A SYSTEM FOR CONSTRUCTING CONFIGURABLE HIGH-LEVEL PROTOCOLS

(Ph.D. Dissertation)

Nina Trappe Bhatti

TR 96-22

December 4, 1996

Department of Computer Science

The University of Arizona

Tucson, Arizona 85721

This research was supported in part by the Office of Naval Research under grants N00014-91-J-1015, N00014-94-1-0015, N00014-96-0207 and National Science Foundation grant CCR-9003161.

A SYSTEM FOR CONSTRUCTING CONFIGURABLE HIGH-LEVEL PROTOCOLS

by

Nina Trappe Bhatti

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 9 6

A SYSTEM FOR CONSTRUCTING CONFIGURABLE
HIGH-LEVEL PROTOCOLS

Nina Trappe Bhatti, Ph.D.
The University of Arizona, 1996

Director: Richard D. Schlichting

Distributed applications often require sophisticated communication services such as multicast, membership, group RPC (GRPC), transactions, or support for mobility. These services form a large portion of the supporting software for distributed applications, yet the specific requirements of the service vary from application to application. Constructing communication services that are useful for multiple diverse applications while still being manageable and efficient is a major challenge.

This dissertation focuses on improving the construction of complex communication services. The contributions of the dissertation are a new model for the construction of such services and the design and implementation of a supporting network subsystem. In this model, a communication service is decomposed into distinct micro-protocols, each implementing a specific semantic property. Micro-protocols have well-defined interfaces that use events to coordinate actions and communicate state changes, which results in a highly modular and configurable implementation.

This model augments, rather than replaces, the conventional hierarchical protocol model. In this implementation, a conventional *x*-kernel protocol is replaced with a composite protocol in which micro-protocol objects are linked with a standard runtime system that externally presents the standard *x*-kernel interface. Internally, the runtime system provides common message services, enforces a uniform interface between micro-protocols, detects and generates events, and synchronously or asynchronously executes event handlers.

The viability of the approach is demonstrated by performance tests for several different configurations of a suite of micro-protocols for a group RPC service. The micro-protocols in this suite implement multiple semantic properties of procedure call termination, message ordering, reliability, collation of responses, call semantics, membership, and failure. The tests were conducted while running within the *x*-kernel as a user level task on the Mach operating system.

Additional micro-protocols for mobile computing applications validate the generality of the model. We designed micro-protocols for quality of service (QoS), transmitting and renegotiating QoS parameters during handoffs, as well as for mobility management,

covering cell detection, handoff, and disconnection. This suite of micro-protocols can be configured to accommodate a range of different service requirements or even to mimic existing mobile architectures such as those found in the Crosspoint, PARC TAB, InfoPad, or DataMan projects.

STATEMENT BY AUTHOR

SIGNED: ____________________________

ACKNOWLEDGMENTS

My greatest thanks go to my husband, Scott Trappe, whose endless patience and encouragement I will always be grateful for. He gave me faith when I had none and constantly reassured me that one day I would finish. His unfailing love and support made all that I have accomplished possible.

The inspiration to pursue doctoral studies came from Gene Lawler, late professor of computer science at the University of California, Berkeley. I credit him for seeing my potential and encouraging me to pursue a research career in computer science. I cherished his love and support, and miss him dearly since his death. His belief that computer scientists must look beyond their research and understand its effect on society have forever shaped my ideals.

Richard Schlichting, my advisor, exhibited steadfast patience, endlessly reviewed chapters, and provided constant guidance while writing the dissertation. He fostered my professional development by encouraging me to attend and present at conferences and workshops and to intern at Xerox PARC. I will always appreciate his support for my interests, especially his defense of my choice to minor in marketing. Larry Peterson introduced me to networking and continues to provide invaluable technical advice. Greg Andrews encouraged and supported my efforts throughout my graduate study.

Chris Puto and Susan Heckler, my minor committee members, taught me how to reason about end-user's needs and identify opportunities. Chris discovered that this "gear-head" has other talents and consistently encouraged me to develop them. He is a teacher in the fullest sense of that word, and a wonderful mentor.

The entire Computer Science Department community made me feel very much at home. I will always be grateful to those cheerful miracle workers known as the lab staff, the office staff, and especially Wendy Swartz for leading me through the maze of university paperwork and regulations. I am indebted to the entire faculty, whose excellent instruction gave me an education I will always cherish and rely upon.

Many, many students — too many for me to name them all — have enriched my life both professionally and personally by sharing their knowledge, friendship and support. I thank Matti Hiltunen for many conversations about fault-tolerance and the event-driven model, and Wanda Chiu for insightful comments and for implementing the GRPC suite. Two groups deserve special mention: the "473 survivors club": Susie, Nisha, Bill, John, Jordan and Craig; and the "poker and B5 gang": Robert, Nick, Nevin, Sanford, Rich and Denise. My heartfelt thanks to all of you.

Finally, my thanks to the National Science Foundation (grant CCR-9003161) and the office of Naval Research (grants N00014-91-J-1015, N00014-94-1-0015, N00014-96-0207) for funding my studies.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Distributed applications often require sophisticated communication services such as multicast, membership, group RPC (GRPC), transactions, or support for mobility. These services form a large portion of the supporting software for distributed applications, yet the specific requirements of the service vary from application to application. Constructing communication services that are useful for multiple diverse applications while still being manageable and efficient is a major challenge.

This dissertation focuses on improving the construction of complex communication services. The contributions of the dissertation are a new model for the construction of such services and the design and implementation of a supporting network subsystem. In this model, a communication service is decomposed into distinct micro-protocols, each implementing a specific semantic property. Micro-protocols have well-defined interfaces that use events to coordinate actions and communicate state changes, which results in a highly modular and configurable implementation.

This model augments, rather than replaces, the conventional hierarchical protocol model. In this implementation, a conventional *x*-kernel protocol is replaced with a composite protocol in which micro-protocol objects are linked with a standard runtime system that externally presents the standard *x*-kernel interface. Internally, the runtime system provides common message services, enforces a uniform interface between micro-protocols, detects and generates events, and synchronously or asynchronously executes event handlers.

The viability of the approach is demonstrated by performance tests for several different configurations of a suite of micro-protocols for a group RPC service. The micro-protocols in this suite implement multiple semantic properties of procedure call termination, message ordering, reliability, collation of responses, call semantics, membership, and failure. The tests were conducted while running within the *x*-kernel as a user level task on the Mach operating system.

Additional micro-protocols for mobile computing applications validate the generality of the model. We designed micro-protocols for quality of service (QoS), transmitting and renegotiating QoS parameters during handoffs, as well as for mobility management, covering cell detection, handoff, and disconnection. This suite of micro-protocols can be configured to accommodate a range of different service requirements or even to mimic existing mobile architectures such as those found in the Crosspoint, PARC TAB, InfoPad, or DataMan projects.

CHAPTER 1
INTRODUCTION

Current computing applications often have sophisticated communication requirements. Users of an automatic teller machine (ATM) withdraw cash without realizing the complex communication performed to verify the transaction and correctly record it at the appropriate bank anywhere in the world. Regardless of location, users trust that the transaction will be performed correctly, reliably, and quickly. This type of functionality is realized by an underlying *communication service*, which governs how messages are exchanged between hosts and what delivery guarantees are associated with them. For example, in this case, an appropriate communication service might provide guaranteed delivery of the transaction and make permanent changes to an account only if the transaction is completed successfully. A large portion of the supporting software of *distributed applications*—that is, applications constructed on collections of machines connected by a network—is composed of communication services.

While communication services are universal for distributed applications, the specific requirements vary depending on the type of application. For example, robust, reliable communication may be required for an ATM application, but less strict requirements are acceptable for sending mail since maximum delivery delays and ordering of messages are typically not guaranteed. Another example is video, which requires predictable delays between frames and ordered delivery, but which can accept occasional frame losses. Hence, video applications can be well supported by a communication service that provides timely, ordered, but unreliable communication. This large variety of possible semantics are difficult to realize in a single service.

A major challenge is constructing communication services that are useful for multiple diverse applications while still being manageable and efficient. One approach is to use *customization* and *configurability*. Currently, communication systems provide only simple mechanisms, so applications build more complex services from scratch based on basic guarantees such as best-effort delivery. An alternative is to provide more complex semantics by customizing a general communication service to match the exact needs of the application. To implement this customization, users configure a service from a collection of software modules, each of which implements a specific behavior. In addition to facilitating an exact semantic match, this approach allows many applications with similar communication needs to use the same basic service instead of requiring the creation of another similar service.

This dissertation addresses the construction of customized communication services by proposing a new approach in which fine-grain software modules called *micro-protocols* encapsulating a function or property are combined to form a communication service. The selection of a micro-protocol enforcing a particular property such as reliability allows the user to create a service with exactly the desired behavior. In addition, micro-protocols that implement variants of the same behavior can be exchanged; for example, different types of delivery orderings such as FIFO, unordered, causal, and total ordering can be selected. Micro-protocols communicate with other micro-protocols using *events*, which makes all data sharing and dependencies between micro-protocol explicit. The approach has been realized in a prototype implementation based on the x-kernel [HP91].

1.1 Distributed Systems

Communication services with powerful semantics are an important building block for distributed applications. However, as noted above, the specific functionality can vary greatly depending on the type of application. Here, we outline common reasons for writing distributed applications — improved performance, increased dependability, accommodation of physical requirements, and convenient resource sharing — and then discuss their communication requirements.

Performance

For many applications, performance can be improved by exploiting *parallelism*, that is, by distributing parts of the computation among hosts within a distributed system and executing them concurrently. In some cases, the parts can be executed to completion, with the results then being combined to produce the final overall result. In other cases, the process is iterative, with processes periodically exchanging intermediate values. Still other applications may be be parallelized by assigning computation resources to specific stages of the solution; in this scenario, resources are organized as a pipeline, with each resource producing results that are passed to the next stage in the pipeline. While not all problems can be parallelized, parallelism is widely used to obtain performance improvements for scientific computations.

Performance can also be improved using distributed systems by migrating processes to idle hosts to spread the computation load and to make use of idle processing power [Dou89, Dou87, OCD+88]. However, since process migration incurs overhead, a performance improvement is realized only if the target host is underutilized, and if the remaining computation time is greater than the time required to do the migration. This makes this technique useful only in limited circumstances where idle hosts are commonly available, such as offices where large numbers of hosts are unused after business hours.

Dependability

Another reason for building an application using a distributed system is to increase *dependability*, which is a measure of the reliance that can be placed on the service a system delivers [Car82]. The service delivered by a system is its behavior as perceived by users, whether they be humans or other systems [Lap92]. Dependability encompasses a number of different attributes. A dependable ATM system, for example, services transactions 24 hours a day (*availability*), functions correctly under heavy loads (*reliability*), does not dispense money without a valid ATM card (*safe*), and protects the ATM PIN number during transmission (*secure*).

A distributed system may incorporate redundant components that can be used to mask failures. As a result, when a component *fails* (does not execute according to its specification) one of the correctly functioning components may be able to take over and provide the same service. For example, a dependable file system can be created by maintaining a mirror image on a second storage device and applying each state-modifying operation to the mirror image as well. Then, if the original storage device fails (e.g., crashes), the mirror file system can be used. In addition to increasing availability, redundancy can be used to verify correct operation by executing computations on multiple CPUs and then comparing results.

Physical Requirements

The physical nature of an application may force separation of computational resources, which naturally then leads to its construction as a distributed system. For example, an automated manufacturing plant where multiple processors control the manufacturing process is a distributed system because the processors are physically separated and communicate using a network. A similar example is a system composed of medical patient monitoring equipment, in which a specialized computer system is located physically near each patient. To control patient monitors, consolidate patient information, and prepare patient status reports requires using a distributed system.

Distributed systems based on physical separation of processors do not require that those processors remain stationary. In particular, the processors may be embedded in small devices intended to be carried by the user, such as the Apple Newton or Hewlett-Packard 95LX. *Mobile computing systems* have the same basic elements as other distributed systems, but with the additional requirements of coping with changing location information and potentially slower and more error-prone communication links based on radio or infrared technology.

Resource Sharing

Distributed systems can also be used to provide resources to a group of clients. Perhaps the most ubiquitous example of this type of *resource sharing* is a printer attached to a network. In this case, the printer is a server that provides printing services to several hosts (clients) connected to the network. The same idea can be applied to other peripherals, such as a network file system where services are provided to multiple clients by a single shared server. Programs can also be shared in a similar way. For example, a common license agreement is to allow the program to run on any of multiple hosts, but with only a limited number active at a time.

Resource sharing is also an efficient way to use expensive resources. For example, an expensive supercomputer can be cost-effectively shared by using a network to submit jobs, potentially over great distances. This strategy avoids the expense of purchasing multiple specialized machines while still providing access to a large community of users.

Communication Services

The differing characteristics of applications in each of these four categories—performance, dependability, physical requirements, and resource sharing—has motivated the development of a variety of communication services. For example, parallel applications can be simplified using a multicast primitive to disseminate the results of a subproblem or coordinate execution. Similarly, many dependable applications rely on communication services that automatically detect failures and maintain consistent views of which machines are functioning. Communication systems intended for mobile computing applications must maintain communication between physically separate hosts that change location, so a connection and delivery mechanism that tolerates mobility is an essential tool. Finally, resource sharing is facilitated if clients can reliably transmit requests to a group of servers using a single address; using this, any server can respond.

In this dissertation, we focus on two specific types of distributed systems: highly dependable and mobile computing systems. The next two sections elaborate on these areas and the types of communication services that are most useful.

1.2 Dependability and Fault-Tolerance

Several paradigms have been developed to reduce the complexity of *fault-tolerant software* [MS92]. The paradigms are based on network-oriented abstractions that are often realized as communication services with a wide range of behaviors. This section describes several paradigms, their application structure, and their communication services.

The *object/action paradigm* structures an application as a collection of objects that are located across multiple machines in a distributed system. Objects are passive entities that encapsulate state and export operations that modify state; actions invoke these operations and are *serializable*, *atomic*, and *permanent*. An action is serializable if its concurrent execution with other actions always has the same effect as some serial execution order. An action is atomic if intermediate states are never visible despite failures; if a failure occurs, the action is aborted and any state changes are undone. Finally, an action is permanent if once the action commits, the results of the action cannot be undone by subsequent failures.

The object/action paradigm is often implemented using processes for objects and threads for actions. In this scheme, invocation of an operation on a remote object is realized using Remote Procedure Call (RPC), which is similar to standard procedure calls except that the procedure is executed by a different process [Nel81, BN84]. The caller is known as the *client* and the process executing the procedure is known as the *server*. Also, multiple variants of RPC may be needed depending on the specifics of the approach. For example, additional fault tolerance can be provided by replicating server processes. In this case, group RPC is used by the client to transmit the call to the entire group instead of just a single process. Some implementations may also need group RPC services that guarantee that each call is unique or that all servers receive each call. Other abstractions that are useful for supporting the object/action model include stable storage for permanence of actions, atomic actions for atomicity, and resilient processes for failure recovery [MS92].

The *conversation paradigm* uses processes and messages as its main components. An application is structured as a collection of concurrent processes executing on different hosts and communicating by exchanging messages. Processes periodically save an image of their process state, called a *checkpoint*. A programming construct called a conversation is used to ensure that these checkpoints are *consistent*, where a consistent set of checkpoints is one in which every message receive event is matched by a corresponding send event. If a process fails, a replacement process is created and started from the last checkpoint. The other non-failing processes also roll back their state to the corresponding checkpoint so that all processes return to the same point in the computation.

Different variants of RPC are again useful as supporting communication services for the conversation paradigm. For example, depending on the specific semantics, different types of message ordering may be desirable, such as causal or total ordering [CASD85, Cri89, BCG91, BSS91a, BSS91b, Coo90, GMS89, GMS91, GMK88, VM90]. Stable storage is also a useful abstraction for storing checkpoints [Lam81].

The *state machine approach* structures an application as a collection of interacting state machines [Sch90]. *Commands* received as messages from other machines or the environment change the values of state variables. The execution of each command is deterministic and atomic with respect to other commands to provide fault tolerance. State machines can be replicated, in which case each command is received and executed by all replicas. Assuming that all replicas execute the commands in the same order, the states of the replicas will remain consistent. As a result, the failure of some number of replicas can be tolerated without affecting operation of the whole group.

A number of communication services are useful building blocks for the state machine approach. Since all state machines must receive the same commands in the same order, atomic ordered multicast [CM84]—which guarantees atomicity and consistent ordering of messages—is a valuable abstraction. Membership is also an important service; it maintains consistent information about which machines are functioning and which have failed.

The goal of the work presented in this dissertation is to simplify the design and implementation of communication services such as the ones described above for fault-tolerant software. We concentrate primarily on group RPC and, to a lesser extent, membership. Our approach allows these services to be implemented as a configurable collection of fine-grained modules that can be used to meet a variety of fault-tolerance requirements. The mechanism for building these services are *protocols*, which specify the format and meaning of messages that are exchanged by instances of a communication service executing on different machines. The term is also often used to refer to the actual software that implements the agreed-upon behavior. Examples of well-known protocols are the Transmission Control Protocol and Internet Protocol (TCP/IP) [Pos81a, Pos81b] and the User Datagram Protocol (UDP) [Pos80].

