socket socket = new Socket(orighost, coordport);

            DataInputStream dis =
                    new DataInputStream(socket.getInputStream());
            DataOutputStream dos =
                    new DataOutputStream(socket.getOutputStream());

            // Session coordination protocol

            // Send session number.
            long session = Long.parseLong(getParameter("session"));
            dos.writeLong(session);

            // This applet writes nothing and reads one symbol: listenPort.
            dos.writeInt(0);
            dos.writeInt(1);
            dos.writeUTF("listenPort");
            dos.flush();

            String name = dis.readUTF();
            String value = dis.readUTF();
```

Figure 4.13 RespPeerApplet.java (continued on next page).

```
            dis.close();
            dos.close();
            socket.close();

            if (name.equals("ERROR")) {
                System.err.println("Session coordination error: " + value);
                return;
            }

            int port = Integer.parseInt(value);
            peerSocket = new Socket(orighost, port);
        } catch (NumberFormatException e) {
            System.err.println(e.toString());
        } catch (UnknownHostException e) {
            System.err.println(e.toString());
        } catch (EOFException e) {
            System.err.println(e.toString());
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }

    public void run() {
        connectToPeer();
        if (peerSocket == null) {
            textArea.appendText("Cannot connect to peer. Session aborted.\n");
            return;
        }

        try {
            DataInputStream dis =
                    new DataInputStream(peerSocket.getInputStream());
            DataOutputStream dos =
                    new DataOutputStream(peerSocket.getOutputStream());

            // Wait for message from peer.
            String message = dis.readUTF();
            textArea.appendText("The peer says, \"" + message + ".\"\n");

            // Write acknowledgment message to peer.
            dos.writeUTF(message + " back at you");

            dis.close();
            dos.close();
            peerSocket.close();
        } catch (EOFException e) {
            System.err.println(e.toString());
        } catch (IOException e) {
            System.err.println(e.toString());
        }
    }
}
```

Figure 4.13, continued.

```
<!-- origpeer.html -->

<html>
<head>
<title>MiniApp: A Minimal Peer-to-Peer Application</title>
</head>

<body>
<h1>MiniApp: A Minimal Peer-to-Peer Application</h1>

This page contains the applet for the originating peer in
a minimal peer-to-peer application.
<p>

<applet codebase=.. code=MiniApp.OrigPeerApplet.class width=400 height=150>
</applet>

</body>
</html>
```

(a) origpeer.html

```
<!-- resppeer.html -->

<html>
<head>
<title>MiniApp: A Minimal Peer-to-Peer Application</title>
</head>

<body>
<h1>MiniApp: A Minimal Peer-to-Peer Application</h1>

This page contains the applet for the responding peer in
a minimal peer-to-peer application.
<p>

<applet codebase=.. code=MiniApp.RespPeerApplet.class width=400 height=150>
</applet>

</body>
</html>
```

(b) resppeer.html

Figure 4.14 The session file templates for MiniApp.

the `connectToPeer()` methods of the two peer applets (`OrigPeerApplet` and `RespPeerApplet`) perform the common session coordination procedure to establish a connection, as described in the example on page 95.

In the following, please refer to the Java source code in Figure 4.12 and Figure 4.13. Both peer applets are derived from the base class `Applet` and implement the `Runnable` interface. (A class that implements the `Runnable` interface has a `run()` method, which can be the main body of a thread.) When the applets are loaded and started, their `init()` methods are invoked by the Web browsers. In MiniApp, the `init()` methods of the two peer applets are the same. They first create the applets' graphical user interface, in this case just a text area with 5 rows and 40 columns. Then the `init()` methods each start a thread to execute the `run()` method of the applet. The new thread will carry out the session coordination procedure. Note that the session coordination procedure must be performed by a new thread. Performing the possibly blocking session coordination in the `init()` method, which is invoked by a thread of the Web browser, would prevent the browser's thread from rendering the applet's graphical user interface quickly.

In the `run()` method, each thread first invokes the `connectToPeer()` method to perform session coordination. The `connectToPeer()` method of the `OrigPeerApplet` opens a listening socket and then connects to the session coordinator, whose port number is passed in as the applet parameter `coordport`. `OrigPeerApplet` sends to the session coordinator the port number it is listening on as the value of the symbol named `"listenPort"`. Then it waits for the `RespPeerApplet` to connect to this port.

The `connectToPeer()` method of the `RespPeerApplet` also connects to the session coordinator, whose host name and port number are passed in as the applet parameters `orighost` and `coordport`. `RespPeerApplet` asks for the value of the symbol named `"listenPort"`, and then connects to that port. At this point, the connection

between the peer applets has been established. This connection can be accessed using the applets' class member `PeerSocket`.

In the rest of the `run()` method, the peer applets can use their `PeerSocket` member to implement the application functionality. In MiniApp, the peer applets simply do one round of message exchange and close the connection. `OrigPeerApplet` sends a message "Hello" to `RespPeerApplet`. When `RespPeerApplet` receives the message, it sends back an acknowledgment, formed by appending "back at you" to the message. Each applet displays the message it receives from the peer in its text area.

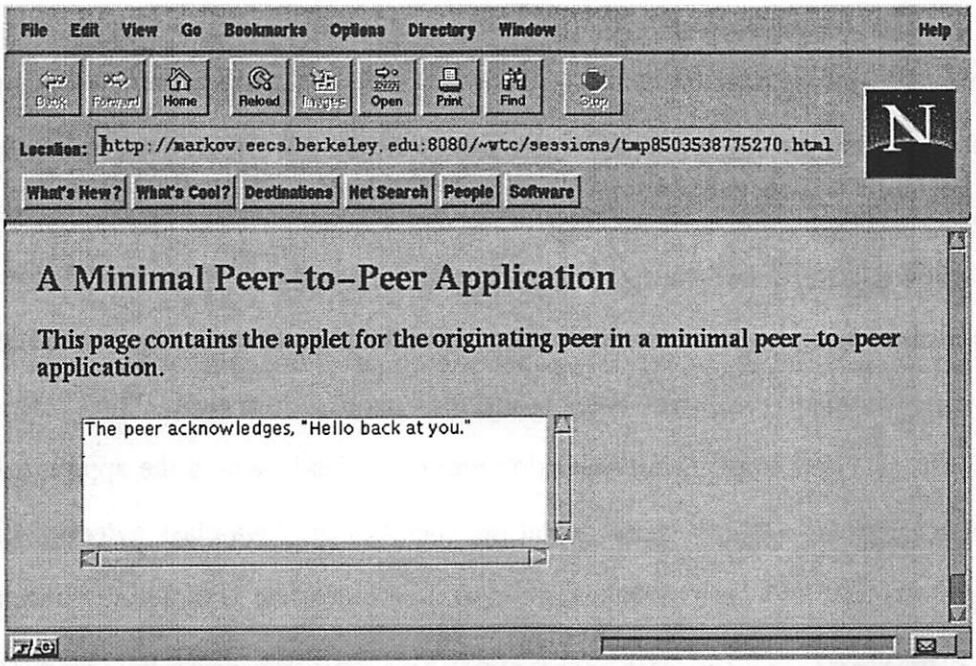
Finally, one has to prepare the session template files in which the applets are embedded (see Figure 4.14). The `code` applet tag specifies the Java class bytecode file for the applet. Because the peer applets are in the package `MiniApp`, their full class names are `MiniApp.OrigPeerApplet` and `MiniApp.RespPeerApplet`. The `codebase` applet tag is the URL for the *root of the package hierarchy.* The `codebase` URL can be given relative to the directory of the session template file. Because the session template files are in the directory `MiniApp`, the root of the package hierarchy is one level up, hence the relative URL ".." (see Figure 4.11). Note that the equivalent absolute URL `http://markov.eecs.berkeley.edu:8080/~wtc/java/apps/` could have been used, but a relative URL allows us to move the `MiniApp` directory to a different codebase without having to modify the `codebase` applet tag in the session template files.

The screen shots of the peer applets are shown in Figure 4.15.

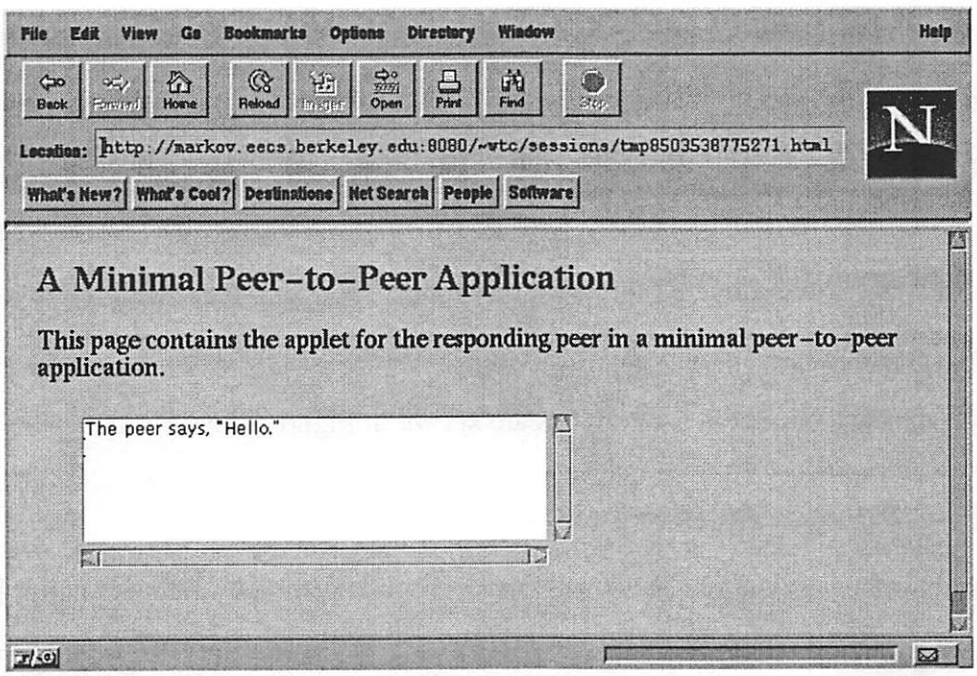
4.6.6 Portability Issues

Although written in the portable Java language, the prototype implementation has some code specific to Unix and X Windows and hence only runs on the Unix/X platform.

The two helper programs `RDClient` and `RDServer` should ideally be integrated into the WWW browsers as local Java applets. In my prototype implementation,



(a) The originating peer



(b) The responding peer

Figure 4.15 Screen shots of the MiniApp peer applets.

`RDServer` does run as a local applet in the remote peer's Netscape browser. However, `RDClient` cannot run as a local applet. This is because `RDClient` needs to read a startup file and create session files, but Netscape does not allow even local applets to access the file system. Therefore, `RDClient` has to run as a stand-alone Java program.

On the Unix/X platform, `RDClient` can control its associated Netscape browser using the `netscape -remote` commands. For example, `RDClient` tells its companion Netscape browser to display the local session file using the `netscape -remote openURL()` command. The `netscape -remote` commands are specific to the Unix/X platform. Unfortunately, there are no equivalent remote control mechanisms for Netscape browsers on PC/Windows and Mac platforms. Therefore, currently `RDClient` runs only on Unix/X. However, `RDServer` and our example applications, all running as applets within Web browsers, have been tested to run on Unix/X and PC/Windows platforms.

When the *signed applets* feature is introduced in the future, Java applets that are digitally signed by trusted parties can be granted more file system access privileges. At that time, we will be able to run `RDClient` as an applet within the originating peer's Web browser.

4.6.7 Possible Extensions

The following are some possible extensions to the prototype implementation:

- Strengthen the security of the session establishment and coordination procedure (see the footnote on page 84).
- Java class libraries for peer applets: Common connection establishment procedures, such as the one described in the example in Section 4.6.4, and common communication primitives should be made available as class libraries for peer applets. We can borrow general solutions to problems in distributed computing, e.g., the simulation of

shared memory [160], but for most networked applications, simpler solutions would suffice. For example, a fault-tolerant algorithm may not be necessary for a shared whiteboard application.

- Multi-user applications: Because of the *applet-host* security policy in Netscape Navigator, a multi-user application must be implemented in a star configuration, i.e., the applet of every responding peer connects back to the initiating peer's applet. Note that the call setup and session coordination protocols in the prototype are general. They work for multi-user applications as well as two-user applications, and in fact are not restricted to those implemented in a star configuration.

4.7 A Security Model

Netscape's applet security policy is rather restrictive for good reasons. However, it also seriously limits the useful functions that an applet can do. In this section, I propose a hypothetical security model that gives sufficient flexibility to Java applets while maintaining safety.

4.7.1 Hypotheses

The security model has the following hypotheses:

1. Applets can be digitally signed.
2. Applets' access to system resources (e.g., CPU time, file system, and networking), host environment (e.g., a Web browser), and other applets can be controlled by a *security policy.*
3. The identity of the communication peers (e.g., the applet repositories, the users, and the terminals) can be authenticated.

4. Communication can be encrypted.

Hypothesis 1 (signed applets) enables us to know who wrote or have examined the applets and to have certain trust on reasonable behavior from the applets. Hypothesis 2 is similar to the protection offered by an operating system: user processes are isolated from the kernel and other user processes, and user processes access system resources by invoking *system calls* into the kernel. Hypothesis 2 is important not only for protection against untrusted applets but also for fair allocation of resources and protection against bugs in trusted applets.