1.3 Mobile Computing

Mobile computing systems are composed of mobile and stationary hosts that exchange messages through wireless and wired communication links. These systems are a direct consequence of the availability of new devices that are inexpensive and light enough to be truly portable. The two key problems in mobile computing are hiding the details of *mobility* and managing *quality of service*. Ideally, mobile systems would behave exactly as any other distributed system with respect to addressing and sending messages. However, while fixed addresses for mobile devices are now the norm, systems vary greatly in how location information is gathered and propagated. The next two sections elaborate on mobility and quality of service to illustrate the multitude of possible communication services.

1.3.1 Mobility

Mobility presents two challenges: routing and disconnected operation. The first concerns how messages can be delivered to hosts that change their location and how host location information is propagated and cached. Making routing transparent to the application is valuable because resources can then be accessed without regard to location. For example, instead of sending a print job to a printer by name, the job can be dispatched to the nearest printer resource. File system caching can be done similarly, with mobile hosts caching files in file servers that are nearby. Knowledge of host location can also provide additional capabilities, such as answering certain queries (e.g., nearest restaurant) using location-specific information [AIB].

A number of solutions have been proposed to the mobile routing problem. Some solutions assign mobile hosts Internet addresses and route messages transparently as hosts roam between different domains in the Internet [IDJ91]. Other solutions deal only with routing messages to mobile hosts in a local area. For example, in the area of maintaining location information, some approaches only maintain data on hosts that are active [AGSW93], while others require hosts to respond explicitly to identification requests [CLR95]. Similarly, some routing protocols propagate information about active hosts in the area to the rest of the system immediately [CLR95], while others send information only to a central manager [KMS+93]. A single configurable system that allows any of these multiple solutions to be easily constructed would be useful for matching the routing to the particular application.

The second challenge concerns complications resulting from failures of a mobile host or disconnection due to failed communication links. Such events occur more frequently in mobile than stationary systems because of the lossy wireless transmission medium, the possibility of mobile hosts roaming out of range, or the unavailability of a host due to battery power loss. Except for the last case, the mobile host may be able to continue functioning, in which case some type of state synchronization will be required when the mobile host reconnects to the network. *Disconnected operation* of this type is only of concern for mobile hosts that have sufficient resources to function standalone; hosts without such resources simply cease to function if disconnected.

There are a variety of approaches for dealing with disconnection Many ideas can be borrowed from fault-tolerance, such as replication of the mobile host state and atomic multicast to ensure that multiple hosts receive messages [VKP93, PKV96]. However, since failures are more common, any technique must be inexpensive, as well as not interfere with the rest of the system. Other solutions are specific to mobile computing, such as cases where routing protocols redirect messages destined for a disconnected host to a proxy that can save messages until the mobile host reconnects. Multiple techniques exist to deal with state synchronization after disconnected operation [Kis90, KS91, HH93, SKM+93, NK93, MBM95]. Thus, like routing, it would be advantageous if the communication service could be altered to provide this variety of options without modifying the rest of the system.

1.3.2 Quality of Service

Quality of service refers to the performance guarantees provided to an application by the underlying system, often after negotiation between the two entities. Determinants of quality of service are *bandwidth*, *latency*, and *jitter*. In mobile communication, bandwidth means throughput, latency is the delay before receiving the first bit of data from the sender, and jitter means the variation in delay between messages. An additional criterion is *connectivity latency* due to movement between the areas serviced by distinct transmission devices *cells*. Moreover, since each cell is a different transmission domain, bandwidth, latency, and jitter can change after entering a new cell. As a result, the agreed-upon quality of service characteristics may have to be re-negotiated with the new cell. Hence, not only must resources be shared as is the case with any system, but frequent host arrivals and departures must also be accommodated.

Quality of service is, almost by definition, associated with different policies, so it is not surprising that a variety of approaches have been defined. For example, bandwidth and mobility are inherently linked. When an application starts a communication stream to a mobile host, it negotiates the required resources. If a host moves into an area where there is already a lot of traffic, some policies ignore the needs of the new host, while others modify existing connections or use priorities. It would be advantageous for a single communication service to support all these variants of system behavior.

1.4 Communication Support

In previous sections we have outlined how different communication services can be used to simplify distributed applications such as those associated with fault-tolerant systems and mobile computing. By providing rich functionality and a high degree of abstraction, such services make it easier to handle the uncertainties inherent in distributed systems, including those associated with network communication, distributed synchronization, mobility, and processor crashes. Often the protocols that implement these services are described as *middleware* since they form a software layer between the application and the operating system. From a networking perspective, they are considered *high-level protocols* because they provide enhanced functionality relative to simple message delivery.

Unfortunately, while such high-level protocols are useful, their construction poses a number of challenges. These protocols are difficult to design, debug, and modify, largely due to the same complex functionality that makes them so useful. Another reason is that they are often built specifically for a given application rather than as a separate layer that can be reused. As a result, the application software is much more complex, and it becomes difficult to update should the communication guarantees required by the application change.

To address these and other challenges, an approach for building communication services would ideally exhibit a number of characteristics. It should facilitate services that are highly configurable to realize multiple semantic variations. It should also aid the protocol writer in designing and implementing these variations, and be simple for users of the service to configure. Finally, configurability should not come at the expense of performance.

An option that approaches this ideal mix is to implement the functionality as a collection of smaller protocol objects (a *protocol suite*) and then use a system like ADAPTIVE [SBS93], Horus [vRHB94], or the *x*-kernel [HP91] to combine the objects into a network subsystem. However, despite their advantages over monolithic realizations, these systems still have a number of deficiencies when it comes to implementing high-level protocols. These include inadequate support for fine-grained modules with complex interaction patterns, limited facilities for data sharing, and an orientation towards hierarchical protocol composition at the expense of more flexible combinations. Experience suggests that these limitations increase the difficulty of implementing high-level protocols using these systems. For example, problems of this type have been encountered with the *x*-kernel, both in Consul, a protocol suite implementing atomic multicast [MPS93a, PBS89], and xAMP, a real-time atomic multicast protocol [VRB89].

The ability to build configurable communication systems is also consistent with the general trend of applications requiring more control over their support services. For example, operating systems are increasingly being designed to allow customized approaches to services, such as scheduling, file systems, and caching [BCE+95, HPM93, MMO+94]. Our research is concerned with providing enhanced control over communication services, with a focus on supporting modular implementations and fine-grain semantic-based configuration.

1.5 Composite Protocol Approach

This dissertation describes a new structuring approach that supports highly modular implementations of communication services. With our approach, a high-level protocol is constructed from a collection of *micro-protocol objects* (or just *micro-protocols*) that implement individual semantic properties of the target system. For example, with atomic multicast, one micro-protocol might implement the consistent ordering requirements, while another might implement reliable transmission. Micro-protocols can also be used to implement different semantic variants of the same property. For example, with RPC, there may be several micro-protocols implementing distinct policies for how a request is handled if the server fails, such as *exactly once*, *at least once*, or *at most once* semantics [PS88]. A system is then configured based on the particular properties needed for the given application.

This micro-protocol approach is realized by augmenting the *x*-kernel's standard hierarchical object composition model with the ability to internally structure protocol objects. The result is a two-level model in which selected micro-protocols are first combined with a standard runtime system or *framework* to form a *composite protocol*. This composite protocol, whose external interface is indistinguishable from a standard *x*-kernel protocol, is then composed with other *x*-kernel protocols in the normal hierarchical way to realize the overall functionality required of the network subsystem. Internally, the framework implements an event-driven execution paradigm, in which micro-protocols are executed whenever events for which they are registered occur —for example, message arrival or a timeout. Thus, when compared with standard x-kernel protocol objects, micro-protocols are typically finer-grain objects that interact more closely and do so using mechanisms provided by the framework rather than the x-kernel Uniform Protocol Interface (UPI).

Our approach has a number of benefits. For example, the flexibility of the two-level model is useful for dealing with dependencies among the properties implemented by complex protocols. It also offers the development benefits associated with modular implementations, as well as an enhanced ability to tailor the system to the specific characteristics of a given application or architecture. In contrast with similar systems for constructing configurable protocols, our approach provides finer granularity and more flexible inter-object communication, which is especially useful for configuring closely-related service variants of the same general type of high-level protocol (e.g., variants of atomic multicast).

The contributions of this dissertation are:

- A new approach to constructing configurable high-level protocols with properties customized to the needs to the application or the specifics of the architecture.

- An x-kernel based system for realizing this approach.

- The use of fine-grain micro-protocols that have limited, well-defined interfaces and are executed according to an event-driven paradigm.

- Examples of designs and implementations based on our approach for two families of communication services, group RPC and mobile computing, as well as a description of the resulting lessons substantiating the viability of the approach.

- An assessment of the execution costs involved with the approach and insights into the incremental execution cost of properties in certain common group communication paradigms.

1.6 Dissertation Outline

This dissertation is organized as follows. Chapter 2 describes related work in the areas of communication support for fault tolerance and mobile computing, approaches for constructing services using modular protocols, parallel execution of protocols and configurable operating systems. Chapter 3 then describes how our composite protocol approach can be used to create micro-protocol suites. Implementation details and basic performance results are given in Chapter 4.

Chapter 5 illustrates the use of the composite protocol approach to build a highly configurable version of group remote procedure call, which is often used in fault-tolerant applications. We also give performance results from a number of experiments involving the x-kernel prototype and multiple different configurations. Similarly, Chapter 6 provides an in-depth look at a configurable composite protocol for mobile computing.

The composite protocol model and the x-kernel implementation are evaluated in Chapter 7. Finally, Chapter 8 makes some concluding remarks and presents future research ideas.

CHAPTER 2

COMMUNICATION SERVICES AND CONSTRUCTION TECHNIQUES

Software for distributed systems is complex and difficult to write. To simplify this task, communication services are used to provide an abstraction that is easier to program. In this chapter, we describe communication abstractions and protocols useful for fault-tolerant systems and mobile computing, including multicast, membership, RPC, and various systems that support mobility. Then, the current state of technology for developing these services is described. In particular, several projects are presented that explore the use of modularization or system customization. Finally, we present recent work on new generation operating systems that emphasize similar customization goals, but in a more general context. The configurability provided by these systems is typically coarse-grained and allows freedom only in selected areas.

2.1 Multicast and Membership

2.1.1 Overview

In fault-tolerant systems, providing consistent information to multiple processes is necessary for constructing many types of distributed applications. One way to provide this is to use multicast, which sends messages to a collection of processes organized as a multicast group. Thus, there is one sender and multiple receivers. Multicast groups typically have a multicast group address, membership rules, and possible restrictions on addressing messages to the group.

Multicast sending of messages may or may not be done with hardware support. Without hardware support, conventional point-to-point messages are used. With hardware support in local area networks, messages can be delivered more efficiently since a single destination address can be used to refer to all group members.

Multicast services in wide-area networks are concerned with managing group membership and efficient delivery of messages [DC90, Dee94, DEF$^+$94, Hug88, WPD88, ALB88, Wal80]. For example, one scheme for multicasting messages in the Internet uses IP routers to disseminate packets. The routers recognize the destination address as a multicast address and forward packets to links if there are group members reachable through the links; otherwise the message need not be propagated. Several spanning tree algorithms have been developed to manage routing topologies, and minimize the number of routers and networks involved in forwarding multicast packets.

While there are many variations of multicast for fault-tolerant systems, reliability and similar guarantees are often more important than efficient packet routing. In fact, multicast services designed for this purpose can be broken into five orthogonal properties [MS92]:

Dissemination. The message is disseminated to all processes in a group. As noted, in local-area networks such as Ethernets or Token rings that provide a multicast primitive, the dissemination can be done efficiently with a single lower-level operation.

Atomicity. Messages are delivered to all operational processes in the group or to none. This property ensures that all processes see the identical set of messages.

Reliability. Message are delivered to every process in the group. If a member of the group has failed, then the message will be provided to the failed process during its recovery.

Order. Messages are delivered in some consistent order to all group members. Especially in wide-area networks, multicast messages can potentially arrive in different order at each of the members, which complicates higher-level software. There are several possible consistent orderings, including:

- *Partial order or causal order:* Messages are delivered in an order that preserves the happened before relationship (causality) [Lam78]. Messages are only delivered after all the messages that precede it have been delivered. Messages for which no happened before relationship exists are considered to be concurrent and can be delivered in any order.

- *Semantic dependent order:* Messages are delivered in an order that depends on the semantics of the information in the message. For example, messages that contain commutative operations can be delivered in different order, while messages with non-commutative operations must be delivered in the same order to all processes.

- *Total order:* Messages are delivered in the same order to all processes. There is no restriction on the ordering, other than it must be exactly the same for each process.

- *Total order preserving causality:* Messages are delivered in the same order to all processes, and the order preserves the happened before relationship between messages. This is the most expensive ordering property to implement.

Termination. The communication protocol is synchronous if every message is delivered to all correct processes within a fixed time interval.

A problem that is closely related to multicast is membership, which involves maintaining information about which processes belong to the group. Membership can be completely static but more interesting groups have dynamic membership. In this case, members are added when they explicitly request to join and removed when they request it or, in the case of fault-tolerant systems, when they fail. Membership of the group may be *open*, allowing any process to join, or *closed*, in which case admission may or may not be granted when requested. While only group members can receive messages addressed to the group, sending may either be restricted to members or unrestricted. Unrestricted sending is important for a group of processes that offer a service and advertise the multicast address for general use.

Since membership change information does not instantaneously disseminate across the network, not all members have the same membership information at the same time. This can make a common view of the group difficult to achieve and therefore, make multicast guarantees difficult to provide if they depend on accurate membership information. As a result, membership services typically provide guarantees about the consistency of information at different members and how this relates to message delivery.

Many papers have been written on multicast and membership. Multicast variations that guarantee atomicity with respect to all functioning group members and ordering are described in [GL92, Spa91, AGKK91, GMK88, GMS89, GMS91, BM89, BSS91a, CM84, CASD85, KTHB89, NCN88, PBS89, VRB89, MPS89]. Other multicast research efforts are concerned with fast multicasting [RS92], low-cost multicasting [BA89], language support for group multicast [Coo90], and multicast communication for mobile hosts [AB93]. As described, membership also has many variations, which are summarized in [HS95b].

2.1.2 Consul

The Consul system provides multicast services for group communication, including causally ordered atomic message delivery and membership [MPS93a]. Causal message ordering is managed by a dependency graph called the *context graph*, which is implemented by the Psync protocol [MPS92]. Each process in the group builds a context graph and only messages for which all predecessors have been received are committed to the application. Messages that have no dependency relationship with respect to each other can be delivered in any order. Consul also provides totally ordered communication in which messages with no ordering relationship have an imposed consistent ordering at each process. Consul manages dynamic closed group membership, requiring all processes to execute join and leave operations. It detects processes that are no longer functioning and removes them from the group after agreement is reached with the other correctly functioning members. Consul has been successfully implemented as a modular system using the *x*-kernel [MPS93b, MPS93c]. However, as mentioned in Chapter 1, this implementation effort revealed that more support for modularity was needed for complex protocols that can be subdivided into many communicating submodules.

2.1.3 ISIS

The ISIS distributed programming toolkit has been used to develop many commercial applications and is perhaps the best known reliable multicast service [BSS91b, BC91, BSS91a, Bir85]. It provides failure atomicity, delivery ordering guarantees, and group addressing. Delivery ordering guarantees are atomic delivery of messages in either total order (ABCAST) or causal order (CBCAST). CBCAST is similar to the partial order delivery of Psync, but extends causality to multiple overlapping groups as well. ABCAST provides total ordering by layering another protocol on top of the casual ordering protocol, giving a total ordering that is causality preserving. In addition to message ordering guarantees, ISIS provides group membership facilities. ISIS has been constructed as a large monolithic implementation that is available on a number of platforms. The Horus system, described in Section 2.4.3.below, is a new modular implementation framework where services are created from protocol objects that can be stacked in a variety of ways. The multicast and membership services of ISIS have also been built using this new modular framework.

2.1.4 Transis

The Transis communication sub-system is a transport layer that supports multicast and automatic maintenance of dynamic membership over arbitrary network topologies [ADKM92]. The network is divided into *broadcast domains* that disseminate messages using hardware-supported broadcast, with point-to-point messages being used between domains. Transis provides immediate reliable delivery of messages to all active sites, as well as causal, totally ordered, and safe multicast. In the last, messages are only delivered after being acknowledged by all active sites. The membership service supports detection of failed hosts, group partitions, and joins. Partitions can be unified through join operations that add multiple hosts to the group.

2.1.5 Totem

The Totem system provides totally ordered delivery of messages for single local-area networks or multiple local-area networks connected by gateways [MMSA+95]. Totem aims to achieve good performance by using hardware broadcast in local-area networks and a scheme based on a logical circulating token. The token is also used as the basis for reliable delivery of messages, total ordering, and failure detection. Local area ordering is provided using sequence numbers associated with the token, with global ordering across multiple networks being handled by timestamps. In particular, gateways that connect two local-area networks receive messages from neighboring rings and order the messages with new sequence numbers based on the timestamp value in the message. Totem also provides a membership service that handles processor failure and recovery, and network partitions.