Hypotheses 3 (authentication) and 4 (encrypted communication) are applicable to the security of communication in general, not just the situations involving executing remote code. The Internet, which started as a network for collaborating researchers, has largely ignored such security issues in its early design. Recently, authentication protocols such as Kerberos [161] and encrypted communication protocols such as the Secure Socket Layer (SSL) [162] have been introduced to enhance the security of Internet.

4.7.2 Security Policies

For networked applications, the applet security policy should be a function of the trust in both the *downloaded applet* and the *communication peer.* When applets are digitally signed by a trusted party and the identity of the peer can be authenticated, one can grant more access privileges to the applets. Here I consider the case where neither the applet nor the communication peer can be trusted.

For peer-to-peer applications, the applet-host security policy must be modified to allow network connections between the communication peers. The applet-host security policy is based on the assumption that the applet host coincides with one of the communication peers, which is true in most client-server applications. In peer-to-peer applications, however, this assumption breaks down in the third-party applet repository scenario (see

Figure 4.1) and rules out this useful possibility. Therefore, the security policy on applets' networking capabilities should be based on *communication peers*, not the *applet host*.

When applets are granted networking privileges, the major security risks are *information leakage* and *imposture*. These two risks can be dealt with as follows (see Figure 4.16).

1. Information leakage: Given that there exists a network connection to a remote peer, the applet should not be allowed to read arbitrary files or access any other host within the firewall. The applet's user interface should be blatantly marked as untrusted, so that the user is alerted when asked to enter any information. Then the applet is prevented from extracting sensitive information and leaking it through the network connection

The applet may be granted access to a restricted, isolated portion of the local file system. For example, an applet should be allowed to save the result of a whiteboard or *talk* session in a file under a directory designated for the applet.

2. Imposture: When communicating with other hosts on the network, the applet assumes the identity of the local host and user. Therefore, the applet should not be allowed to communicate with an arbitrary host or service, even those outside the firewall. This includes sending e-mail or posting newsgroup articles. However, the applet may be allowed to com-

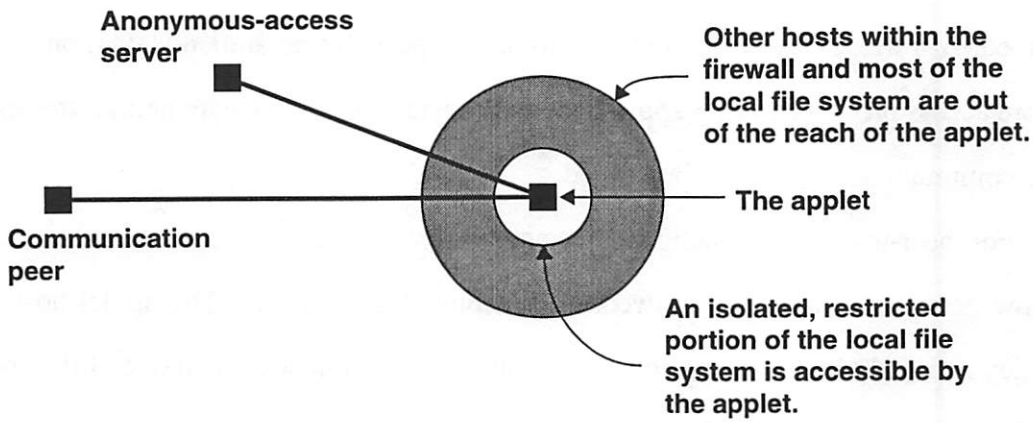


Figure 4.16 Security of an applet's networking privileges.

municate with certain well-known hosts or services that allow anonymous access, such as anonymous ftp and Web servers.[1]

4.8 Example: Distributed CAD Environment

In Figure 4.17 I show a typical collaborative application in a computer-aided design (CAD) environment as a peer-to-peer application example. Suppose two designers are working together on a design. A collaborative application that helps the designers to communicate ideas and exchange information might include the following components:

- Voice and video conferencing to let the designers see and talk to each other.
- Remote-controlled, coordinated WWW browsing (i.e., one person can control the other people's WWW browsers so that their browsers are displaying the same docu-

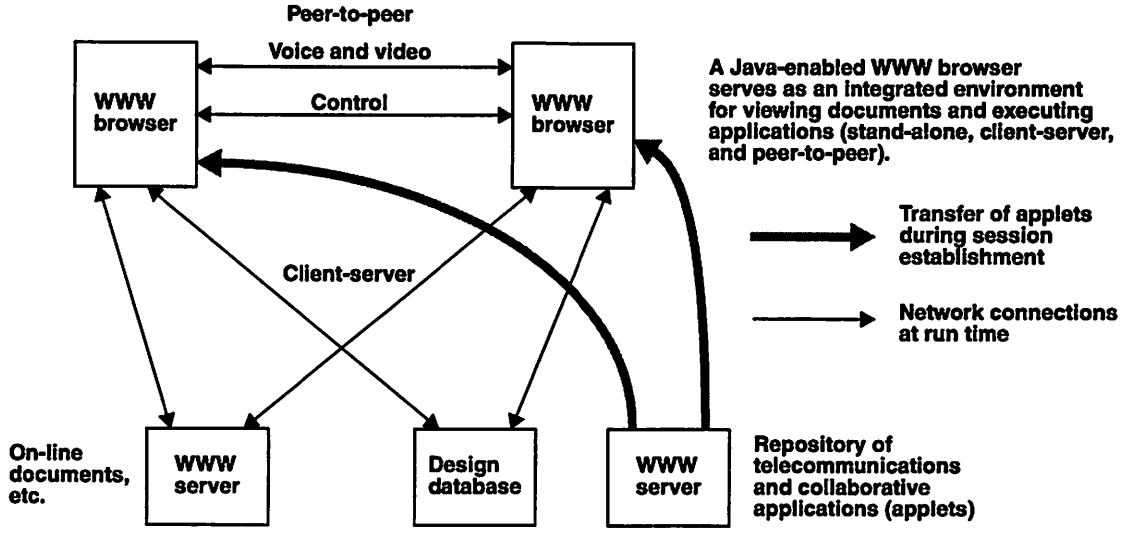


Figure 4.17 An application of our technique to a typical collaborative design scenario in CAD.

1. Some Web servers, such as httpd from NCSA, grant access to restricted files based solely on the network address of the client. This practice is insecure, similar to the infamous .rhosts file on Unix workstations. A malicious applet may exploit this feature and gain access to restricted files.

ments) to let them jointly view documents. A couple of *chat* applications, e.g., Netscape Chat of Netscape, already combine IRC (Internet Relay Chat, a text-based communication application [6]) and remote-controlled WWW browsing.

- A shared whiteboard on which the designers can draw sketches of design ideas.
- Shared editing of a document, schematic, or design specification.

In general, such an application would be a mixture of graphics and continuous-media peer-to-peer communications, remote-controlled, coordinated client-server access to multimedia databases, and shared peer-to-peer applications. In my approach, a user can use a standard WWW browser as a framework for participating in any collaborative design context, be it client-server or peer-to-peer.

Every stand-alone application such as a word processor or design editor has a natural peer-to-peer shared version, which would provide a mirrored view and remote control of the shared object (document, design, etc.) to the peer. Because my technique allows standard WWW software environments to support peer-to-peer applications, it should attract more energy and talent to the creation of new, compelling peer-to-peer applications.

4.9 Conclusions

In this chapter, I have explained the general dynamic deployment approach for networked applications implemented in the peer-to-peer architecture, and described the design of a prototype implementation based on the WWW and Java framework. The prototype consists of two helper programs that run in conjunction with the Netscape browsers and communicate using a coordination protocol, and is written entirely in Java for portability. However, because of the constraints imposed by Netscape's security policy on Java applets' access to the file system, the originating peer's helper program, `RDClient`, currently only runs on the Unix/X platform. As soon as Netscape gives file system access to a

trusted applet, such as one that is digitally signed by a trusted party, `RDClient` can be made to run on other platforms. The responding peer's helper program, `RDServer`, has been tested on both Unix/X and PC/Windows platforms.

Currently we have two applications that can be deployed on this infrastructure: a text-based peer-to-peer communication application (*JavaTalk*) similar to the Unix *talk* program and a shared whiteboard application. New peer-to-peer applications can be developed to be deployed and executed on this infrastructure. Because Java is platform-independent, any peer-to-peer Java applets written to work with our coordination protocol should run transparently across all platforms.

I proposed a security model that takes into account of the needs and risks of peer-to-peer applications. In general, the security policy should depend on the trust in both the downloaded applet and the communication peer. The networking privileges should be granted based on the communication peer, not the applet host. For the common case where there is no trust in either the applet or the communication peer, I outlined a security policy regarding networking and file system access privileges that enables the applet to perform useful functions while not compromising security.

The prototype is dubbed *Applet-to-Share*, because it allows people to share Java applets. Source code and documentation of Applet-to-Share are available on the Web at the URL http://ptolemy.eecs.berkeley.edu/dgm/javatools/applet-to-share/index.html.

Dynamic deployment avoids a major obstacle to user-to-user applications, the requirement that users have previously obtained and installed the necessary interoperable application software, and thus will encourage a proliferation of innovative user-to-user applications. By adding two helper programs and a coordination protocol to Java and HTTP, I have demonstrated dynamically deployed peer-to-peer applications using two instances of existing Java-enabled WWW browsers. Thus, I have demonstrated that peer-to-peer applications can run in a standard widely-deployed commercial software environ-

ment originally defined for client-server applications. The community of users equipped with Java-enabled WWW browsers become candidates for any Java-described peer-to-peer applications, bypassing the current obstacle that the users have to have anticipated the application and obtained and installed the required application software. This dynamic deployment approach also ensures interoperability between peer programs. I believe the dynamic deployment approach has the potential to make the peer-to-peer application space as rich and dynamic as the current client-server application space.

5

Conclusions

5.1 Conclusions

The goal of this dissertation is to increase the diversity of networked applications, in particular user-to-user multimedia applications implemented in the peer-to-peer architecture. I have studied this problem in both the design and deployment phases. I have proposed a design methodology tailored to the characteristics of networked multimedia applications, and a dynamic deployment approach that extends the current use of Java applets in the World Wide Web to user-to-user applications in the peer-to-peer architecture.

5.1.1 Heterogeneous Design Methodology

The design methodology I proposed takes the heterogeneous approach of the Ptolemy design environment — using domain-specific models of computation, such as dataflow and finite state machines (FSMs), to design different portions of the system, and combining them in a precise and unambiguous manner. The models of computation are overlaid on top of general programming languages. The models of computation describe the high-level skeleton of an application, while the general programming languages fill in the low-level details. The models of computation serve two purposes. First, they provide special-

ized constructs that are not available in general programming languages, yet useful and suitable for particular application domains. Second, their semantics are usually small, so that some verification problems at the higher abstraction level of these models remain decidable, and crucial properties can be checked at compile time.

Networked multimedia applications are reactive systems that combine signal processing with sophisticated control, therefore the design methodology particularly addresses the characteristics of the mixture of numeric computation and intricate control flow. To complement the more mature dataflow models in the Ptolemy design environment, which are suitable for designing signal processing subsystems, I have focused on the models of computation for describing complex control functionality and their interfaces to dataflow and other concurrency models.

The guidelines I have followed when developing the models of computation for control are that the models should be based on the intuitive notions of states and events, and that they should have constructs for handling large, complex controllers. After studying past work in this area, most notably Statecharts and its variants, I recognized that the three components of Statecharts, i.e., hierarchy, FSM, and concurrency, can in fact be cleanly separated if the noncompositional constructs like inter-level transitions are removed from Statecharts. I showed that these noncompositional constructs are either not useful or can be easily emulated. In return for their removal, we obtain a flexible scheme to nest FSM hierarchically with concurrency models. The mix-and-match of FSM, hierarchy, and concurrency can capture the essential features of many Statecharts variants. I call this scheme *charts, which can be viewed as a generalization of Statecharts. Then I proposed a hierarchical FSM model that can be mixed with various concurrency models.

The hierarchical mixing of FSM and concurrency models has been implemented in the Ptolemy design environment for the synchronous dataflow (SDF) domain. I described the implementation strategy and illustrated it with a digital watch programming example.

5.1.2 Dynamic Network Deployment

The relatively small number of existing user-to-user networked applications is mainly due to the network externality problem faced by new applications of this class in their deployment. This intrinsic deployment obstacle makes user-to-user applications unattractive to service providers and application developers. Service providers hesitate to deploy new user-to-user applications, and application developers are reluctant to devote their effort to creating compelling user-to-user applications.

To free end users from having to rely on a service provider's decision to offer an application, user-to-user applications can be implemented in the peer-to-peer architecture, which does not require centralized servers. To alleviate the effect of network externality, I have proposed a dynamic network deployment approach to distribute applications to the hands of end users quickly.

I have designed an infrastructure for the dynamic deployment of peer-to-peer applications based on the WWW/Java framework, and implemented a prototype, consisting of two helper programs that run in conjunction with a Java-enabled WWW browser (such as Netscape Navigator) and communicate using a session establishment protocol. The design uses standard, widely available software components, and the additional code is written in Java. There is also a flexible session coordination protocol that assists peer applets within the same session to exchange information and connect to each other during establishment. The session establishment and coordination protocols are general enough for setting up multi-user as well as two-user networked applications.

The current prototype of dynamic deployment does not implement the case where application descriptions are stored in some central repository because of the restrictive applet security policy enforced by Netscape Navigator. I proposed a security model that would allow a full implementation of dynamic deployment.

The deployment infrastructure that I designed turns a Java-enabled WWW browser into an integrated execution environment for dynamically deployed peer-to-peer as well as client-server applications. Dynamic deployment avoids the requirement that the users have previously obtained and installed the necessary interoperable application software, and thus will encourage a proliferation of innovative user-to-user applications.

5.2 Open Issues

The design and deployment of networked applications are broad problems. Throughout this dissertation I have pointed out the issues one can pursue based on my work. For example, Chapter 1 contains an overview of the issues in designing and deploying a networked application that can be set up and executed, with low cost and high subjective quality, under heterogeneous and dynamic network and user terminal conditions. In Chapter 2 and Chapter 4, I have listed possible extensions to the proposed approaches and implementation.

The integration of the design methodology and dynamic deployment approach gives the most leverage, and we have previously demonstrated this using Ptolemy as the design and run-time environment and the Ptolemy interpreter language, based on Tcl, as the application description language. The current dynamic deployment infrastructure uses the more widely disseminated WWW and Java. At the present time, the Ptolemy group is devoting significant effort to developing the Java code generation capabilities in Ptolemy. When that is finished, applications developed using the models of computation in Ptolemy will be able to be translated into Java code, and readily deployed over the dynamic deployment infrastructure.