2.2 Remote Procedure Call

2.2.1 Overview

A well known and commonly used IPC mechanism is Remote Procedure Call (RPC). To the caller, RPC appears to be an ordinary procedure call except that the procedure is executed by a separate process that may be executing on a separate host. RPC is convenient for programming distributed applications, because it provides a well-understood procedural interface and automatic marshaling of arguments for network transmission. The calling process is referred to as the *client*, and the callee process is the *server*.

When all processes are functioning correctly, RPC gives results indistinguishable from a local procedure call except for timing. However, failure is an inherent problem given the uncertainties associated with a distributed system; a host can be unresponsive because it has crashed, messages were lost, or the network has become partitioned. Therefore, RPC semantics also describe what can be inferred after a failure of this type. In addition to failures, RPC request messages may arrive out of order, or be lost or duplicated. If RPC is to provide a good abstraction for building distributed applications with fault-tolerant requirements, there must be clear semantics defined for the case when processes are not behaving correctly.

Non-group RPC services can be differentiated based on what can be inferred by the client process about the number of times a remote procedure has been executed when a server is unresponsive and the call terminates abnormally. Below are listed common execution semantics, from the weakest to the strongest guarantees:

- *At Least Once.* If the invocation terminates normally the remote procedure has been executed one or more times. If abnormal termination occurs, then no conclusion can be drawn; it may have executed one or more times, not at all, or partially.

- *Exactly Once.* If the invocation terminates normally the remote procedure has been executed exactly one time. If abnormal termination occurs, then it has been executed only once, not at all, or partially.

- *At Most Once.* If the invocation terminates normally then the procedure has been executed exactly once. Otherwise, if the termination was abnormal, then the procedure was executed once or not all; execution is atomic even if a failure occurs.

Group RPC (GRPC) is similar to conventional point-to-point RPC, but instead of sending a request to a server, the request is sent to a server group [Che86, Coo90, CGR88, SS90]. This is especially useful for constructing fault-tolerant services using replicated servers. It is also possible to have a group of clients interacting with a group of servers. GRPC has several semantic aspects in addition to those found in point-to-point RPC [HS95a]. These include:

- *Ordering.* FIFO order guarantees that all calls issued by a single client are executed by each server in the same order. Total order guarantees that all calls by all clients are executed in the same order by each server. Causal ordering of client requests preserves the causal relationship of client requests.

- *Collation.* Collation semantics governs how responses from a server group are combined and returned to the client. Specifically, it governs how many responses are needed to complete the call and if these responses need to be the same.

- *Failure.* Failure semantics characterize what can be said about execution of a client request during a failure of the server, specifically unique execution and atomic execution [LG85]. Unique execution of requests guarantees that the request will be executed no more than once. Atomic execution guarantees that the request will either have been completely executed or not at all; there is no visible intermediate state.

2.2.2 Sun RPC

Sun RPC is perhaps the most common RPC protocol and can be configured with two different transport protocols, UDP and TCP. When using TCP, requests are received at most once and in order; if the server crashes then the client is guaranteed exactly once semantics. When using UDP, only at least once semantics are guaranteed. Sun RPC has broadcast capability to send a client request to a group of servers, but no server responses are allowed and there are no ordering guarantees. If responses are required, the servers use separate RPC calls that are manually collated by the client. Other RPC services are provided by [SB90, BALL90, Cou81, BN84, ATK91, PS88].

2.2.3 Group RPC Systems

Sun RPC has been used as a basis for a fault-tolerant group RPC service in [YJT88]. In this system, the servers are organized as a linearly ordered group, with the first process in the group —the *primary*— receiving all RPC calls and sending all replies. The primary forwards calls to the next server, which is one of the *secondary processes*, and each forwards the request to its successor. Each server is deterministic and calculates a reply and sends it to its predecessor; thus all servers will calculate a response before the client's RPC request returns. Failures are detected by the server group and, if the primary is unresponsive, by the client. When a failure is detected, the successor of the primary becomes the new primary. Note that no state recovery is necessary since all secondary servers execute all calls. At most once execution is provided, as well as FIFO ordering. All servers responses are identical, but only one response is returned to the client process.

Another Sun RPC-based group service offers three variations of semantics to support common classes of applications [WZZ93]. A *lookup* style of GRPC dispatches an RPC call to a group of servers and if any of them respond, the call completes. No ordering or execution guarantees are provided, so this is ideal for servers that are stateless and just respond to lookup requests from clients. The *functional-convergence* GRPC style provides no ordering, but all server responses are collected before the call returns. The strongest guarantees are provided by the *update* GRPC style, which guarantees total ordering of requests. In this case, all servers must successfully respond before a call can be completed. As a result, if any server declares a failure, the call will return abnormally.

Circus is a group RPC service that is distinguished by the fact that the *troupe* of servers do not communicate with one another [Coo85]. This simplifies the system since the lack of interaction means that servers do not have to manage membership. Instead, clients detect server crashes and remove the failed server from the server troupe. Circus supports exactly once execution and FIFO ordering. Collation of results can be unanimous, majority, or first result.

Fault-Tolerant Concurrent C is a parallel programming extension of the C language that provides group RPC to support active replication of deterministic identical server processes [CGR88]. The collation semantics is first result; all other replies are discarded. All calls are executed in FIFO order by all servers.

2.3 Mobile Computing

2.3.1 Overview

Mobile computing is an emerging area of distributed systems, so unifying concepts and themes have yet to be defined. The current state of research is a multitude of systems with different hardware configurations and different wireless communication technologies,

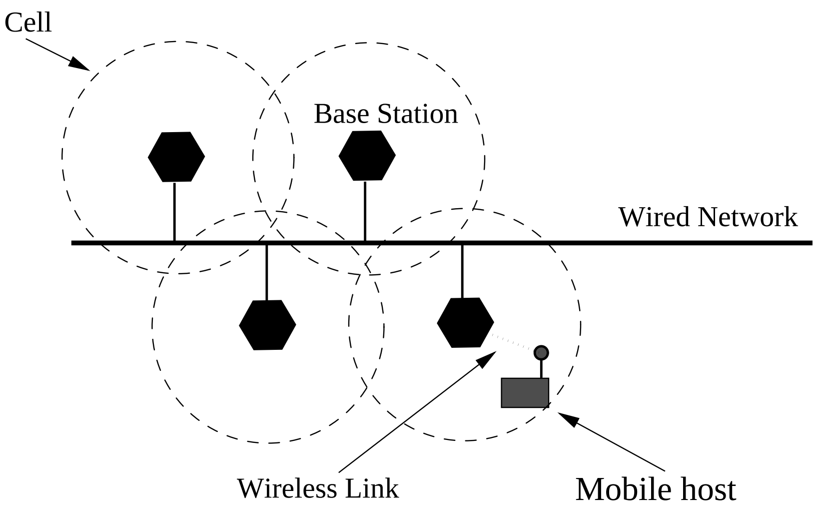

Figure 2.1: Hardware components of a typical mobile distributed system.

all attempting to provide connectivity for roaming users. The mobile devices can roam small office-size areas, campus-wide areas, or even across the country.

Despite this variety, all systems have a number of common core elements, as shown in Figure 2.1. These include:

Mobile Hosts. A roaming device equipped with a wireless communication link. May range in power from a dumb terminal to a powerful stand-alone machine.

Base Station. A stationary computer with a wireless communication link and a connection to a conventional wired network. Base stations can receive and transmit wireless signals within a small local area called a *cell*. Coverage for large areas are provided by distributing base stations so signals can be received from any location a host may occupy in the region. In general, complete coverage creates overlapping cells.

Wireless Link. Communication to the mobile host from a given base station is provided through wireless transmissions, using either radio or infrared technology. These links can be very lossy and range from 9600 kbps to 1-2 Mbps.

Wired Network. The conventional network that connects wireless hosts to greater resources provided by stationary hosts and peripherals. Wired links are much faster and more reliable than wireless links.

The software components of a mobile system execute on the mobile hosts, stationary hosts, and base stations. The most common components are:

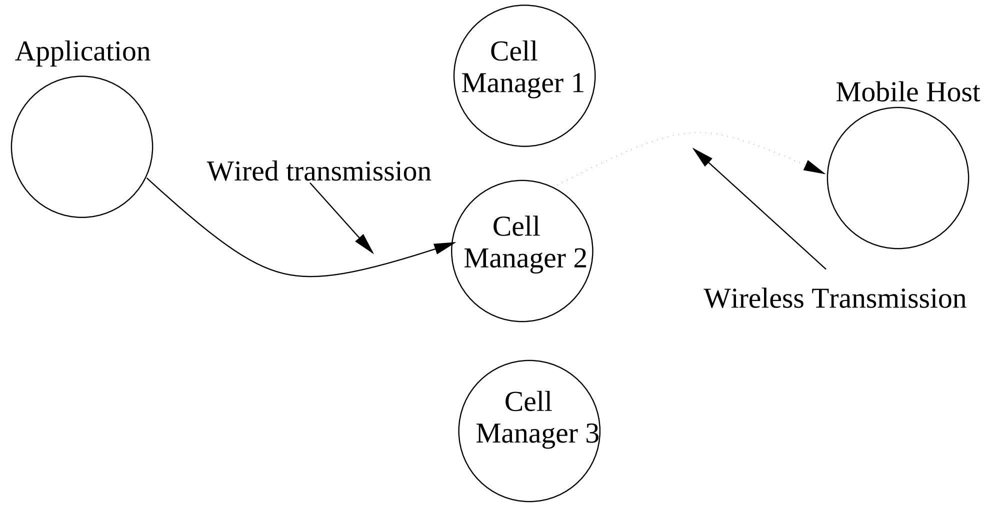

Figure 2.2: Software components of a mobile distributed systems.

Cell Manager. Executes on the base station or on a stationary host controlling the base station. Manages the wireless communication within the cell, forwards routing information to other cell managers, and controls traffic between wired and wireless networks (Figure 2.2).

Application. Executes on stationary hosts and communicates with mobile hosts through cell managers and perhaps agents.

Agent. Some architectures maintain an *agent process* that controls application connections to a mobile host and provides an indirect address for mobile hosts (Figure 2.3). The proxy or agent for a given mobile host is a stationery process that is responsible for delivering messages to mobile hosts and caching information about the mobile host's movements received from base stations.

What follows are examples of representative systems and how they solve routing and *cell handoff*, that is, switching a mobile host between two base stations. These systems were custom built and designed for their specific hardware infrastructure. The variety of systems is due to architectural accommodation and semantics. Architectural accommodation is necessary because mobile hosts have different capabilities, so different systems have emerged for hosts that are dumb terminals versus hosts that can function as autonomous entities. Protocol requirements such as TCP or IP, and fixed infrastructure such as wireless communication protocols or base station requirements also necessitate accommodation. Semantic variations exist to provide support for different mobile host applications, such as Web browsers, multimedia viewers, or portable patient monitor. The systems presented here are intended to illustrate the many directions of mobile computing

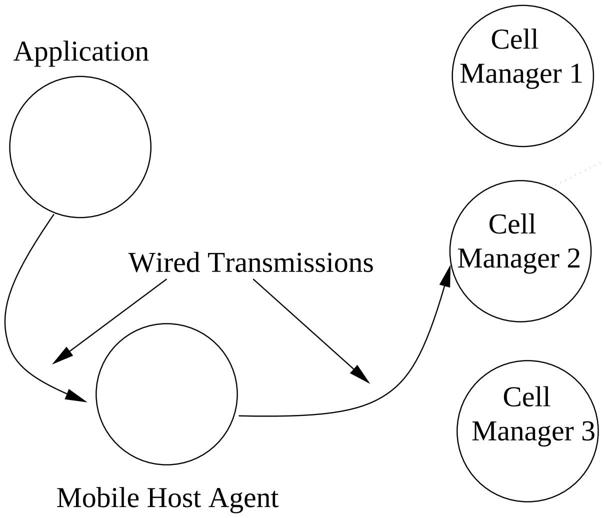

Figure 2.3: Agent-based mobile system.

research, with a special focus on cell handoff and quality of service.

2.3.2 Mobile IP

Several mobile computing protocols focus on routing messages to a mobile host anywhere in the Internet. Mobile IP is one such set of IP based protocols [IDJ91, Per96, IJ93] that handles routing and addressing of packets at the network layer. The architecture covers Intranet as well as Internet routing and delivery, and has the benefit of not affecting routers that do not wish to participate.

Mobile IP works by modifying the way IP addresses are assigned and implemented. Usually an address describes the network and subnet of the host, which is related to its physical network connection. In this approach, each mobile host is given an IP address that is in the mobile name space of a *virtual network*, which is made up of cells that are administered by Mobile Support Routers (MSRs). An MSR is a gateway to the rest of the wired network and a cell manager for one or more cells in the local area. Each cell has the same subnet number resulting in a mobile subnet that is comprised of many unconnected cells. Each MSR caches the current mobile hosts in the cell and MSRs for other hosts. Thus, when a stationary host wishes to send to a mobile host, it simply forwards the packet to the nearest MSR and the MSRs handle the delivery from there. Specifically, the following is done when an MSR receives a packet for delivery:

1. If the MH is present in the area managed by the MSR, forward the message to the appropriate base station.

2. If there is a forward pointer for the mobile host, forward the message to the other referenced MSR since the host has migrated.

3. If there is no MSR to which to forward the message, broadcast a request to all other MSRs asking that they attempt to locate the mobile host in their areas.

For this scheme to work, each mobile host must execute a handshake protocol with the MSR of any new cell it enters. To facilitate this, each MSR periodically transmits a beacon in its cell and every new mobile host responds with a greeting message that indicates its last MSR. The new MSR sends a forward pointer to the previous MSR so packets can be forwarded. Each MSR expires its entry for a mobile host if the host does not communicate again within a specified time interval, which is communicated to the mobile host as part of the handshake process. Forwarding the packet to the nearest MSR is handled by the normal Internet addresses advertised in the ordinary way with RIP, Hello, or IGRI [Hed88, Mil83].

Mobile IP also supports routing to mobile hosts that temporarily relocate to another part of the Internet that has Mobile IP support. When the host arrives at a network in a different mobile domain — i.e., a different subnet — it is referred to as a *pop-up* and it is assigned a temporary address known as a *Nonce*. The mobile host, with assistance from local MSRs, informs an MSR in its old domain of its new location and Nonce address. The old MSRs now forward packets for the mobile host through the Internet to the Nonce address to be delivered by local MSRs. Routing proceeds the same as before, except that MSRs in the mobile host's permanent subnet forward messages with the Nonce destination address.

Other IP protocols have been developed that provide similar functionality [Car92, CPR92, IDJ91, PB94, Rek93, TYT91, WYOT93].

2.3.3 InfoPad

The InfoPad is a mobile computing system specifically designed to support multimedia applications, including video, audio, pen, text and graphics [KMS+93, MSK+93, LSBR94, LBSR95, ABSK95]. The collection of mobile devices, called InfoPads, along with an indoor radio cellular network and wired backbone network form the *InfoNet*. The InfoPad has no general purpose computation resources so it serves only as a multimedia display and I/O device, with actual computation being performed by machines on the backbone. Communication to InfoPads is over a dedicated contention-free radio channel running at a data rate of 700 kbps. Communication originating from an InfoPad uses a contention-based control channel running at 244 kbps. It is typical to use the InfoPad as a multimedia data sink, therefore the asymmetry of the design supports large data streams to the InfoPad and not from the InfoPad to the backbone. Because of the multimedia requirements, quality of service is a primary consideration of this design. The InfoPad must control jitter for sound data and provide good throughput for video.

The software components that comprise the InfoNet can execute on many workstations and are similar to the generic system outlined above:

PadServer. For each InfoPad, a PadServer acts as the agent and runs on a workstation attached to the backbone. It is responsible for managing applications connected to the

InfoPad, controlling access to the InfoPad, and allocating InfoPad resources such as microphone, display, and speaker. It also negotiates quality of services requests on behalf of applications. This negotiation may occur several times as the pad moves from location to location crossing cell control boundaries. The InfoPad is unaware of location changes and does not participate in mobility and connection maintenance.

CellServer. The CellServer is the cell manager and controls the allocation of resources among the InfoPads in the cell. When a new InfoPad comes into the cell, the CellServer negotiates its quality of service parameters. CellServers can also negotiate with geographically neighboring cells to hand off InfoPads whose transmission quality has degraded and can be improved with a different cell connection.

Gateway. The Gateway is responsible for converting messages between the wired and wireless networks. It receives wireless messages and routes them to the stationery destination; it also receives messages from the wired network and transmits them through wireless transmissions to the InfoPad.

Network Controller. The Network Controller is a generalized name service that provides all parties in the InfoNet with name and address mapping information for InfoPads, PadServers, CellServers and Gateways. There is one Network Controller for the entire InfoNet.

Applications. Applications run on stationary workstations and are created with knowledge about their quality of service requirements, which can be characterized by the latency, jitter, and bandwidth characteristics of the data streams. The application must be able to adapt when all of the quality of service requirements cannot be met, e.g., by sending fewer video frames. The application communicates with PadServers and in some cases directly with the Gateway.