For dynamic deployment, an important research issue is a security model that allows dynamically deployed code to provide useful functions while not compromising the secu-

rity. Netscape's applet security policy is very restrictive on what an applet is allowed to do. Ousterhout *et al.* have proposed a nice security model for Safe-Tcl, which cleanly isolates the remote applets and the local host execution environment in multiple Tcl interpreters [72]. It would be interesting to investigate whether Java can emulate Safe-Tcl's security model.

In the rest of this section, I would like to touch on two performance-related issues that are critical to the success of the dynamic deployment approach, using Java applets as an example, and speculate on possible solutions.

5.2.1 Downloading Time

The time it takes to download Java applets contributes to the session establishment delay, and should be kept as short as possible. Broadband networking is the best solution. In addition, some possible techniques to minimize the download time are:

- Avoid downloading: Java applets downloaded from the network can be cached or saved permanently in the local storage. A related issue is the versioning of Java classes, which can be used to determine whether a new copy of a locally cached Java class needs to be downloaded.
- Download efficiently: All the Java classes and data used by a Java applet can be packaged into one archive file, such as the Java Archive (JAR) format that Sun is defining, and downloaded in one network transaction to avoid the round-trip delays from downloading each of them separately. The archive file can be compressed for better bandwidth efficiency.
- Hide the latency: A Java applet can be structured so that a minimal subset of the Java classes that implements its core functionality can be downloaded first and started immediately, while the rest of the classes are being downloaded in the background.

Additionally, some optional Java classes may be downloaded only when they are needed.

5.2.2 Performance of Java for Multimedia Processing

Java source code is compiled into Java bytecode, the instructions for the Java virtual machine. The virtual machine layer provides security protection and achieves platform independence, at the cost of degrading the execution performance. Java bytecode interpretation is about 10-15 times slower than compiled C code. Although microprocessors are becoming faster by Moore's Law, there will always be applications that need every bit of performance from the processor. Most notable are multimedia applications. High fidelity audio, video, graphics, and animation all require heavy-duty numeric computation, typically repetitive operations on streams of data at high rate.

Some possible techniques to enhance Java's run-time performance are:

- Just-in-time bytecode compilers: Some Java virtual machines contain a just-in-time (JIT) compiler, which translates Java bytecode into native code and caches the native code while the Java program is executing. JIT bytecode compiler can enhance run-time performance to what is comparable to unoptimized C code. According to a benchmark test performed by Pendragon Software Corporation [163], JIT compilers are especially good at optimizing the mathematical operations, achieving a speedup factor of 40-80. "This acceleration will make computationally intensive animation and 3D modeling much faster." The findings of this performance report are encouraging.
- Hardware implementation of the Java virtual machines: These dedicated Java processors, such as Sun's Java chips [164], natively understand Java bytecode without the overhead of a bytecode interpreter or JIT compiler. This approach is mainly useful for embedded Java applications, such as cellular phones, TV set-top boxes, and network

computers. For general-purpose computers, their powerful CPUs with JIT compilers are likely to offer sufficient performance without the cost of adding a Java co-processor.

- Native multimedia instructions: Recently microprocessors have begun to have specialized instructions for common multimedia processing operations, e.g., the VIS instructions in Sun's UltraSPARC processors and the MMX instructions in Intel's x86 processors [165][166]. These SIMD (single instruction, multiple data) instructions operate on multiple integer operands packed into a register in parallel. Java's run-time system should take advantage of these specialized instructions. Java interpreters and JIT compilers may not be smart enough to exploit the specialized instructions. A possible approach is to define a standard multimedia class library, with native code implementation, of the commonly used components in signal processing.

Bibliography

[1] J. Grudin, "On Computer Supported Collaborative Work," *Collaborative Computing, Communications of the ACM*, Vol. 34, No. 12, 1991.

[2] J. Grudin, "Computer-Supported Cooperative Work: History and Focus," *Computer*, Vol. 27, No. 5, pp. 19-26, May 1994.

[3] T. Berners-Lee, A. Caillau, A. Loutonen, H. F. Nielsen, and A. Secret, "The World Wide Web," *Communications of the ACM*, Vol. 37, pp. 76-82, August 1994.

[4] H. L. Berghel, "The Client Side of the Web," *Communications of the ACM*, Vol. 39, pp. 33-40, January 1996.

[5] M. Handley and J. Crowcroft, *The World Wide Web — Beneath the Surf*, UCL Press, London, 1994.

[6] S. Harris, *The IRC Survival Guide: Talk to the World with Internet Relay Chat*, Addison-Wesley, Reading, Mass., 1995.

[7] C. A. Ellis, S. J. Giggs, and G. L. Rein, "Groupware: Some Issues and Experiences," *Communications of the ACM*, Vol. 34, No. 1, January 1991.

[8] A. S. Tanenbaum, "The Application Layer," Chapter 7 in *Computer Networks*, Third Edition, Prentice Hall PTR, Upper Saddle River, New Jersey 07458, 1996.

[9] D. G. Messerschmitt, "The Convergence of Telecommunications and Computing: What are the Implications Today?", *Proceedings of the IEEE*, Vol. 84, No. 8, pp. 1167-1186, August 1996.

[10] D. G. Messerschmitt, "The Future of Computer Telecommunications Integration," *IEEE Communications Magazine*, Vol. 34, No. 4, pp. 66-69, April 1996.

[11] F. Fluckiger, *Understanding Networked Multimedia: Applications and Technology*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[12] National Research Council, Computer Science and Telecommunications Board, *Realizing the Information Future: The Internet and Beyond*, National Academy Press, Washington D.C., 1994.

[13] D. E. Comer, *Internetworking with TCP/IP*, Vol. 1, 3rd Ed., Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[14] W. R. Stevens, *TCP/IP Illustrated*, Vol. 1, Addison-Wesley, Reading, Massachusetts, 1994.

[15] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications," RFC 1889, November 1995.

[16] P. A. Bernstein, "Middleware: A Model for Distributed System Services," *Communications of the ACM*, Vol. 39, No. 2, pp. 86-98, February 1996.

[17] M. De Prycker, *Asynchronous Transfer Mode: Solution for Broadband ISDN*, Second Edition, Ellis Horwood, New York, 1993.

[18] J.-Y. Le Boudec, "The Asynchronous Transfer Mode: A Tutorial," *Computer Networks and ISDN Systems*, Vol. 24, pp. 279-309, May 1992.

[19] K.-Y. Siu and R. Jain, "A Brief Overview of ATM: Protocol Layers, LAN Emulation, and Traffic Management," *Computer Communication Review*, Vol. 25, pp. 6-20, April 1995.

[20] D. C. Cox, "Wireless Network Access for Personal Communications," *IEEE Communications Magazine*, Vol. 30, No. 12, pp. 96-115, December 1992.

[21] D. J. Goodman, "Trends in Cellular and Cordless Communications," *IEEE Communications Magazine*, Vol. 29, No. 6, pp. 31-40, June 1991.

[22] J. E. Padget, C. G. Gunther, and T. Hattori, "Overview of Wireless Personal Communications," *IEEE Communications Magazine*, Vol. 33, No. 1, pp. 28-41, January 1995.

[23] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, Vol. 19, No. 7, pp. 395-404, July 1976.

[24] A. Sinha, "Client-Server Computing: Current Technology Review," *Communications of the ACM*, Vol. 35, No. 7, pp. 77-98, July 1992.

[25] S. Broadhead, "Client-Server: The Past, Present and Future," *Network Computing*, Vol. 4, No. 12, pp. 38, 40, 42-43, December 1995.

[26] *Cornell University's CU-SeeMe Page*. (URL http://cu-seeme.cornell.edu/)

[27] T. Dorcey, "The CU-SeeMe Desktop Videoconferencing Software," *ConneXions*, Vol. 9, No. 3, pp. 42-45, March 1995.

[28] M. Sattler, *Internet TV with CU-SeeMe*, Sams net, Indianapolis, Indiana, 1995.

[29] P. E. Haskell, *Flexibility in the Interactions Between High-Speed Networks and Communications Applications,* Ph.D. Dissertation, University of California at Berkeley, December 1993.

[30] L. C. Yun and D. G. Messerschmitt, "Digital Video in a Fading Interference Wireless Environment," *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, Atlanta, GA, May 1996.

[31] D. G. Messerschmitt, "Complexity Management: A Major Issue for Telecommunications," *International Conference on Communications, Computing, Control, and Signal Processing*, Stanford University, Palo Alto, CA, June 22-16, 1996.

[32] J. C. McDonald, *Fundamentals of Digital Switching*, Second Edition, Plenum Press, 1990.

[33] GRINSEC, *Electronic Switching*, North-Holland, 1983.

[34] A. Cambell, G. Coulson, F. Garcia, and D. Hutchison, "A Continuous Media Transport and Orchestration Service," *Proceedings of ACM SIGCOMM '92 Conference. Communications Applications, Architectures and Protocols*, Baltimore, MD, USA, August 17-20, 1992.

[35] B. Wolfinger and M. Moran, "A Continuous Media Data Transport Service and Protocol for Real-Time Communication in High Speed Networks," *Network and Operating System Support for Digital Audio and Video. Second International Workshop Proceedings*, Heidelberg, Germany, November 18-19, 1991. Edited by: R. G. Herrtwich, pp. 171-182, Springer-Verlag, Berlin, Germany, 1992.

[36] A. Cambell, G. Coulson, and D. Hutchison, "A Quality of Service Architecture," *Computer Communication Review*, Vol. 24, pp. 6-27, April 1994.

[37] C. Aurrecoechea, A. Campbell, and L. Hauw, "A Review of QoS Architectures," to appear in *Multimedia Systems Journal*, 1996, and invited paper in *Proc. 4th IFIP International Workshop on Quality of Service*, Paris, March 1996.

[38] J. S. Turner, "New Directions in Communications (or Which Way to the Information Age?)," *IEEE Communications Magazine*, Vol. 24, No. 10, pp. 8-15, October 1986.

[39] A. K. Parekh and R. G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Network – the Single-Node Case," *IEEE/ACM Transactions on Networking*, Vol. 1, No. 3, pp. 344-357, June 1993.

[40] R. Han and D. G. Messerschmitt, "Asymptotically Reliable Transport of Multimedia/Graphics Over Wireless Channels," *Proc. Multimedia Computing and Networking*, San Jose, CA, January 29-31, 1996.

[41] A. Y. Lao, J. M. Reason, and D. G. Messerschmitt, "Asynchronous Video Coding for Wireless Transport," *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, Dec. 1994.

[42] J. M. Reason, L. C. Yun, A. Y. Lao and D. G. Messerschmitt, "Asynchronous Video: Coordinated Video Coding and Transport for Heterogeneous Networks with Wireless Access," chapter in *Mobile Computing*, H. F. Korth and T. Imielinski, Eds., Kluwer Academic Press, Boston, MA, 1995.

[43] P. E. Haskell and D. G. Messerschmitt, "In Favor of an Enhanced Network Interface for Multimedia Services," submitted to *IEEE Multimedia Magazine,* 1995.

[44] C. Huitema, *IPv6: The New Internet Protocol*, Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[45] S. A. Thomas, *IPng and the TCP/IP Protocols: Implementing the Next Generation Internet*, John Wiley & Sons, Inc., 1996.

[46] E. Lyghounis, I. Poretti, and G. Monti, "Speech Interpolation in Digital Transmission Systems," *IEEE Transactions on Communications*, Vol. COM-22, No. 9, pp. 1179-1189, September 1974.

[47] D. S. Taubman and A. Zakhor, "Rate and Resolution Scalable Subband Coding of Video," *1994 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, Adelaide, SA, Australia, April 19-22, 1994.

[48] D. S. Taubman, *Directionality and Scalability in Image and Video Compression*, Ph.D. Dissertation, University of California at Berkeley, 1994.

[49] S. McCanne, *Joint Source/Channel Coding for Multicast Packet Video*, Ph.D. dissertation, Computer Science Division, University of California at Berkeley, summer 1996.

[50] A. A. Lazar and G. Pacifici, "Control of Resources in Broadband Networks with Quality of Service Guarantees," *IEEE Communications Magazine*, Vol. 29, No. 10, pp. 66-73, October 1991.

[51] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," *Internetworking: Research and Experience*, Vol. 1, No. 1, pp. 3-26, September 1990.

[52] A. G. Greenberg and N. Madras, "How Fair is Fair Queueing?", *Journal of the Association for Computing Machinery*, Vol. 39, No. 3, pp. 568-598, July 1992.

[53] S. Minzer, "A Signaling Protocol for Complex Multimedia Services," *IEEE Journal on Selected Areas in Communications*, Vol. 9, No. 9, pp. 1383-1394, December 1991.

[54] J. R. Cox, Jr., M. E. Gaddis, and J. S. Turner, "Project Zeus," *IEEE Network*, Vol. 7, No. 2, March 1993.

[55] M. Wakamoto, M. W. Kim, K. Fukuda, and K. Murakami, "A Communication Network Control Architecture to Integrate Service Control and Management," *IEICE Trans. Commun.*, Vol. E77-B, No. 11, pp. 1342-1349, November 1994.

[56] R. Cohen and Y.-H. Chang, "Video-on-Demand Session Management," *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 6, pp. 1151-1161, August 1996.

[57] P. Moghé and I. Rubin, "Enhanced Call: A Paradigm for Applications With Dynamic Client-Membership and Client-Level Binding in ATM Networks," *IEEE/ACM Transactions on Networking*, Vol. 4, No. 4, pp. 615-628, August 1996.

[58] D. Ferrari and D. C. Verma, "A Scheme for Real-Time Channel Establishment in Wide-Area Networks," *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 3, pp. 368-379, April 1990.

[59] T. Nishida and K. Taniguchi, "QOS Controls and Service Models in the Internet," *IEICE Trans. Commun.*, Vol. E78-E, No. 4, pp. 447-457, April 1995.

[60] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala, "RSVP: A New Resource ReSerVation Protocol," *IEEE Network*, Vol. 7, No. 5, pp. 8-18, September/October 1993.

[61] M. E. Lukacs and D. G. Boyer, "A Universal Broadband Multipoint Teleconferencing Service for the 21st Century," *IEEE Communications Magazine*, Vol. 33, No. 11, pp. 36-43, November 1995.

[62] A. Fox and E. Brewer, "Reducing WWW Latency and Bandwidth Requirements via Real-Time Distillation," *Proceedings of the Fifth International World Wide Web Conference (WWW-5)*, Paris, France, May 1996.

[63] S.-F. Chang, *Compositing and Manipulation of Video Signals for Multimedia Network Video Services*, Ph.D. Dissertation, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1993.

[64] M. J. Woodridge and N. R. Jennings, Eds., *Intelligent Agents. Proceedings of ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, Amsterdam, The Netherlands, Springer-Verlag, August 1994.

[65] M. J. Woodridge and N. R. Jennings, "Agent Theories, Architectures, and Languages: A Survey," *ECAI-94 Workshop on Agent Theories, Architectures, and Languages*, Amsterdam, The Netherlands, August 8-9, 1994.

[66] D. Chess *et al.*, *Itinerant Agents for Mobile Computing*, Research Report RC 20010, IBM Research Division, Yorktown Heights, N. Y., 1995.

[67] L. F. Bic, M. Fukuda, and M. B. Dillencourt, "Distributed Computing Using Autonomous Objects," *Computer*, Vol. 29, No. 8, pp. 55-61, August 1996.

[68] J. E. White, *Telescript Technology*, Tech. Report, General Magic, Mountain View, California, 1994.

[69] J. Tardo and L. Valente, "Mobile Agent Security and Telescript," *Digest of Papers, COMPCON '96, Technologies for the Information Superhighway, Forty-First IEEE Computer Society International Conference*, pp. 58-63, IEEE Computer Society Press, Los Alamitos, California, 1996.