2.3.4 PARC TAB

The PARC TAB is an infrared-based communications network designed to operate in office building sized areas [STW93, AGSW93]. Named for the palm sized mobile host, the system is constructed as a collection of Unix processes providing reliable connections from applications to mobile hosts. The PARC TAB has very limited storage and compute power, and therefore is treated as a mobile terminal rather than a standalone machine. As a result, most of the actual computation is done by application processes that execute on stationary resource-rich Unix workstations. Routing is agent-based, so applications send messages to agents of mobile hosts and the agent routes the message to the current base station in contact with the mobile host. Each base station is controlled by a Unix process and forwards detection of a mobile host to the host's agent. Agents use this information to forward messages destined for the mobile host. Traffic originating at the mobile host is easier to route since it is all directed to the stationary agent. The agent then forwards the message to the appropriate application process. In general, the agent sends PARC TAB responses to the application, while the application sends screen updates and receives touch pad and button press events from the PARC TAB. Thus, the user perceives the

PARC TAB as running applications when in fact the application is really executing on a workstation somewhere in the building.

2.3.5 Dataman

Dataman is a mobile Web browser designed to be used while roaming a campus [BB95, IBar, IB93, AB93, ABI93, BAI93a, BAI93b, BBIM93]. Although the Web is its targeted application, the architecture is general and supports a range of applications. The main goal of the Dataman architecture is to allow applications that are TCP/IP based to function with mobile hosts; the application cannot detect whether the TCP/IP connection is with a stationary or mobile host. The Dataman architecture also supports location independence so the host can move and transparently access resources from the local area. For example, a Web browser can get the next page (hyper-link) from the nearest server.

The architecture uses Mobile IP for addressing mobile hosts and modifies TCP/IP so the mobile side of the connection appears stationery by using a stationary intermediate host. The modified version of TCP/IP is Indirect-TCP/IP (I-TCP/IP). Mobile IP is used to make connections from the stationary machine to the stationary MSR (Mobile Support Router), which administers a cell with a base station (Section 2.3.2). The MSR then runs an I-TCP connection to the mobile host. The MSR "fakes" the mobile host side of the TCP/IP connection so that it appears stable and non-mobile. I-TCP/IP requires applications to check that all packets are delivered, unlike TCP/IP; hence, it depends on end-to-end application layer reliability guarantees. Other changes to TCP/IP are that, when a connection migrates during cell handoffs, the slow start timer is reset so that the new MSR has time to establish a connection before other packets are sent. Other versions of modified TCP for mobile computing are described in [BSAK95, ABSK95, BKPV95].

When the mobile hosts moves in Dataman, the mobile host requests that the new MSR ask the previous MSR to migrate its I-TCP connection. The migration includes the TCP/IP connection, sockets, protocol numbers, and any buffers to be received or sent.

2.3.6 Crosspoint

Crosspoint is a campus-wide wireless mobile network designed to enable students to maintain connectivity while roaming freely within a campus [CR94, CLR95]. The system's primary goal is to have enough aggregate bandwidth to handle massive synchronized movements of hosts as students change classes. In addition to handling huge volumes of mobility information, a secondary goal is that the architecture not require any modification to network software within routers. The Crosspoint architecture has a fast ATM switching fabric for fast interconnect between base stations and routers connected to the regular campus internet. The ATM switching network provides two virtual circuits between each base station: a high priority control channel and a lower priority data channel. The communication between routers and base stations is unique but the rest of the architecture is typical; base stations transmit and receive packets transmitted over the wireless medium and then route those packets by an ATM switching network to their destination. Each base station informs all others about what hosts are in its area. Mobile hosts communicate with each other by the base station picking up the wireless signals and routing to another base station that transmits to the other mobile hosts. A packet headed for the wired network is routed through the ATM switching fabric to the routers controlling access to the campus internet.

2.4 Modular Protocols

2.4.1 Overview

Most networking protocols have traditionally been large complex software systems, which has made them difficult to debug, extend, and modify. Originally, these systems were monolithic implementations for performance reasons, but now there is increasing recognition that modular implementations can be competitive in this area. One way to create a modular implementation with good performance is to implement services as software modules, which are then optimized into a monolithic executable using compiler technology [AP93]. This section presents systems for constructing protocols that are implemented and executed as modular code structures. We also discuss modular systems that regard modularity as an opportunity to improve runtime performance. In particular, such systems execute protocol modules in parallel using the modular structure as a framework for parallelization.

2.4.2 The *x*-kernel

The *x*-kernel is a system for composing protocol modules that facilitates experimentation with communications systems. It provides a "protocol backplane" for protocol compatibility and interoperability. Each protocol supports a common set of operations that form a uniform protocol interface (UPI): push, pop, and demux of messages between protocols. The UPI supports the construction of protocols that can be hierarchically composed in a protocol graph. The *x*-kernel has a thread per message architecture. That is, a thread is created for each message to shepherd the message through the protocol graph, executing the code at each layer. Most common protocols are available from an extensive library of implemented protocols.

Messages are the main mechanism for communication and information sharing between protocols. There are also limited control operations that allow execution of an arbitrary operation by another protocol. The *x*-kernel provides efficient services for typical networking protocol operations, such as message assembly, fragmentation, and header additions and deletions. Mapping utilities are also provided for associating keys with data and looking them up.

2.4.3 Horus

Horus provides applications with configurable communication support by allowing users to combine individual layers in a protocol stack to achieve some desired overall functionality [vRBF+95, vRBG+95, vRHB94, RBM96]. Horus is designed primarily for building communication services for fault-tolerant systems. For example, a configurable implementation of the ISIS system described in Section 2.1.3 has been built using Horus. Protocol objects in Horus are assembled into a stack at runtime. First, complex services are decomposed into simple protocols, where each is written using a common set of upcalls and downcalls termed the Horus Common Protocol Interface (HCPI). This interface supports the same operations as the x-kernel UPI plus additional calls, for example, to join a group, merge views, and send a message to a subset of members. Each layer is normally written in ML, but can also be written in C to improve performance. There is a library of 20 common protocols, each one providing a particular communication feature.

To facilitate implementation, Horus provides a standard set of objects for protocol writers, including endpoints, groups, messages and threads. An endpoint represents a communicating entity and has an address used for membership. Endpoints can send and receive messages, although messages are not addressed to endpoints but rather to group objects. A process can have multiple endpoints and each stack of protocols has an endpoint. Group objects maintain the local protocol state of an endpoint and a view of the group membership. The Horus message tool supports pushing and popping of headers, similar to the x-kernel. Threads perform computation and are not bound to any particular endpoint, group, or message. A process can contain multiple threads, which are created when a message arrives by another thread or by a timer. Threads are executed concurrently and run preemptively. Protocols are designed for multiprocessing so they are asynchronous and re-entrant.

2.4.4 ADAPTIVE

ADAPTIVE (A Dynamically Assembled Protocol Transformation, Integration, and eValuation Environment) is a flexible transport environment for developing protocols of diverse quality of service requirements running on high-performance networks [SBS93, SS94]. The main benefits of ADAPTIVE are customized lightweight sessions and alternative process architectures for parallel processing. Sessions are the state of a connection and contain roundtrip timers, local and remote addresses, sequence numbers, and flow control window advertisements. Instead of incorporating complete functionality in a single protocol, the transport layer is created from several smaller protocols that are customized for the application's needs. To facilitate this process ADAPTIVE provides lightweight sessions that can automatically be configured to create a system on shared-memory or message passing multiprocessors. Configuration can be done either at compile time or at runtime; in the latter case intermediate switching nodes are used to determine if the application's quality of service requirements can be met [SS94]. The system also has performance monitoring and data collection to analyze the performance of different configurations. The data collection can also be used to adjust the system behavior dynamically, which makes it ideal as a platform for adaptive systems.

2.4.5 Object-Oriented Framework

An approach to creating middleware protocols to support application-specific configurations has been described in [Gol92, GL93]. This approach is based on a *framework*, which is an object-oriented description of the components of a system. The framework is specifically built as an environment for group communication such as would be needed for coordination of replicated objects in a distributed system. The system has component objects that can be included or left out depending on the guarantees required. The various components can also have different implementations for similar services, such as different ordering implementations or different membership support. The framework includes a number of predefined data structures, including a message log, message summary information, and a group view.

2.4.6 Streams

System V Streams supports modularization of protocols using a hierarchical composition model [Rit84]. In this system, an *I/O stream* consists of several modules linearly linked together, where each module has an identical read/write interface to facilitate interchangeability. A module includes queues of incoming and outgoing message blocks, a put routine for queuing, and a service routine to perform the module's operation. The runtime system manages queue flow control by managing the scheduling of the service routine. The modules only communicate by sending message blocks, either data or control.

The pipeline of modules can be extended by pushing a module on or popping a module off. This facility was originally envisioned for terminal I/O, where a module can be pushed on for one type of terminal driver or network device. System V Streams has also been used to provide interprocess communication by using PT (Pseudo Terminals) to link Streams on two hosts.

2.4.7 Parallel Protocol Execution

The emphasis of this project is on developing modular implementations of protocols to improve performance and configurability of communication support for gigabit/second networks [MS93, LAKS93]. Faster performance is achieved by executing the individual modules on the individual processing units of a parallel machine. In addition to speed requirements, these protocols send voice and video transmissions with image and data encryption, which presents even further protocol processing requirements. The fine-grained modularity of these protocols allows communication services to be configured to match the transmission, encryption, and compression requirements of the application.

A protocol is created from a collection of fine-grained protocol *objects* that perform isolated processing tasks. By dividing the tasks to be performed, a processor can be assigned to each task to gain the advantages of parallel execution; protocols can process incoming and outgoing packets concurrently. Protocol objects communicate by sending messages and can coordinate their actions in a more flexible manner than traditional protocol architectures. In addition to protocols sharing information using asynchronous messages, synchronous communication is also possible by executing a method in another protocol object.

The contributions of this project are parallelism applied to protocols and the ability to configure the protocol objects to match application requirements. To achieve this performance, the layered protocol model of communication services is violated to increase parallelism by adopting a more free-form layered model.

2.4.8 Parallel versions of the x-kernel

A parallel implementation of the x-kernel that runs on Silicon Graphics shared memory multiprocessors is described in [NYKT94]. In this system, *packet-level parallelism* is used in which packets can be processed on any processor. To make the x-kernel multiprocessor safe, locks were added to routines that access x-kernel data structures. The basic x-kernel structuring of protocols objects remains the same, but each object now needs to be concerned about protecting its data structures from concurrently executing threads. The system was built and measured using a TCP/IP stack, which shows performance improvements and good scalability A separate and similar parallel implementation of the x-kernel is described in [Bjo93].

2.4.9 Parallel Protocol Framework

The Parallel Protocol Framework (PPF) defines a hierarchical implementation and parallel execution environment for protocols [GNI92]. The PPF provides many efficient primitives so the protocols can be written more consistently and easily. Events are used for communication between layers; in particular, protocols communicate by posting events to another protocol. The receiving protocol is explicitly identified when the event is generated and only one receiver is permitted. Protocols may also communicate with other protocols in the same layer using events. The events can be ordered by sequence numbers to prevent messages from being delivered out of order in a particular connection due to parallel execution. For non-connection oriented protocols, events can be executed with no ordering restrictions. This event interface between protocols supports some interchange of protocols, although protocols must have the same interface and expect identical arguments. Like the x-kernel, the PPF supports a hierarchical graph of protocols. Also like the x-kernel, each message is shepherded through the protocol graph, executing each protocol.

2.5 Configurable Operating Systems

A recent trend in operating systems (OS) is towards configurable systems that place traditional OS functionality under application control to improve performance. Many studies have shown that when OS policies are not well matched to the application, poor performance can result. For example, the file system requirements of Web applications, compilers, and scientific computations are very different so it is unlikely that one file caching policy will give peak performance for all three. This section describes several operating systems that give control over many policies of this type to applications.

2.5.1 Spin

The Spin operating system allows OS services to be tailored to specific applications. The focus of Spin is on extensibility, safety, and efficiency [BCE+95]:

- *Extensibility.* The system provides fine-grained access to system resources and functions. Extensions are dynamically linked into the kernel virtual address space and protection domain. Using this mechanism, users can augment the memory management system, scheduling, and network subsystems. For example, the system can be configured with a different page replacement algorithm specially tuned for a particular database application.

- *Safety.* Applications can install new policies, but these should not affect other applications. The extension mechanism contains the effects of different extensions by using language features to enforce type safety and logical protection domains to manage processes' control over resources. These protection domains can be disjoint or overlapping to provide sharing between processes.

- *Efficiency.* The system is extensible, but not at the cost of performance. To provide an efficient system, events are used for structuring. In particular, applications provide handlers for events triggered by the system, which allows user code to extend OS behavior. To reduce event overhead, handlers are invoked using procedure calls; since handler code is executed in the kernel address space and protection domain, no user/kernel boundary crossings are needed.

2.5.2 Exokernel

The Exokernel is a customizable operating system that provides opportunities for domain-specific optimization through extending, specializing, or even replacing object-oriented libraries [EKO95]. The Exokernel is a micro-kernel that allows untrusted software running in user space to implement normal OS functions, such as virtual memory and interprocess communication. Most applications use one of a handful of available library OS with popular interfaces, e.g. POSIX, but they can also create their own customized versions. Like Spin, the Exokernel is concerned with performance, so it allows the OS libraries to access the hardware directly. User processes can directly access hardware in secure ways through capabilities granted by the kernel, with access being revoked if a process misbehaves. The system implements secure bindings for capabilities that cannot be forged by another process so the kernel does not have to check every access. The OS library code is downloaded into the kernel for execution to avoid the cost of crossing the user/kernel boundary.

2.5.3 Scout

Scout is a configurable communications-oriented operating system targeted to supporting "information appliances" rather than just computation [MMO$^+$94, MP96]. The specialized tasks performed by such systems are implemented as customized software, which allows the use of inexpensive commodity components. To facilitate development, Scout provides a toolkit for configuring modules required by the application. Scout employs specialized compiler techniques to optimize predictable execution of OS code to increase instruction cache hits.

Quality of service is also a concern in this type of OS, so Scout is organized around the idea of a *path*. A path is an extension of a network connection into the host operating system from data source to data sink. Resources (CPU, memory buffers, I/O Bus, cache, TLB) are allocated based on the quality of service requirements of a particular path. Scheduling in the OS is based on the path and not on threads. Scout does not enforce a particular quality of service model but rather provides the mechanisms to support a variety of policies.

2.5.4 Application Controlled File Caching

Application controlled file caching is designed to improve performance through customized caching policies [CFL94b, CFL94a]. The goals of the approach are never to perform worse than LRU and to prevent misbehaving processes from negatively impacting the performance of other processes sharing the file cache. The approach is as follows. First, when a cache miss occurs, the OS selects a process to give up a block. The process can select any block but if its decisions result in increased cache misses relative to what the kernel would have provided, the kernel re-assumes control. An application can also rely on any one of several replacement policies already implemented by the system (e.g., MRT, LRU) based on the usage pattern of files. Even with the extra overhead of extra crossings of the user/kernel boundary to consult the user process for block replacement, the improvements to file caching result in a reduction of block I/Os by as much as 80%. For applications that select a replacement policy already implemented by the kernel, no extra user/kernel boundary crossing are incurred.

2.6 Summary

Software for distributed systems can be simplified using communication services and abstractions such as multicast, membership, RPC, and various systems that support mobility. While all services of a particular type have the same basic functionality and structure, a variety of specific systems have been defined that specialize the semantics to match the needs of particular applications. A number of projects have developed modular approaches that allow a degree of customization, but the modules are relatively coarse grain and composition is constrained to be hierarchical.

CHAPTER 3
COMPOSITE PROTOCOL MODEL

3.1 Motivation and Goals

Early protocol systems were designed as monolithic entities, and their implementations reflected this. Even as the layered model gained acceptance as a conceptual tool to view protocol composition, implementations still tended to be ad hoc, reflecting a concern that implementing each protocol as a distinct entity would result in significant performance penalties. It is only recently, in fact, that software support for protocol composition has reached a level where hierarchical collections of protocol objects can be combined into a system whose performance is competitive with monolithic implementations.

Constructing a communication service from collections of protocol objects has a number of advantages. Perhaps the most important is that it allows, at least in theory, reuse to construct new services. In other words, a new service can be constructed by writing a new object that implements just the new aspect of the service, and then combining it with existing, well-tested objects that provide the other necessary functionality. Over time, a comprehensive library of objects can be developed, thereby simplifying the development effort, facilitating performance comparisons between protocol implementations, and allowing experimentation with new protocol concepts.

While hierarchical composition has worked well for a large class of protocols, a persuasive case can be made that it lacks the flexibility needed to implement certain types of protocols. For example, in designing and implementing Consul using the *x*-kernel, a number of inherent problems with the model were discovered [MPS93b]. These problems can be summarized briefly as follows:

- Provisions for communicating between protocol objects on the same machine are insufficient to implement the necessary complex interactions. In the *x*-kernel, the specific problem is that the Uniform Protocol Interface (UPI) lacks sufficient flexibility, thus requiring the programmer to use control operations as a workaround.

- Lack of communication support leads to implicit dependencies between objects, where one object "expects" another to realize some functionality. When compared to an explicit dependency caused by an invocation, implicit dependencies make the software difficult to debug and modify.