[70] N. S. Borenstein, "E-mail with a Mind of its Own: The Safe-Tcl Language for Enabled Mail," *Upper Layer Protocols, Architectures and Applications, IFIP TC6/ WG6.5 International Conference,* Barcelona, Spain, June 1-3, 1994. Also in *IFIP Transactions C (Communication Systems)*, Vol. C-25, pp. 389-402, 1994.

[71] J. Y. Levy and J. K. Ousterhout, "A Safe Tcl Toolkit for Electronic Meeting Places," *Proceedings of the First USENIX Workshop of Electronic Commerce*, New York, New York, July 11-12, 1995, pp. 133-135, USENIX Association, Berkeley, California, 1995.

[72] J. K. Ousterhout, J. Y. Levy, and B. B. Welch, "The Safe-Tcl Security Model," draft paper, Sun Microsystems Laboratories, Mountain View, California, November 16, 1996.

[73] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[74] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, Massachusetts, 1996.

[75] M. A. Hamilton, "Java and the Shift to Net-Centric Computing," *Computer*, Vol. 29, No. 8, pp. 31-39, August 1996.

[76] D. Dean, E. W. Felten, and D. S. Wallach, "Java Security: From HotJava to Netscape and Beyond," *Proc. Symp. Security and Privacy*, pp. 190-200, IEEE Computer Society Press, Los Alamitos, California, 1996.

[77] W. Li, *Agent-Based Signaling for Service Negotiations on Broadband Networks*, Ph.D. Dissertation in preparation, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA, 94720.

[78] D. J. Le Gall, "MPEG: A Video Compression Standard for Multimedia Applications," *Communications of the ACM,* Vol. 34, No. 4, pp. 46-58, April 1991.

[79] D. J. Le Gall, "The MPEG Video Compression Algorithm," *Signal Processing: Image Compression*, Vol. 4, No. 2, pp. 129-140, April 1992.

[80] M. Grossglauser, S. Keshav, and D. Tse, "The Case Against Variable Bit Rate Service," *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, pp. 307-310, Durham, NH, USA, April 18-22, 1995.

[81] M. Grossglauser, S. Keshav, and D. Tse, "RCBR: A Simple and Efficient Service for Multiple Time-Scale Traffic," *ACM SIGCOMM '95*, Cambridge, MA, USA, August 28 - September 1, 1995. *Computer Communication Review*, Vol. 25, No. 4, pp. 219-230, October 1995.

[82] D. J. Reininger, D. Raychaudhuri, and J. Y. Hui, "Bandwidth Renegotiation for VBR Video Over ATM Networks," *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 6, pp. 1076-1086, August 1996.

[83] R. K. Berman and J. H. Brewster, "Perspective on the AIN Architecture," *IEEE Communications Magazine*, Vol. 31, No. 2, pp. 27-32, February 1992.

[84] J. J. Garrahan, P. A. Russo, K. Kitami, and R. Kung, "Intelligent Network Overview," *IEEE Communications Magazine*, Vol. 31, No. 3, pp. 30-36, March 1993.

[85] A. A. Lazar, K.-S. Lim, and F. Marconcini, "Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture," *IEEE Journal on Selected Areas in Communications*, Vol. 14, No. 7, pp. 1214-1227, September 1996.

[86] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *Proc. of Multimedia Computing and Networking 96*, San Jose, California, January 1996.

[87] A. A. S. Danthine, "Protocol Representation with Finite-State Models," *IEEE Transactions on Communications*, Vol. COM-28, pp. 632-643, April 1980.

[88] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *International Journal of Computer Simulation*, special issue on "Simulation Software Development," Vol. 4, pp. 155-182, April 1994.

[89] B. Stroustrup, *The C++ Programming Language*, Second Edition, Addison-Wesley, Reading, Massachusetts, 1991.

[90] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1994.

[91] N. Economides, "The Economics of Networks," *International Journal of Industrial Organization*, Vol. 14, No. 6, pp. 673-700, October 1996.
(URL http://edgar.stern.nyu.edu/networks/top.html)

[92] C. Antonelli, "The Economic Theory of Information Networks," in *The Economics of Information Networks*, C. Antonelli (Ed.), North Holland, Amsterdam, 1992.

[93] C. Antonelli, "Externalities and Complementarities in Telecommunications Dynamics," *International Journal of Industrial Organization*, Vol. 11, No. 3, pp. 437-447, September 1993.

[94] W.-T. Chang, W.-Y. Li, D. G. Messerschmitt, and N. Chang, "Rapid Deployment of CPE-Based Telecommunications Services," *Proceedings of GLOBECOM'94*, Vol. 2, pp. 876-880, San Francisco, California, USA, November 28 - December 2, 1994. (URL http://ptolemy.eecs.berkeley.edu/dgm/PAPERS/94/Globecom1/)

[95] W.-T. Chang and D. G. Messerschmitt, "Dynamic Deployment of Peer-to-Peer Networked Applications to Existing World-Wide Web Browsers," *Proceedings of the Telecommunications Information Network Architecture (TINA) '96 Conference*, Heidelberg, Germany, September 3-5, 1996.
(URL http://ptolemy.eecs.berkeley.edu/dgm/PAPERS/96/TINA1/)

[96] W.-T. Chang, S. Ha, and E. A. Lee, "Heterogeneous Simulation – Mixing Discrete-Event Models with Dataflow," invited paper, RASSP special issue of the *Journal on VLSI Signal Processing*, to appear, 1997.
(URL http://ptolemy.eecs.berkeley.edu/papers/96/heterogeneity/)

[97] W.-T. Chang, A. Kalavade, and E. A. Lee, "Effective Heterogeneous Design and Co-simulation," *NATO Advanced Study Institute Workshop on Hardware/Software Co-design*, Lake Como, Italy, June 18-30, 1995.
(URL http://ptolemy.eecs.berkeley.edu/papers/effective/)

[98] D. Harel and A. Pnueli, "On the Development of Reactive Systems," *Logics and Models of Concurrent Systems*, K. R. Apt, editor, volume F13 of *NATO ASI Series*, pp. 477-498, Springer Verlag, 1985.

[99] G. Berry, "Real Time Programming: Special Purpose or General Purpose Languages," *Information Processing*, G. Ritter, Ed., Elsevier Science Publishers B.V. (North Holland), Vol. 89, pp. 11--17, 1989.

[100] J. B. Dennis, *First Version Data Flow Procedural Language*, Technical Memo MAC TM61, MIT Laboratory for Computer Science, May 1975.

[101] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data flow Programs for Digital Signal Processing," *IEEE Transactions on Computers*, Vol. C-36, No. 1, pp. 24-35, January 1987.

[102] E. A. Lee and D. G. Messerschmitt, "Synchronous Data Flow," *Proceedings of the IEEE*, Vol. 75, No. 9, pp. 1235-1245, September 1987.

[103] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Synchronous Dataflow Graphs*, Kluwer Academic Press, Norwell, Mass., 1996.

[104] E. A. Lee, "Consistency in Dataflow Graphs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 2, April 1991.

[105] J. T. Buck, *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*, Technical Report UCB/ERL 93/69, Ph.D. Dissertation, Department of EECS, University of California at Berkeley, Berkeley, California 94720, 1993.

[106] R. Lauwereins, P. Wauters, M. Adé, J. A. Peperstraete, "Geometric Parallelism and Cyclo-Static Dataflow in GRAPE-II," *Proc. 5th Int. Workshop on Rapid System Prototyping*, Grenoble, France, June 1994.

[107] E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, Vol. 83, No. 5, pp. 773-801, May 1995.

[108] T. M. Parks, *Bounded Scheduling of Process Networks*, Technical Report UCB/ERL-95-105. Ph.D. Dissertation. EECS Department, University of California at Berkeley, Berkeley, California 94720, December 1995.

[109] A. Benveniste and G. Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1270-1282, September 1991.

[110] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993.

[111] S. A. Edwards, *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*, Ph.D. dissertation in preparation, Department of EECS, University of California at Berkeley, Berkeley, CA 94720 USA.

[112] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass., 1979.

[113] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Second Edition, Prentice-Hall, 1988.

[114] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, Vol. 8, No. 3, pp. 231-274, June 1987.

[115] M. von der Beeck, "A Comparison of Statecharts Variants," *Proc. of Formal Techniques in Real Time and Fault Tolerant Systems, LNCS 863*, pp. 128-148, Springer-Verlag, Berlin, 1994.

[116] A. C. Uselton and S. A. Smolka, "A Compositional Semantics for Statecharts Using Labeled Transition Systems," *CONCUR '94: Concurrency Theory, 5th International Conference*, Uppsala, Sweden, August 22-25, 1994. Springer-Verlag LNCS 836, pp. 2-17.

[117] F. Maraninchi, "The Argos Language: Graphical Representation of Automata and Description of Reactive Systems," *Proceedings of the IEEE Workshop on Visual Languages*, Kobe, Japan, October 1991.

[118] F. Maraninchi, "Operational and Compositional Semantics of Synchronous Automaton Compositions," *Proceedings of CONCUR '92, Third International Conference on Concurrency Theory, LNCS 630*, pp. 550-564, Springer Verlag, August 1992.

[119] G. Kahn, "The Semantics of a Simple Language for Parallel Programming," *Proceedings of the IFIP Congress 74*, North-Holland Publishing Co., 1974.

[120] G. Kahn and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," *Information Processing 77*, B. Gilchrist, editor, North-Holland Publishing Co., 1977.

[121] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *SIAM Journal*, Vol. 14, pp. 1390-1411, November 1966.

[122] G. S. Fishman, *Principles of Discrete Event Simulation*, John Wiley, New York, 1978.

[123] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, Second Edition, McGraw-Hill, Inc., 1991.

[124] J. Banks, J. S. Carson, II, and B. L. Nelson, *Discrete-Event System Simulation*, Second Edition, Prentice Hall, Upper Saddle River, New Jersey, 1996.

[125] R. H. Katz, *Contemporary Logic Design*, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1994.

[126] F. Belina, D. Hogrefe, and A. Sarma, *SDL with Applications from Protocol Specification*, Prentice Hall International (UK), Hemel Hempstead, Herfordshire, 1991.

[127] O. Færgemand and A. Olsen, "Introduction to SDL-92," *Computer Networks and ISDN Systems*, Vol. 26, No. 9, pp. 1143-1167, May 1994.

[128] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1994.

[129] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering,* Vol. 16, No. 4, pp. 403-414, April 1990.

[130] A. Orailoglu and D. D. Gajski, "Flow Graph Representation," *Proceedings of the Design Automation Conference*, 1986.

[131] J. Lis and D. D. Gajski, "Synthesis from VHDL," *Proceedings of the International Conference on Computer Design*, 1988.

[132] M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems," *Proceedings of the IEEE*, Vol. 78, No. 2, pp. 301-318, February 1990.

[133] B. Lee, *Fusing Dataflow with Finite State Machines*, M.S. Report, Dept. of EECS, University of California, Berkeley, CA 94720 USA, 1996.

[134] M. Pankert and S. Ritz, "Event Handling in Signal Flow Oriented Simulation and Synthesis," *Proc. Summer Computer Simulation Conference*, pp. 269-273, Reno, Nevada, July 1992.

[135] S. Ritz, M. Pankert, and H. Meyr, "High Level Software Synthesis for Signal Processing Systems," in *Proc. of the Int. Conf. on Application Specific Array Processors,* IEEE Computer Society Press, August 1992.

[136] M. Pankert, O. Mauss, S. Ritz, and H. Meyr, "Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis," *Proceedings of the 1994 IEEE International Conference on Acoustics, Speech, and Signal Processing,* Vol. 2, pp. 449-452, Adelaide, Australia, April 19-22, 1994.

[137] M. Jourdan, F. Lagnier, F. Maraninchi, and P. Raymond, "A Multiparadigm Language for Reactive Systems," *Proceedings of the 1994 IEEE International Confer-*

ence on Computer Languages (ICCL '94), pp. 211-218, Toulouse, France, May 16-19, 1994.

[138] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1305-1320, September 1991.

[139] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine, "The Algorithmic Analysis of Hybrid Systems," *Theoretical Computer Science*, Vol. 138, No. 1, pp. 3-34, February 1995.

[140] T. A. Henzinger, "The Theory of Hybrid Automata," *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 1996)*, pp. 278-292.

[141] S. Narayan, F. Vahid, and D. D. Gajski, *Modeling with SpecCharts*, Technical Report 90-20, Department of Information and Computer Science, University of California, Irvine, July 25, 1990.

[142] S. Narayan, F. Vahid, and D. D. Gajski, "SpecCharts: A Language for System Level Specification and Synthesis," *Proc. of the 10th Intl. Symp. on Computer Hardware Description Languages*, Marseille, France, April 1991.

[143] S. Narayan, F. Vahid, and D. D. Gajski, "System Level Specification and Synthesis with the SpecCharts Language," *Proc. of the International Conference on Computer Aided Design (ICCAD)*, November 1991.

[144] M. Jourdan and F. Maraninchi, "A Modular State/Transition Approach for Programming Reactive Systems," *ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, Orlando, Florida, 1994. (URL http://www.imag.fr/VERIMAG/SYNCHRONE/ACM-sigplan94.html)

[145] N. Lynch and M. Tuttle, *Hierarchical Correctness Proofs for Distributed Algorithms*, Master's thesis, Department of Electrical Engineering and Computer Sci-

ence, Massachusetts Institute of Technology, Cambridge, MA, April 1987. Technical Report MIT/LCS/TR-387. Abbreviated version in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pp. 137-151, Vancouver, British Columbia, Canada, August 1987.

[146] N. Lynch and M. Tuttle, "An Introduction to Input/Output Automata," *CWI-Quarterly*, Vol. 2, No. 3, pp. 219-246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam. Also, in Technical Memo MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, November, 1988.

[147] A. Girault, "Semantics of Hierarchical Finite State Machines," draft paper.

[148] R. Alur and T. A. Henzinger, "Reactive Modules," *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 96)*, pp. 207-218, IEEE Computer Society Press, 1996.

[149] M. J. McLennan, "The New [incr Tcl]: Objects, Mega-Widgets, Namespaces and More," *Proceedings of the Tcl/Tk Workshop 95*, pp. 151-159, Toronto, Ontario, Canada, July 6-8, 1995. USENIX Assoc., Berkeley, CA, USA, 1995.

[150] *Tycho, the Syntax Manager.* (URL http://ptolemy.eecs.berkeley.edu/tycho/)

[151] V. R. Lesser, S. H. Hawab, and F. I. Klassner, "IPUS: An Architecture for the Integrated Processing and Understanding of Signals," *Artificial Intelligence*, Vol. 77, No. 1, pp. 129-171, August 1995.

[152] G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, Vol. 19, No. 2, pp. 87-152, November 1992.

[153] F. Boussinot and R. De Simone, "The ESTEREL Language," *Proceedings of the IEEE*, Vol. 79, No. 9, pp. 1293-1304, September 1991.

[154] G. Berry, *Programming a Digital Wristwatch in Esterel v3.2*, Rapport de recherche n°8, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, 1991. (URL http://cma.cma.fr/ftp/esterel/wristwatch.ps.gz)

[155] M. Liou, "Overview of the $p \times 64$ kbit/s Video Coding Standard," *Communications of the ACM*, Vol. 34, No. 4, pp. 59-63, April 1991.

[156] P. Wayner, "Inside the NC," *BYTE*, Vol. 21, No. 11, pp. 105-110, November 1996.

[157] D. G. Messerschmitt, "The Flexible Connection," unpublished manuscript, 1992.

[158] *Frequently Asked Questions — Applet Security.* (URL http://java.sun.com/sfaq/)

[159] *Promondia — Java-Based Interactive Communication.* (URL http://www4.informatik.uni-erlangen.de/Projects/promondia/)

[160] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc., San Francisco, California, 1996.

[161] B. C. Neuman and T. Ts'o, "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications Magazine*, Vol. 32, No. 9, pp. 33-38, September 1994.

[162] A. O. Freier, P. L. Karlton, and P. C. Kocher, *The SSL Protocol*, Version 3.0, Internet Draft, March 1996. (URL http://home.netscape.com/eng/ssl3/index.html)

[163] Pendragon Software Corporation, "Battle of the Browsers", *The Java Performance Report*, July 1996. (URL http://www.webfayre.com/battle.html)

[164] P. Wayner, "Sun Gambles on Java Chips," *BYTE*, Vol. 21, No. 11, pp. 79-88, November 1996.

[165] T. R. Halfhill, "x86 Enters the Multimedia Era," *BYTE*, Vol. 21, No. 7, pp. 59-60, July 1996.

[166] J. Khazam and B. Bachmayer, "Programming Strategies for Intel's MMX," *BYTE*, Vol. 21, No. 8, pp. 63-64, August 1996.