- A protocol object may need to store and examine multiple messages at a time to implement, for example, message ordering properties. Such a processing paradigm differs from traditional protocol objects, which typically deal with a single message at a time.

- Multiple protocol objects may need to coordinate their actions or synchronize relative to a given message or set of messages. Such coordination is difficult in the current model.

A remarkably similar experience has been reported independently by the developers of xAMP [Fon94].

While these limitations are directly relevant only to atomic multicast protocols like Consul and xAMP, there are several reasons to believe the lessons are applicable to other types of protocols as well. First, increasingly sophisticated services are being implemented as network protocols, in part because of the advent of protocol-oriented kernels such as the *x*-kernel. These services, like atomic multicast, are the type most likely to stretch or break the hierarchical model. Second, as distributed applications become more common, the demand for new types of specialized protocols very different from current protocols will increase. Doing such specialization in a hierarchical model—especially fine-grained specialization—is likely to be difficult. Finally, applications are demanding more control over their execution environment, including the communication substrate, in order to achieve the best possible performance. Such configurability will further increase the complexity and variety of protocols that must be supported.

This research is based on the premise that the construction of network services through the composition of protocol objects is the appropriate paradigm. Our objective, however, is to relax the restrictions on intra-machine inter-object communication imposed by the hierarchical approach. In our approach, protocol objects performing unrelated tasks are located in different layers and communicate normally using the standard UPI of the *x*-kernel. However, protocol objects that need to communicate more often or cooperate more fully—our micro-protocols—are co-located within a structure that provides richer facilities for this type of interaction. Micro-protocols have no direct knowledge of each other; communication is achieved indirectly through an event mechanism. This structure, described in detail in the next section, has a number of benefits, including:

- *Expressibility.* The micro-protocol execution environment provides a new, more general model for structuring protocol objects. Micro-protocols can communicate with an arbitrary number of other micro-protocols, can synchronize when necessary, and can operate on collections of messages. The environment also supports multiple threads of execution.

- *Configurability.* A network service is constructed out of modular micro-protocols, each of which implements a specific semantic property. The result is an approach that supports a high degree of configurability and the construction of services that are customized to the needs of the application.

- *Efficiency.* Since a network service can be customized, the application avoids execution overhead that can result from the inclusion of unnecessary properties. For example, it is easy to build an atomic multicast that includes no consistent ordering of messages, thereby avoiding the delay inherent in doing such ordering.

- *Resuability.* Micro-protocols implementing various semantics can be used in multiple services. For example, a liveness micro-protocol that checks that all processes have sent a message within some given time interval can be used in a variety of protocol suites.

- *Ease of debugging and maintenance.* Since a service is constructed from small micro-protocols, each can be debugged and maintained independently. This process is also simplified since interactions between micro-protocols are largely explicit.
- *Explicit dependencies.* Dependencies between micro-protocols are explicit since "back door" communication channels are unnecessary. This makes understanding the micro-protocols easier and the interactions obvious.
- *Future opportunities for optimization.* Explicit dependencies create the potential for code optimization. For example, it may be possible to in-line code using techniques similar to [AP93] to yield a system with efficiency competitive to monolithic implementations.
- *Availability of x-kernel protocols.* Since our system is incorporated in the *x*-kernel, all existing and future *x*-kernel protocols can be used without modification.

In summary, then, our goal is to extend current technology to encompass more fine-grained composition of protocol objects, both to simplify development and to increase the configurability of the network subsystem. Note that a second prototype implementation of the model has been constructed in C++ [Hil96].

3.2 A Two-Level Model of Protocol Composition

In the standard *x*-kernel model, a hierarchical graph of protocol objects is used to realize a communication service. A thread shepherds each message along a path through the graph executing the *x*-kernel operations `call`, `push`, `pop`, and `demux` to route the message on the correct path from the application to the network or vice versa. Messages can be modified, destroyed or created as they traverse the graph.

In addition to processing application messages, a protocol object uses messages to communicate with other protocol objects to which it is connected in the graph. Since this graph is hierarchical, however, communication flexibility is limited, especially with regard to allowing communication among protocol objects at the same level of the graph. Thus, our scheme augments this model by adding composite protocols, which essentially create new ways for protocol objects at the same level to communicate. In addition, we have extended the one-thread-per-message model to multiple-threads-per-message model and provided an event-driven mechanism for protocol communication.

In our model, the standard *x*-kernel hierarchical model is augmented with the ability to include composite protocols in the protocol graph in conjunction with simple *x*-kernel protocols. Unlike simple protocols, each composite protocol has an internal structure formed of a collection of micro-protocols executed in an event-driven manner. The major components of a composite protocol are:

- *Micro-protocols*: A section of code that implements a single well-defined property or provides some specific functionality. Consists of header information, private data,

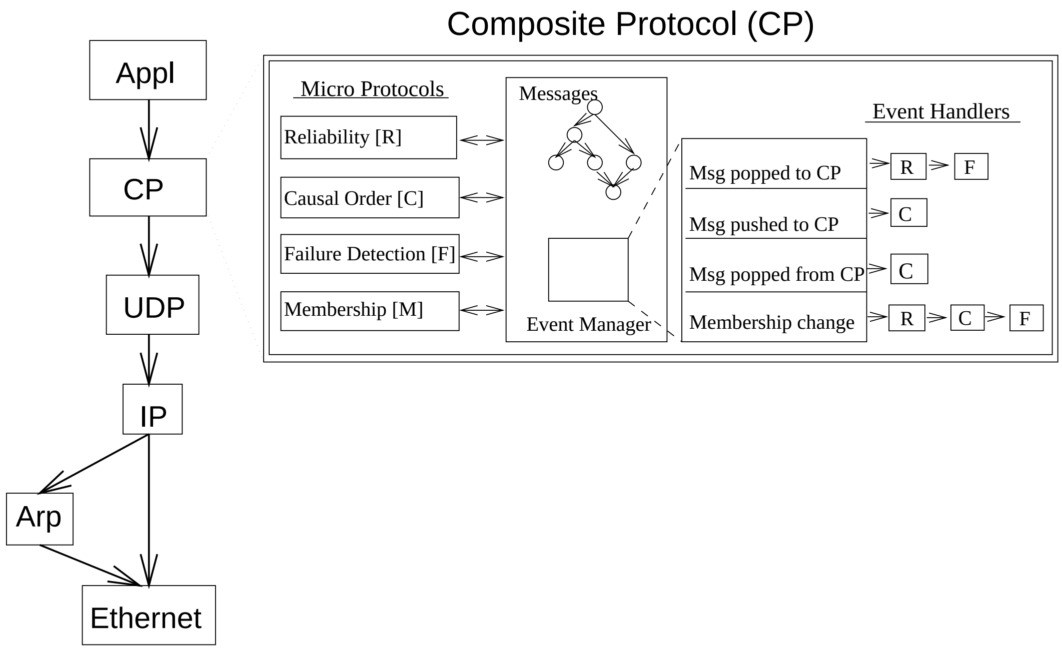

Figure 3.1: Composite protocol within an x-kernel protocol graph.

initialization code, and a collection of event handlers. May export data for use by other micro-protocols.

- *Events*: An occurrence that causes one or more micro-protocols to be invoked. Event handlers are invoked (logically) in parallel. Event types specify whether the triggering micro-protocol is blocked until completion or not. Some events of interest are predefined and generated by the framework (e.g., message arrival); others are defined by micro-protocols (e.g., change in group membership).

- *Framework*: A runtime system that implements the event registration and triggering mechanism, and contains shared data (e.g., messages) that can be accessed by more than one micro-protocol.

An example of this model is shown in Figure 3.1. To the left is an x-kernel protocol graph that contains a composite protocol CP implementing atomic multicast. To the right is an expanded view of CP illustrating the components of the model. In the middle of CP is the runtime framework, which contains a shared data structure—in this case a bag of messages—and some event definitions. The boxes to the left represent micro-protocols, while to the right are some common events with the list of micro-protocols that are to be invoked when the event occurs.

micro-protocol *name* {
  ... *Decl of exported events, message attributes,*
      data inspection, modification routines ...
  ... *Decl of imported events, global variables* ...
  ... *Decl of private events, message attributes, variables* ...
  ... *Initialization code* ...
  ... *Event handlers* ...
  ... *Local procedures* ...
} **end micro-protocol** *name*

Figure 3.2: Micro-protocol schema

3.3 Micro-Protocols

A micro-protocol is structured as a collection of export, import, and private declarations, and code for event handlers and local procedures, as shown in Figure 3.2. The export section lists procedures and events implemented and declared in this micro-protocol, but available for use elsewhere. The import section lists procedures and events that are provided by other micro-protocols. For brevity, events that are provided by the framework are not included in the import list since as they can be freely imported by any micro-protocol. Events and data that should not be accessed by another micro-protocol are listed in the private section. Micro-protocols use private events for internal communication. The initialization section contains statements that are executed at system start time to, for example, initialize private micro-protocol data. The next section contains the event handlers that make up the majority of micro-protocol code. Finally, the last section contains local procedures.

The general form of an event handler is:

$$\text{event name [\&\& boolean-expr]*} \quad \rightarrow \quad \text{handler}$$

Each handler is preceded by an event name and an optional boolean expression to make the handler execution conditional. This boolean expression or *guard* may reference event parameters, message attributes, and micro-protocol variables. When the event is raised, the guard is evaluated and the handler code executed if the result is *true*. [1] For example, a micro-protocol that resets timeout timers when an acknowledgment messages arrives might have a guard that checks that the message type is "acknowledgment".

Micro-protocols often manage data that can be exported by public data inspection routines. For example, a membership micro-protocol might export a routine that returns the current membership list. In these situations, only the micro-protocol declaring the data can alter it, so that changes by other micro-protocols must be requested by raising an event or calling an exported routine that modifies the data. When a micro-protocol modifies its data, it will often raise an event to notify other micro-protocols about the state change. For example, the membership micro-protocol might react to a "timeout" event

[1] Guarded events are not implemented in the prototype. Each event guard is translated by the protocol writer to an *if statement* at the beginning of the event handler.

by suspecting that a process has failed. If after further checking—for example, by running an agreement protocol with the other processes—it determines that a failure has indeed occurred, it would update the membership list and raise an event declaring a change to that list.

We express micro-protocols in an informal Protocol Description Language (PDL) that supports the structure of micro-protocol programming described above, and enforces visibility and modularity rules. Micro-protocols written in PDL are currently translated by hand into C files that are compiled using the standard C compiler. The intent of the language is to provide a common syntax for expressing micro-protocols that can then be translated into source code that is linked with the framework code to create composite protocols.

Other aspects of micro-protocols shown in Figure 3.2 (e.g., message attributes) are described below.

3.4 Events and Handler Execution

Events are a general communication mechanism used to inform micro-protocols that something of interest has happened. A micro-protocol requests notification from the runtime system for a given event by declaring a handler as shown above. Each event may have multiple handlers, and the handlers are not necessarily known to the micro-protocol raising the event. The latter property helps decouple micro-protocols from one another, thereby simplifying the task of writing micro-protocols that can be combined in a flexible fashion with other micro-protocols. For example, one micro-protocol can be responsible for detecting a situation, with another implementing the policy for resolving it. This type of structure allows the policies for each to be realized orthogonally based on the needs of the application and the specific collection of micro-protocols configured into the framework.

Events can also have parameters. For example, when an event corresponding to the expiration of an acknowledgment timer occurs, we might also want to communicate which message is lacking the acknowledgment. Such functionality can be realized by passing that information as an argument to the registered event handlers. All parameters are passed by value.

Events can either be user-defined or predefined by the runtime system. A user-defined event, such as the one related to timer expiration above, is exported (declared) by a given micro-protocol and explicitly raised by invoking a routine implemented by the framework. Predefined events, on the other hand, are exported by the runtime framework and implicitly raised when the framework detects that the event has occurred. In both cases, the event can be imported (handled) by any number of other micro-protocols.

The following list gives the predefined events currently supported; here, xMsg refers to an *x*-kernel message and CPMsg refers to a composite protocol message, both of which are described in more detail in Section 4.1:

- `Message_Popped_To_CP(xMsg)`: An *x*-kernel message from a lower-level *x*-kernel protocol has been popped to the composite protocol.
- `Message_Popped_From_CP(CPMsg)`: A message has been popped from the composite protocol to the *x*-kernel higher-level protocol.

- `Message_Pushed_To_CP(xMsg)`: An x-kernel message from a higher-level x-kernel protocol has been pushed to the composite protocol.
- `Message_Pushed_From_CP(CPMsg)`: A message has been pushed from the composite protocol to the x-kernel lower-level protocol.
- `Message_Inserted_Into_Bag(CPMsg)`: A message has been constructed and inserted into the shared bag of messages.
- `Message_Deleted_From_Bag(CPMsg)`: A message has been deleted from the shared bag of messages.
- `Message_Ready_To_Be_Sent(CPMsg)`: All micro-protocols are satisfied that the message can leave the composite protocol, either to be popped or pushed.

Handlers are scheduled for execution when an event is raised. If there are multiple handlers registered for that event, the order in which they are executed is indeterminate. In fact, they may be executed in parallel given the appropriate hardware. Dependencies between handlers are programmed explicitly using the event mechanism.

Execution of a micro-protocol that raises an event can either block until all handlers have completed (*synchronous*) or proceed without blocking (*asynchronous*). The choice of semantics is specified as an argument in the system call that raises an event, implying that it can vary on a per-invocation basis. These semantics extend as expected through multiple levels of recursively raised events.

3.5 Timer Events

Support is also provided for timer events that are generated after a specified amount of time has passed, rather than by any particular action of a micro-protocol or framework. Timer events are essential for detecting timeouts and performing periodic protocol functions. There are no predefined timer events, so all timer events are user-defined, and, like other events, value parameters can be passed to handlers. When a timer event is set, the user specifies the timer interval and if the event will repeat or be a single occurrence. When the specified timer interval expires, the framework asynchronously executes all handlers in parallel.

Unlike other events, timer events can be canceled. Cancellation is atomic with respect to handler execution; even if handlers have started to execute when the cancellation occurs, they are allowed to run to completion. When a single occurrence timer event is canceled, there are three possible event states: the event has not yet been triggered, the event has completed, or event handlers are currently running. If the event has not been triggered, then cancellation ensures that the event will never be raised. If all handlers for the event have completed execution, then cancellation has no effect. If the event has been triggered and handlers have started to execute, then all handlers are allowed to complete.

Cancellation of repeating events is similar to single occurrence events, but also prohibits any further occurrences of the event. As with single occurrence events, if the event has not been triggered, cancellation ensures that it will never trigger. If the event has already occurred but no handlers are currently executing, then no further occurrence will be triggered. If cancellation occurs during execution of the handlers associated with the event, then all handlers are allowed to terminate and further occurrences are prohibited. For both single occurrence and repeating events, a return value from the cancel function indicates if the timer event was successfully cancelled.

3.6 Framework

The framework is a runtime system that implements the event mechanism and provides a shared bag of messages on which micro-protocols operate. It also implements an x-kernel compliant interface for the composite protocol, which enables it to inter-operate with other x-kernel protocols in the standard way.

The framework accepts messages from the x-kernel and transfers control to micro-protocols by raising the appropriate events and executing the appropriate event handlers. As already noted, in the x-kernel, one thread shepherds any given message through the entire protocol graph, executing code in various protocol objects on behalf of the message. To handle the execution of potentially many event handlers, however, we extend this model to allow multiple threads to execute on behalf of the message during its residence in the composite protocol. This model provides more flexibility than the one thread per message in the context of composite protocols, and also allows the possibility of true parallel execution, as noted above. The one thread per message model is restored when a message leaves a composite protocol and is handed over to a standard x-kernel protocol object.

Messages that arrive at a composite protocol are placed in an unordered bag of messages maintained by the framework that functions as a global pool accessible to all micro-protocols. This feature is intended to support two aspects of programming that are common in the type of high-level protocols for which this approach is intended. First, it allows micro-protocols to make state changes based on information in an entire collection of messages, rather than just a single message as is typical in a hierarchical system. This can be important, for example, in an atomic multicast protocol that requires waiting for a collection of messages to arrive and then deterministically sorting the collection before presenting messages to higher levels [PBS89, MPS93a]. Second, a shared bag of messages allows multiple micro-protocols to access messages concurrently. This can be important, for example, in a situation where a message is acknowledged by one micro-protocol while concurrently being ordered relative to other messages by a second micro-protocol.

Prior to being placed in the bag, a *verify micro-protocol* is executed to determine if the message is acceptable. For instance, a message might be rejected if corruption is detected or if it is destined for a process that no longer exists. If the message is acceptable, the verifying micro-protocol places the message in the bag using a routine provided by the framework. The verify micro-protocol is written by the user, so that message screening and bag insertion are under program control; deletion from the bag is similarly done by the user. Commonly-used variants of verify can be supplied from a library, if desired.

Each message in the bag has a collection of *attributes* that encode certain types of per-message information. *Predefined attributes* are supplied by the framework. For example, one such attribute is direction, which indicates whether the message is being sent up or down the *x*-kernel graph. *Micro-protocol attributes* contain micro-protocol-specific information about the message. For example, a reliability protocol may keep private state information about the message indicating whether it was acknowledged or is being retransmitted and by which hosts. Such attributes can be declared either private or public; a private attribute is visible only to the micro-protocol that defines it, while a public attribute can be read by all micro-protocols. In addition, attributes are used to build headers for messages that are pushed from the framework. This is done by an attribute-to-header routine provided by the user and invoked by the framework as a message is exiting the composite protocol. Similarly, when a message is popped to the framework, a header-to-attribute mapping routine is invoked that unpacks the header and creates attributes using this information. Both of these mapping routines are currently supplied by the user, although it is easy to imagine generating them automatically from appropriate specifications.

As already noted, data defined within a micro-protocol can also be shared by exporting appropriate inspection routines. Any necessary synchronization within these routines is done explicitly using semaphores. With our prototype implementation, such synchronization is only necessary if the data is not written atomically and either a message push or an explicit event triggering is done in the middle of the code effecting the change.

3.7 Message Sending and Garbage Collection

In many cases, when to send a message up to the application or down to a lower-level protocol is a decision that cannot be made by one protocol alone, so coordinated sending is needed. For example, consider a message that has arrived from the network via a lower-level protocol. The acknowledgment protocol has dispatched a reply message to acknowledge receipt of the message, so it is completely satisfied that the message can be sent up to the application layer. However, an ordering protocol that places strict requirements on message ordering may wish to force the message to stay in the composite protocol. Realizing such coordination is especially difficult since it must function correctly for any combination of micro-protocols.

Determining when a message is "ready to send", then, can be a complex process involving multiple micro-protocols. The framework supports such coordination with the use of *send bits* associated with a message. There is one bit per micro-protocol, and when all send bits have been set, the framework automatically sends the message. If a micro-protocol does not need to restrict when a message can be sent, it sets its send bit by default. Often, this is done in the handler bound to the `Message_Inserted_Into_Bag(CPMsg)` event, which is always one of the first executed when a message arrives. Usually, the last send bit is set by the micro-protocol with the most restrictive conditions.

Similar to send bits are *deallocate bits*, which function to coordinate message deallocation. Again, each micro-protocol has a unique deallocate bit for each message. When all deallocate bits are set, the framework raises the `Message_Ready_To_Be_Deallocated(CPMsg)` event. The micro-protocols can then free any information related to the message. Note that if there are any outstanding send operations with references to the message, then deallocation is deferred until all such operations have completed. This is done to avoid pending send operations with references to deallocated memory.

An alternative to coordinated deallocation is a more centralized scheme that employs a micro-protocol that knows when it is safe to free messages based on other events. Some composite protocols naturally lend themselves to this approach. For example, once a message is sent or written to stable storage it can often be deallocated safely. Conversely, other micro-protocol suites may need to keep messages for retransmissions and hence, multiple micro-protocols could be involved. To provide a flexible environment, the framework accommodates both styles of garbage collection.

In addition to coordinated sending of messages, micro-protocols can also send messages without any other micro-protocol being informed. This is referred to as an *out of band* message because it is sent directly without using send bits or being inserted into the shared bag. Similarly, no events are triggered. Micro-protocols use out of band communication for sending control messages to peers without notifying other micro protocols in the composite protocol. Out of band messages are only supported for messages sent to the network via a lower-level protocol and not messages sent to the application. Sending of the message is synchronous and no `Message_Pushed_From_CP(CPMsg)` event is raised after the send operation is complete.

3.8 Examples

To illustrate the structure of micro-protocols and the event-driven programming paradigm, we present two short examples of micro-protocols that might be part of a suite used to implement an group communication service. The first is a simple membership micro-protocol that updates a membership list whenever a host is suspected of having failed. The second is an acknowledgment micro-protocol that sends an ACK message for each reply message received, sends a "still working" message to a client if the reply from the local server is slow, and raises the `Suspect_Host_Dead` event if a server is suspected to have failed. Both are written in PDL pseudo-code.

3.8.1 Membership Micro-Protocol

Figure 3.3 shows the code for the membership micro-protocol. At the top is an `exports` section that specifies inspection routines, events, and attributes that are exported for use by other micro-protocols. Here, an event for membership change and a routine for accessing the current group membership are provided. Note that the event specification includes a parameter to indicate whether the event of interest is the failure or recovery of a host. The exports are followed by an `imports` section, in this case an event corresponding to a suspected failure. This particular event is raised by the acknowledgment micro-protocol below and fielded by an event handler in `MEMBERSHIP`. Note that this specification also includes a parameter, specifically, an indication of which host is suspected to have failed. Next, the micro-protocol includes declarations for any private data, attributes, and events. In this case, the only private data is the membership list maintained by the micro-protocol.

```
micro-protocol MEMBERSHIP {
  exports{
    event   Membership_Change(ch_t type);
    proc    memList_t GetGroup();
  }
  imports{
    event   Suspect_Host_Dead(mem_t host);
  }
  private{
    memList_t  MemberList;
  }
  initialize{
    initMembershipList();
  }
  actions{
    Suspect_Host_Dead(mem_t host)   →
      if (find(host, MemberList)) {
        deleteMember(MemberList, host)
        raiseEvent(Membership_Change, DELETION, ASYNC)
      }
  }
  ... code for deleteMember, GetGroup, and initMembershipList ...
} end micro-protocol MEMBERSHIP
```

Figure 3.3: Simple membership micro-protocol

The declarations are followed by the procedures that make up the body of the micro-protocol. The first is an initialization routine, which initializes the membership list from some external source; for example, it may be read from a file. This routine is executed, in x-kernel terms, at initialization time prior to execution of the standard `open` or `openenable` routines.

After the initialization code is the `actions` section, which contains the event-handling code. In `MEMBERSHIP`, there is one handler that deletes a member from the list when the `Suspect_Host_Dead` event is triggered. The parameters to the event are available to the handler, as is any private data declared within the micro-protocol.

The remainder of the micro-protocol contains inspection routines for export, local procedures, etc. In this micro-protocol, there are three such routines: `deleteMember`, `GetGroup`, and `initMembershipList`. Their code is omitted here for simplicity.

3.8.2 Acknowledgment Micro-Protocol

Figure 3.4 shows the code for a simple acknowledgment micro-protocol `ACK` that generates the `Suspect_Host_Dead` event when a message has not been acknowledged after some interval of time. This interval can be adjusted by a call to the `setInterval` routine. The timer is started at the time the message is pushed from the composite protocol. The timer is set by the `setTimerEvent` call, which gives the interval to wait and an indication that this event is to be generated only once rather than periodically. This `Timeout` event is declared in the private section of the protocol and is therefore raised and handled only by

```
micro-protocol ACK {
  exports {
    event   Suspect_Host_Dead(mem_t host);
    proc    SetInterval(int millisec);
  }
  imports {
    boolean client, server;
  }
  private {
    event   Timeout(CP_Msg_t msg);
    attribute serverList_t servers ;
    int     interval;
  }
  initialize{
    InitTimerVal();
  }
  actions {
    /* Set timer event for each message sent to detect loss.*/
    Message_Pushed_From_CP(CP_Msg_t msg) && client &&
      msg.attr.type == REQUEST  →
      setTimerEvent(Timeout, CP_msg, interval, ONCE);

    /* Set timer event for request received so we reply in time.*/
    Message_Inserted_Into_Bag(CP_Msg_t msg) && server &&
      msg.attr.type == REQUEST  →
      setTimerEvent(_Timeout, CP_msg, interval, ONCE);

    /* Send an ACK message for each reply message received.*/
    Message_Inserted_Into_Bag(CP_Msg_t msg) &&
      msg.attr.type == REPLY  →
      sendAckToSender(msg,REPLY_RECEIVED);

    /* Send a "still working" message if server slow.*/
    Timeout(CP_Msg_t msg) && Server  →
      sendAckToSender(msg, STILL_WORKING );

    /* Some server has not responded in time.  */
    Timeout(CP_Msg_t msg, host) && Client  →
      if ( hostNotResponding(msg, host)) {
        RaiseEvent(Suspect_Host_Dead, host, ASYNC);
      }
  }

  ... code for SetInterval, InitTimerVal, sendAckToSender, and
        hostNotResponding ...

} end micro-protocol ACK
```

Figure 3.4: Simple acknowledgment micro-protocol

ACK.

The second set of tasks done by ACK involve acknowledging any messages that are received. It accomplishes this by handling the Message_Inserted_Into_Bag event for messages of type REPLY. The event is qualified so that only reply messages are acknowledged. Request messages are only acknowledged if the server is slow in responding, which is also handled using the Timeout event. The server and client sides of the communication are handled by the same micro-protocol, with the imported state variables server and client being used in the code to distinguish between the two.

3.9 Summary

Our approach to constructing configurable communication services is realized using the two-level protocol composition model. The first level is the *x*-kernel protocol graph, which defines the basic characteristics of the network subsystem using both composite and simple protocols. The second is the composite protocol, which defines the specific semantics of the relevant communication service using micro-protocols. The framework encapsulates the micro-protocols and supports event-driven micro-protocol interaction. Since the composite protocol exports the *x*-kernel UPI, it can be combined with existing *x*-kernel protocols, thereby making it easier to build new communication services on top of simpler existing networking protocols.

The micro-protocol structure and composite protocol model allow a protocol designer to create modular implementations of communication services. Each micro-protocol in the composite protocol implements a specific property or functionality, so the specific micro-protocols included govern the behavior of the resulting service. This approach allows applications to have fine-grained control over their communication support. The event-driven model also provides a novel execution paradigm and structured communication between micro-protocols.

CHAPTER 4
IMPLEMENTATION

Our prototype implementation of the framework is based on x-kernel version 3.2, which runs as a user-level task on Mach version MK82. Written in C, the prototype is structured as a collection of library routines that are linked with the user-written micro-protocols to create a composite protocol. The x-kernel, framework, and micro-protocols are compiled with gcc version 2.1. The composite protocol is then included in the x-kernel protocol graph in the normal way. The x-kernel was selected as the implementation environment because of its efficient message handling, novel thread execution architecture, ease of configuration and modification, and portability.

The primary test platform consists of DecStation 5000/240s connected by a 10 Mb Ethernet. These systems are based on MIPS R3000 micro-processor running at 40 MHz with a separate off-chip 64 KB instruction and data caches, and 16 MB of memory.

Here, we focus on describing the implementation details of the runtime framework since much of the system's functionality is implemented there. Initial performance results from a group RPC micro-protocol suite are given in Chapter 5.

4.1 Framework

4.1.1 Uniform Interfaces.

The framework encapsulates micro-protocols and delivers messages to and from other x-kernel protocols. Externally, the framework provides the standard x-kernel interface operations such as `call`, `push`, `pop`, and `demux`. This allows composite protocols to be added to an existing x-kernel protocol graph without requiring changes to the existing protocols. The framework can be configured to provide a synchronous call interface or an asynchronous push interface to accommodate both styles of x-kernel protocols. A call-style protocol is blocked when doing a call operation and is unblocked only after the reply message can be returned. If the push style is used, the caller is not blocked and the reply message (if any) is returned asynchronously.

4.1.2 Thread Management

As described in Chapter 3, multiple threads of control may be spawned in the course of executing event handlers. In the prototype, the x-kernel thread facility based on Mach C-threads is used as the underlying mechanism. The choice to use this facility rather than spawning C-threads directly was made for two reasons. The first is that this makes the threads visible to the x-kernel, which permits the programmer to use the built-in x-kernel features for doing execution monitoring and debugging, simplifying the programming process. The second is that it allows us to exploit the x-kernel's optimized thread management. In particular, the x-kernel preallocates a pool of C-threads at initialization time and manages them directly, which avoids the overhead of thread creation when an event is raised.

When an event is raised, a thread is allocated from this pool to execute each associated handler. The execution model is logically multi-threaded, so that multiple handlers—either associated with the same or different event occurrences—may in general be executed concurrently. No new threads are allocated for events that are executed synchronously; rather, the same thread that triggered the event will execute all handlers for the event one by one in the order specified in the event definition. Such semantics can simplify micro-protocol code in certain cases when execution order is important, such as when a subsequent event handler depends on a side effect caused by an event handler. However, this can lead to implicit dependencies between handlers so caution should be used. Section 7.2.3 contains further discussion of handler dependencies.

The protocol writer can choose to have event handler invocations be implemented by procedure calls rather than threads even in the case when the event is raised asynchronously. This optimization is targeted for sequential machines where a procedure call is typically more efficient than spawning a thread. No changes are required in the code for the micro-protocols. In fact, which version of the runtime is used is transparent to both the x-kernel and the protocol writer.

We also alter the x-kernel thread behavior by assuming control over a thread that enters the composite protocol. In general, it will execute some sequence of event handlers and then a `push` or `pop` to exit the composite protocol. Alternatively, it can simply terminate within the protocol after the last event has been handled. The thread behavior is naturally different depending on whether handler execution is implemented by threads or procedure calls. In the thread implementation, the thread that enters the composite protocol returns to the caller after raising the first event. Once the event is raised, other threads are activated to execute the handlers. On the other hand, with the procedure-based implementation, the entering thread executes each event handler until all handlers are executed (recursively) and then returns to the caller. Timing events are necessarily implemented as threads and are based on x-kernel timer events.

4.1.3 Messages

The composite protocol exists to receive, process, and send messages so it is unsurprising that the bag of messages is a centralized structure available to all micro-protocols. As described in Chapter 3, messages in the bag, referred to as CP Messages, are network messages augmented with additional attributes that micro-protocols use to share per-message data with each other. Since micro-protocols collectively process messages, the coordination of when a message is "finished" — ready to send or discard — is more complex than the layered model where only one protocol is in control of the message at any time. The framework provides for coordinated control of message attributes, creation of headers and attributes, sending, and deallocation.

CP Messages. CP messages are based on x-kernel messages, which optimize manipulations such as header pushes and pops, fragmentation, and assembly. The usual x-kernel message operations are supported, but we add additional information in the form of attributes that are efficiently accessed. The scope of private attribute names is limited to the micro-protocol in which they are declared, but public attributes must have globally unique names.

Bag of Messages. A CP message is a structure that contains an x-kernel message, attributes, and send bits. The attributes are created by combining the attribute declarations from all micro-protocols into a "super structure" of attributes.

The following operations are provided for manipulating the shared bag of messages:

- `CPMsg = newItem(xMsg, direction)`: Allocates and initializes a new CP message; returning a pointer to the appropriate structure. `direction` indicates if the message is traveling up or down through the x-kernel protocol graph when it enters the composite protocol.
- `insertItem(CPMsg)`: Inserts `CPMsg` into the bag. Automatically triggers the `Message_Inserted_Into_Bag` event.
- `deleteItem(CPMsg)`: Removes `CPMsg` from the bag, but does not deallocate storage for the item. Deallocation is done under micro-protocol control, although a message is usually deallocated as soon as it is deleted unless needed for retransmissions, etc. Automatically triggers the `Message_Deleted_From_Bag` event.
- `empty()`: Removes all messages in the bag.
- `n = count()`: Returns a count of the number of messages in the bag.
- `setSendBit(ProtocolID, CPMsg)`: When all micro-protocols have called `setSendBit` (i.e., all send bits are set) the `Message_Ready_To_Be_Sent` event is triggered.
- `setDeallocateBit(ProtocolID, CPMsg)`: Sets the deallocation bit for micro-protocol `ProtocolID`. When all bits are set (i.e., all micro-protocols have called `setDeallocateBit`), the `Message_Deallocate` event is triggered.
- `sprintItem(string, CPMsg)`: The current state of `CPMsg` (including attribute values) is placed into `string`. Used for debugging.
- `printBag()`: Prints the current contents of the bag to `stdout`. Useful for debugging.

Attributes and Headers. CP message attribute values are often derived from information contained in message headers, such as the sender id, destination id, and message type. To aid this function, all protocol suites are required to include a single `header_to_attribute` procedure that sets attributes based on header values and localizes header format knowledge to one procedure. Typically, this procedure is called by a verification micro-protocol after the incoming message has been validated. The `CPMsg` attributes are initialized and the message is inserted into the shared bag of messages so that other protocols can access message header information without knowledge of message header formats. Similarly, when a message is about to be sent, the message header is normally constructed from the attributes. The `attribute_to_header` procedure is called by the framework during a send message operation for this purpose.

Coordinated Sending of Messages. When a message is created with `newItem`, all the send bits are cleared. A micro-protocol sets its corresponding bit with the `setSendBit(CPMsg, ProtocolID)` procedure. Protocols distinguish their send bit by their unique protocol id that is assigned at initialization. When all send bits are set, the CP Message is ready to be sent and the `Message_Ready_To_Be_Sent` event is raised by the framework. The send bits restrict sending of messages both in the upward direction (to applications) and downward direction (to the network). If a micro-protocol is not directly involved in the decision when to sent a message, it normally would set its send bit when the message is inserted into the bag (i.e., when handling the `Message_Inserted_Into_Bag(CPMsg)` event). Typically, the last protocol to set its send bit has the strongest restrictions about when a message can be sent. For example, an ordering micro-protocol will not mark a message for sending until all predecessors of the message have been delivered to the application.

Sending Out of Band. Although coordinated sending is the expected norm, there are occasions when a particular micro-protocol might wish to send a message without another micro-protocol's interference or knowledge. This is accomplished with `sendMessageOutOfBand(CPMsg)`, which sends the message without raising any events.

Coordinated Deallocation of Messages. As described in Chapter 3, garbage collection in the composite protocol can be realized in one of two ways: either centralized into one micro-protocol or distributed among many protocols through the use of *deallocation bits*. Deallocation bits are very similar to send bits: for each CP message, there is one bit for each micro-protocol. In the distributed deallocation style, an unset deallocation bit indicates that a message is still in use by a micro-protocol. When all the deallocate bits have been set, the `Message_Ready_To_Be_Deallocate_Message(CPMsg)` event is raised. The handlers for this event perform the actual mechanics of message deallocation and deletion from the bag (i.e., calling `deleteItem()` and then freeing memory). The user chooses the style of deallocation support that is desired by setting a C preprocessor variable that activates the deallocation-based events and bits.

4.1.4 Implementation Portability.

The runtime framework relies almost entirely on facilities provided by the x-kernel. As a result, it is nearly automatically portable to another environment that has a working x-kernel implementation. The only non-x-kernel facilities that are used beyond normal C language library routines are three Mach functions for C-thread management: `cthread_yield` for assistance in cthread scheduling, `cthread_set_data` for associating data with a thread, and `cthread_data` for event execution management.

4.2 Events

This section describes the C structures and execution architecture used to implement events. Predefined events and user defined events, whether timer or regular, have the same structure and scheduling. All have a common event description structure that is initialized once with the handler functions, handler names, and the number of event parameters. This structure is passed as the event descriptor for every `raiseEvent` call. The structure is never modified after the event handlers are initialized. A second structure is allocated when the the event is raised that contains the parameter values for the current invocation. One invocation structure is created for each event handler of the triggered event. The structures remain the same for asynchronous and synchronous execution but the execution is performed differently. Timer events require additional structures to record the state of timer event and the current execution status to support cancellation and repetition.

4.2.1 Event Operations

The following operations are provided for manipulating events:

- `event = createEvent(eventName, numParams)`: Allocates and initializes a new `event_t` structure and returns a handle to the event that is used for later operations. `eventName` is a descriptive string naming the event, and `numParams` is the number of parameters that will be passed to handlers when the event is raised. Used for both regular and timer events.

- `addEventHandler(event, handler, handlerName)`: Appends a `handler` function pointer to the list of handlers for the event. The ordering of the add operations determines the execution order for sequential events. `handlerName` is a descriptive name for the handler, used for debugging and execution tracing.

- `deleteEventHandler(event, handler)`: Removes `handler` from the list of handlers for `event`.

- `raiseEvent(event, type, numParams, param1, param2, param3, param4, param5)`: Used by micro-protocols for triggering user-defined events, and by the framework for predefined events. An instance of the `event` is triggered and `numParams` are passed to handlers (maximum of five). All unused parameters values should be set to `NULL`. `type` is `SYNC` if the event is to be executed synchronously (blocking and sequentially executed), or `ASYNC` if executed asynchronously (non-blocking and handlers execute in any order). Note that the `raiseEvent` call determines how the event will be handled rather than the event definition.

- `event_invoke = setTimerEvent(event, type, interval, numParams, param1, param2, param3, param4, param5)`: Sets a timer event to execute after `interval` microseconds have elapsed. `type` determines if this timer event will schedule itself to repeat or execute once only. Handlers are executed concurrently for asynchronous execution. `numParams` indicates how many parameters are passed. The `event_invoke` is a timer event handle that is needed for the cancel and detach functions.

- `outcome = cancelTimerEvent(event_invoke)`: Cancels a timer event. Return value of `outcome` indicates:

  - `unknown`: No such event is known (bad handle value); the request is ignored.

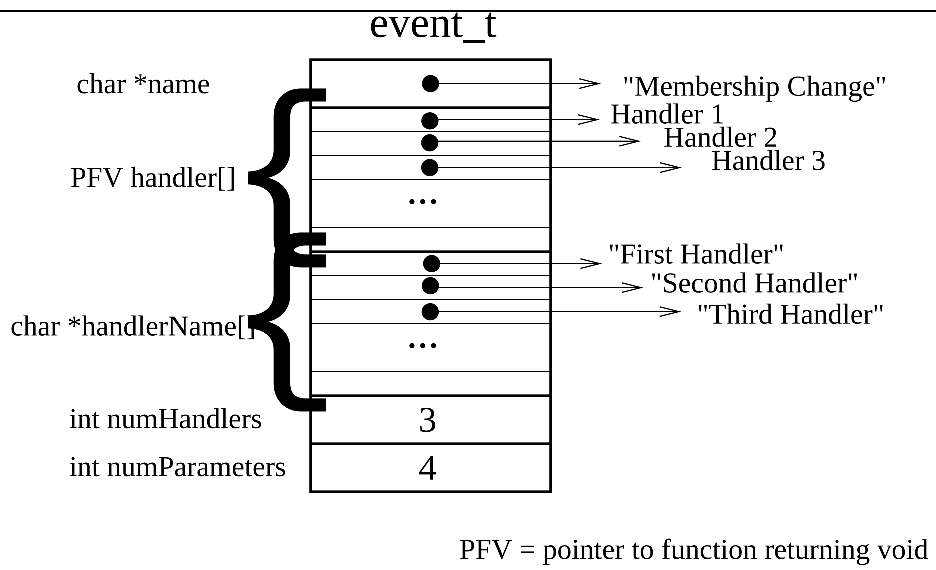

Figure 4.1: Event description structure.

- `started`: The event has already started to execute so if it was a once-only event, the timer event will run to completion; cancelling will have no effect. A repeating event will not reschedule itself but the current execution will be allowed to terminate normally.
- `completed`: The event has completed and so there is no instance to cancel. This can occur only for once-only events.
- `cancelled`: The event has already been cancelled. This cancel request will have no effect.
- `successful`: The event has successfully been cancelled.

- `detachTimerEvent(event_invoke)`: Since timer event handles are returned from the `setTimerEvent` function, the micro-protocol must indicate when the `event_invoke` structure can be deallocated. A detach call on an event will cause the timer event to deallocate structures when the event completes. If a timer event will never be cancelled, then the timer can be set and immediately detached. Otherwise, the event should be detached after cancellation or completion.

4.2.2 Event Structures

Predefined events are declared and raised by the framework, while user-defined events are created with `createEvent(eventName, numParams)`, which returns an allocated and initialized event structure of type `event_t` (Figure 4.1). Micro-protocols create events and assign handlers to events during their initialization. Handlers are registered for events through the `addEventHandler(event, handler, handlerName)` procedure. The order in which handlers are added to the event specifies the execution order for sequential events. Figure 4.1 shows an example event named “membership change” with three handlers and four parameters. The `event_t` structure is static since it is a description of the event and does not change. Note that the parameters are not stored within this structure; they are stored in the `invoke_t` structure, which is automatically created for an individual occurrence of an event. This structure is shown in Figure 4.2. Each `invoke_t` structure has a reference to the event structure, which is essential for timing events and debugging support. The invocation structure is automatically deallocated when the framework detects that all event handlers have terminated.

The *x*-kernel does not provide general purpose thread support, so the framework creates threads by scheduling *x*-kernel timing events to execute with 0 seconds delay. These *x*-kernel events are, in turn implemented with C-threads. This is the only place where the implementation of the *x*-kernel is explicitly used by the framework. The *x*-kernel timing event implementation makes the composite protocol threads visible to *x*-kernel debugging tools, and also reduces the context switch time since the threads are allocated from a pool created at initialization time. Specifically, when a `raiseEvent` procedure triggers an event, the timing event is scheduled, which upon expiration places the allocated thread on the ready list to be scheduled with other C-threads.

The *x*-kernel timer events accept a function pointer and one parameter. Since framework events can have multiple handlers and parameters, the `invoke_t` structure containing the handler function pointer and parameters is passed as the single *x*-kernel timer event parameter. The *x*-kernel event is passed a "super handler" procedure to execute when the timer expires. The super handler unpacks parameters from the `invoke_t` structure and passes them individually to the handler. Thus, the super handler acts as a procedural wrapper around handler executions, recording the start and termination of handler execution.

To maintain uniformity, synchronous event execution uses the same `invoke_t` structure to pass parameters. However, no *x*-kernel event is used to schedule execution in this case. Instead, the super handler is invoked directly as a procedure, which provides a synchronous call style with blocking semantics. The super handler is the same as in the asynchronous case, and, in fact, is unaware whether it was called from an *x*-kernel event or directly as a procedure. As mentioned earlier, the framework allows for asynchronous execution to be optimized as procedure calls. When the user defines the `PROC_CALLS_ONLY` C preprocessor variable, asynchronous event triggers are also executed with procedure calls.

4.2.3 Timer Event Structures

Timer events, since they can be repeating or cancelled, have an information structure that contains the state of the execution. This structure is created by the `setTimerEvent` function and passed back as the timer event handle that is used in cancellation. It has information about whether the handlers for the event are waiting to execute, started to execute, or completed execution.

Timing events are handled as asynchronous invocations that are scheduled to execute with a delay. The same event description structure, event invocation structure, and super handler is used. A repeating event will schedule itself again after the last handler has finished executing. However, since timing events can be cancelled, an additional structure is needed to cancel the underlying *x*-kernel events. Also, timing event cancellation must

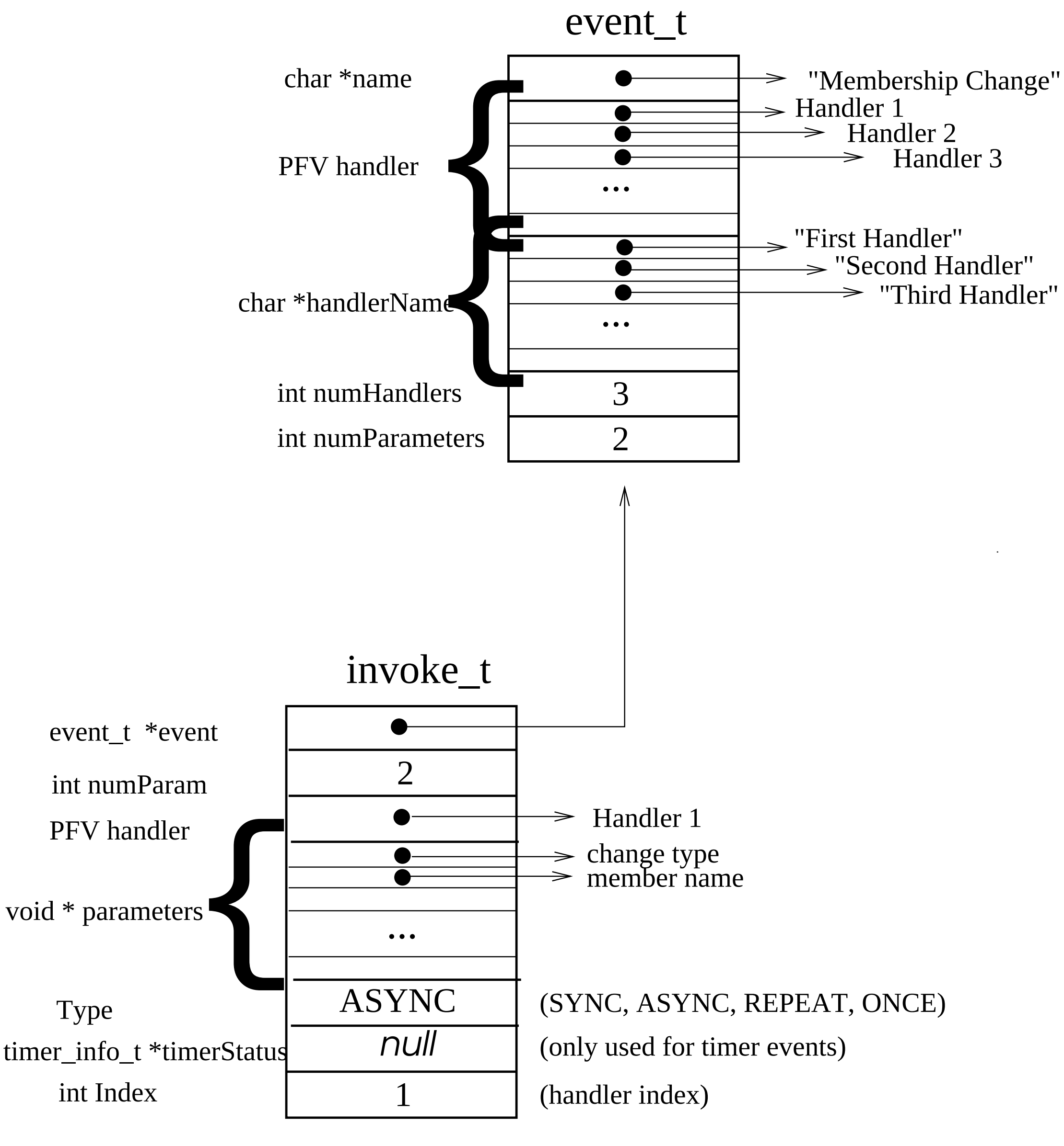

Figure 4.2: Event invocation structure with event description structure.

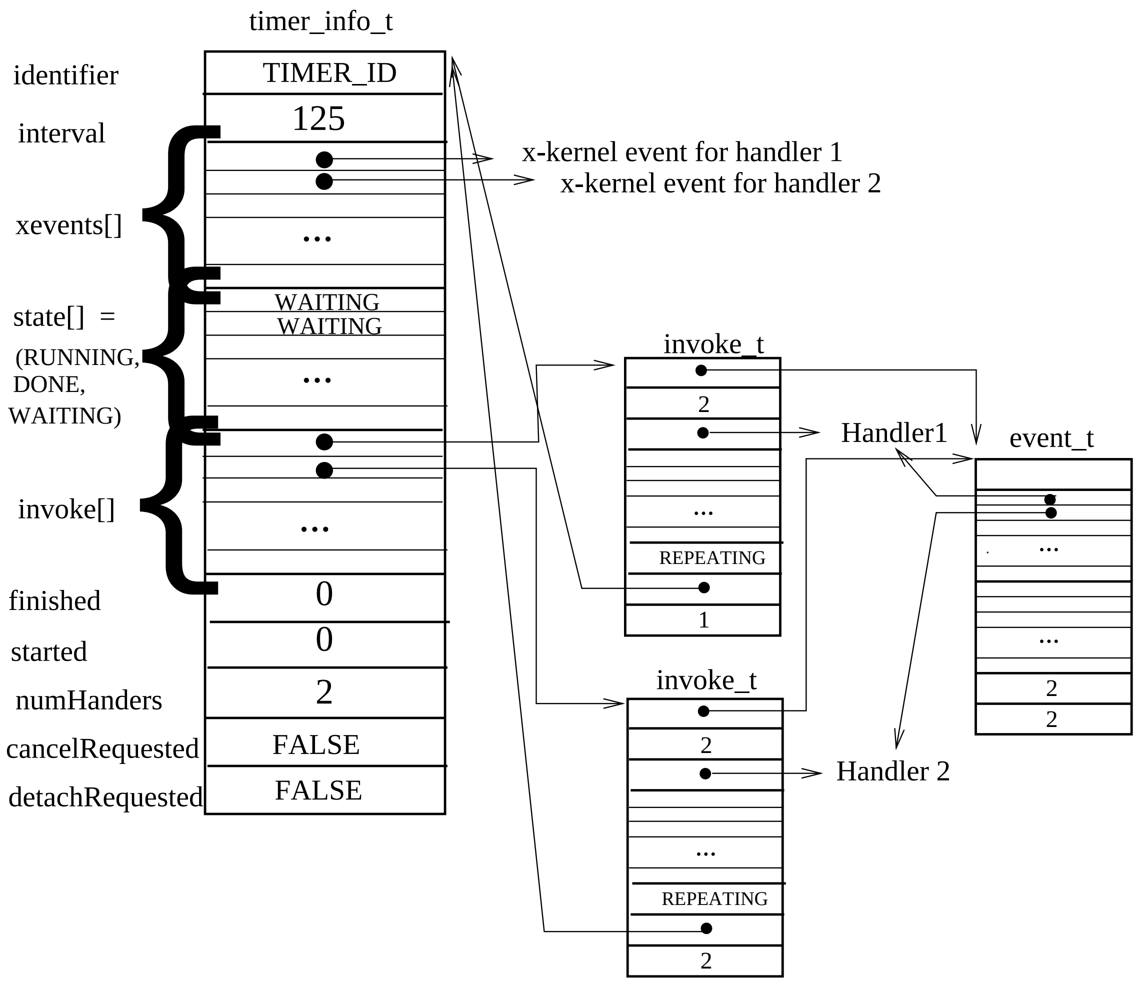

Figure 4.3: Timer event information structure for repeating event with two event handlers.

be atomic with respect to handler execution, so the structure must contain information about the status of handler execution. A unique handle is returned for each set timer event call, since the same timer event may have several concurrently executing instances. This handle is known is a `timer_info_t` structure and is given in Figure 4.3. `timer_info_t` contains pointers to the same type of `invoke_t` structures used for regular events, including a pointer to the event description structure `event_t`.

When the timer handler handle is passed to the cancel routine, the identifier is validated and then the timer can be cancelled. If execution has not started for this period execution (as recorded in the `started` count field), then all the *x*-kernel events (saved in the `xevents` array) can be cancelled. Otherwise, the `cancel_requested` boolean is marked as true and a repeating event will not reschedule itself for the next period. The status (running, done, waiting) of each handler is recorded in the `state` field.

4.2.4 Call Depth

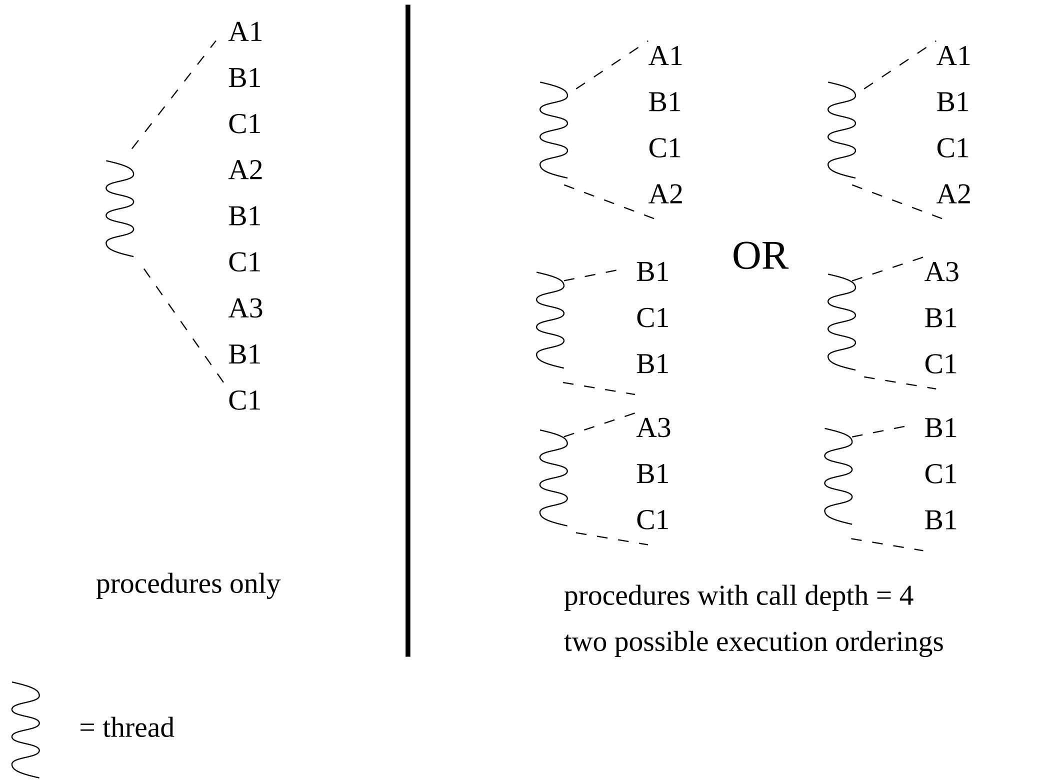

Figure 4.4: Possible event handler executions with and without call depth bounding.

Nesting of events executing as procedure calls can cause stack overflow and unfair scheduling. Recall that synchronous event execution is implemented as procedure calls and asynchronous execution can be optimized in this way. It is typical for events to raise other events, creating nested events. If this nesting goes very deep, stack overflow can occur. In addition, we found during experimentation that asynchronous event execution was sometimes necessary to make progress in certain protocols. If all asynchronous execution is implemented as procedure calls, then a call chain will continue to execute event handlers while other events not raised by the call chain go unserviced, which results in starvation of other events. In other words, procedure calls favor the current call chain leaving asynchronous events unserviced. Section 7.2.2 provides additional discussion of the need for asynchronous thread execution.

To eliminate this problem, *call depth bounding* can be enabled. This creates a hybrid execution structure that will continue to execute handlers as procedure calls until a specific nesting depth has been reached. At that point, the next asynchronous trigger will be executed by a new thread and the current thread will be allowed to terminate, thereby completing the call chain. The current call depth is recorded by associating a call depth count with each C-thread that executes events. When another event handler is executed as a procedure call, the call depth is incremented.

As an example, consider an event A with three handlers A1, A2, A3. Each of these handlers raise event B, which has one handler B1. B1 in turn raises event C with one handler C1. All events are raised as "ASYNC" events and the framework is configured to execute events with procedure calls. The execution order of the procedure-based events will be A1 B1 C1 A2 B1 C1 A3 B1 C1, as shown in the left panel of Figure 4.4. If the framework is configured with a call depth of 4, then A1 B1 C1 A2 would execute as procedure calls by a single thread. Having reached the maximum call depth, this thread would terminate, and B1 and A3 and would be scheduled asynchronously to execute in random order. If B1 were to execute first then the rest of the execution order would be B1 C1 A3 B1 C1. Both instances of C1 and the second B1 would be executed as procedure calls. The situation is analogous if A3 were to execute first. Figure 4.4 shows the execution structure of procedure call based events with and without call depth bounding.

4.3 Measurements of Event Implementation Performance

Event invocation and handler execution are the heart of the composite protocol, and therefore, the efficiency of events are central to the performance of the system. As discussed above, there are two implementation of events that can be used: light-weight user-level threads or procedure calls. We considered both styles and compared the performance and runtime behavior of each implementation.

The relative cost of using procedure calls versus a thread-based implementation was assessed using a null composite protocol designed to measure event execution times. Each test measured the round trip message transmission time based on 1000 round trips for two processes. The first is a normal *x*-kernel implementation of UDP without composite protocols; this provides a baseline. In the second, a composite protocol using the procedure call event implementation (CP-P) is inserted between the UDP protocol and user program on both the client and server sides. On the client side, CP-P simply passes messages and acknowledgments to the UDP protocol and user program, respectively, with no changes. On the server side, CP-P generates an acknowledgment for each message, as well as passing it through to the user program. 19 events are generated for each message round trip, and 19 handlers are invoked. The third test is identical, except that a runtime framework with the thread-based event mechanism is used. This composite protocol is called CP-T. Figure 4.5 illustrates the structure and message flow of the second and third configurations.

The results are shown in Table 4.1. Although these numbers clearly indicate some overhead, the results are encouraging. Based on the one byte test, each event handler activation costs no more than 33.7 microseconds for procedure-based event dispatching and 206 microseconds for thread based. Note that this figure includes amortizing all execution costs associated with a composite protocol over the handler activations, not just the cost of the invocation itself. The variance was observed to be low.

4.4 Creating a Composite Protocol

Source files are used to structure the components of a composite protocol. There are three categories of files: user supplied, user modifiable, and read only.

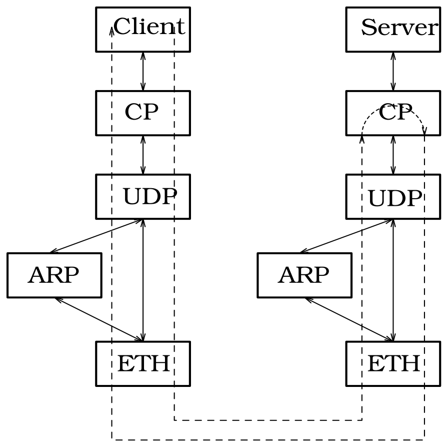

Figure 4.5: Experimental configuration

Packet Size	x-kernel UDP	+CP-P	+CP-T
1 byte	1.57	2.2	5.48
1 K	4.18	4.84	8.19
2 K	7.39	7.89	11.38
4 K	12.65	12.93	16.96
8 K	23.77	23.78	27.63

Table 4.1: Roundtrip time for null CP (in msec)

- *User supplied* files contain micro-protocol code and required routines such as attribute-to-header, header-to-attribute, attribute printing, and the initialization micro-protocol. The majority of the user's efforts are in creating this code.

- *User modifiable* files exist but can be modified to further customize the service. For example, push and pop procedures can be customized for multicast or other sending styles. The user can also modify defines to configure the composite protocol for call style interface or push style, enable procedure based execution of asynchronous event execution, bound call depth, and enable deallocation bit support. Some of the possible modifications require x-kernel specific knowledge, such as changing active and passive keys used to lookup sessions. However, the user only needs to make modifications if different behavior is needed, so for most protocol suites the minimal setup should be sufficient.

- *Read only* files contain only framework-specific code and are not alterable. These files include standard functions, such as bag of messages routines, event support, and the x-kernel encapsulation protocol. The user links these files with the rest of the composite protocol.

One user modifiable file concerns the lower-level protocol used. By default, a composite protocol uses UDP, which is sufficient for any composite protocol that only requires unreliable datagram service. However, the user has the option of changing the lower-level protocol to any x-kernel protocol, perhaps even another composite protocol. To do this, a support file must be created that contains procedures to create participant addresses and manage communication channels built on the new lower-level protocol sessions.

While many protocol suites can be build using UDP, the selection of the lower-level protocol naturally affects the selection of micro-protocols in the composite protocol. For example, if the lower-level protocol is an unreliable multicast protocol, then the send routine in the composite protocol can be much simpler since the lower-level protocol can issue a message to each group member automatically.

4.5 Possible Optimizations

Two additional optimizations that we have considered for reducing event overhead are in-lining of event handlers and evaluation of event guards. The simplest way to reduce event invocation overhead is to remove invocation entirely and in-line all event handlers. With complier support, the framework `raiseEvent()` procedure could be replaced with the micro-protocol code. The compiler would enforce all visibility rules and rename variables in the handler code that clash with variables in the surrounding micro-protocol code.

Implementation of event guards can greatly reduce the number of event invocations that terminate quickly after checking that the event guard is unsatisfied. Currently, event guards used in the PDL descriptions of micro-protocols are unimplemented, so the micro-protocol evaluates the guard explicitly after the handler has been invoked. While this is semantically equivalent to evaluation of guards before event handler execution, it results in greater overhead. If the guard could be evaluated by the `raiseEvent()` procedure in the framework instead, handlers with unsatisfied guards — i.e., guards that evaluate to false — could be discarded. For example, almost all micro-protocols register for the `Message_Inserted_Into_Bag` event, but most handlers are concerned with only a specific message type. In the current implementation, all the handlers are invoked and each handler executes an a conditional statement that succeeds in only the few handlers that go on to execute the rest of the handler. The other handlers simply exit, having added cost to the event execution time. Event guard evaluation could reduce the number of handler invocations substantially.

4.6 Summary

We have presented the implementation of the composite protocol approach based on the x-kernel. The implementation supports event execution using both threads and procedure calls, a shared bag of messages, coordinated sending of messages and deallocation, and message attributes. Primary event data structures and the organization of timer as well as regular events was discussed.

CHAPTER 5
GROUP RPC PERFORMANCE

In this chapter, we present performance measurements of multiple communication services configured from a collection of micro-protocols implementing different variants of regular and group RPC (GRPC). Micro-protocols are configured together into a composite protocol called `Group_RPC`. As described in Chapter 3, once constructed, `Group_RPC` gets included in an x-kernel protocol graph with UDP as its lower-level supporting protocol (see Figure 3.1). Measuring the performance of `Group_RPC` therefore yields the relative cost of the different configurations and their underlying semantic properties.

Our version of GRPC is based on point-to-point messages, so clients send individual requests to each server group member. Figure 5.1 shows the process level architecture and message flow between clients and servers. Request messages, Req(x), are sent from clients, while servers send reply messages, Rep(x), back to the client. Once reply messages are received, the client creates a return value according to its collation semantics. If total order is included, then one server acts as a coordinator that determines the ordering of requests. Thus, for each request, the coordinator sends an ordering message, Order(Req(x)), to all other servers.

5.1 Group RPC Micro-protocols

The micro-protocol suite is based on the semantic variations of GRPC described in [HS95a]; the categories that follow represent semantic variations of termination, ordering, communication, collation, call style, membership, and failure.

5.1.1 Termination Semantics

Termination semantics specify the guarantees that are given about the termination of a call. Included in the client composite protocol (CP).

- `BOUNDED (BND)`. Provides for bounded termination of client requests, i.e., either the request is executed within some interval or an exception is returned. When a request is sent, a timer event is set to generate a timeout.

- `UNBOUNDED(UBND)`. No *a priori* bound is set on a client request, so the client may wait indefinitely for a response.

5.1.2 Ordering semantics

Ordering semantics determine what guarantees are given about the execution order of requests by servers. If none of the micro-protocols are included, any ordering is possible.

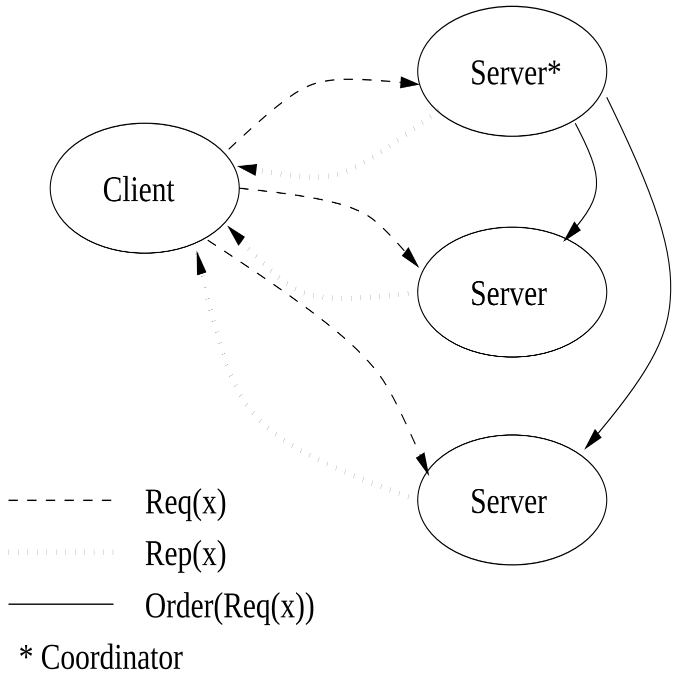

Figure 5.1: Process and message architecture.

- `FIFO`. Forces FIFO ordering of client requests at a server; if not included, the server may receive requests from a given client in any order. Servers order client requests using sequence numbers that are assigned by clients. Requests from multi-threaded clients are serialized before transmission. Included in both clients and servers.

- `TOTAL`. Forces total ordering of all client requests at all servers; if not included, the servers may not execute clients request in the same global order. The ordering of client requests is determined by a designated server process that acts as a coordinator. All non-coordinator servers receive requests but do not execute them until an ordering message is received from the coordinator. The coordinator processes requests only after at least one other server process has acknowledged receipt of the ordering message, which ensures a correct ordering even if the coordinator fails. Included only in servers.

- `FIFO` and `TOTAL`. Total order that preserves FIFO ordering. Requires inclusion of both FIFO and TOTAL micro-protocols.

5.1.3 Communication Semantics

Communication semantics specify guarantees about the communication between the client and server. Reliable transmissions are guaranteed if acknowledgment and retransmission micro-protocols are both included.

- `ACK`. Acknowledges request and response messages, and handles timeouts. If an acknowledgment message is not received in time, an event is raised notifying other protocols for possible retransmission. Included in clients and servers.

- `RETRANSMIT (RET)`. Sends retransmission requests for missing messages and responds to retransmission requests. Included in clients and servers.

- `CONTROL_RETRANSMISSION(CRET)`. Only used with totally ordered communication. Sends acknowledgments and waits for acknowledgment of control messages between servers. Included in servers only.

5.1.4 Collation Semantics

Collation semantics specify how responses from the server group are combined and the result returned to the client. All protocols included only in clients.

- `ONE_ACCEPT (1ACC)`. Implements a policy of accepting the first reply from any server as satisfying the client's request. Other responses are ignored.

- `ALL_ACCEPT (AAC)`. Implements a policy of collecting replies from all functioning servers before the RPC call is completed. If a server is no longer functioning, the new membership is used to prevent waiting forever for a response from a failed server.

5.1.5 Call Semantics

Call semantics specify whether the call thread in the client is blocked for synchronous call style or if it returns immediately for an asynchronous style. All protocols are included only in clients.

- `SYNC`. Provides synchronous request/reply call-style interface. The call thread is blocked until the call completes.

- `ASYNC`. Provides asynchronous push-style interface. The result of the call is returned by an upcall.

5.1.6 Membership Semantics

Membership semantics specify how information is collected about failed and functioning processes, and what can be guaranteed about the correctness of this information. Since point-to-point messages are used, clients must maintain information about server group membership to send requests and for collation of responses.

When total ordering of messages is used, server groups must also maintain their own membership to determine if all messages are received by all hosts and to ensure that requests are executed by all hosts. A single failure of any server member is tolerated, including the coordinator. When membership does change, the change event is ordered at each server so servers agree when the change occurred. The virtual synchrony property ensures that the membership change event occurs in the same place in the message stream.

- `CLIENT SERVER MEMBERSHIP (CSMEM)`. Manages the server membership for a client. Using the ACK protocol, a client times out unresponsive servers and removes them from the server list. This membership list is used by the `ALL_ACCEPT` micro-protocol to determine when all responses are received and for sending point-to-point messages to all servers. CSMEM is required for all configurations and is included in clients.

- `SERVER MEMBERSHIP (SMEM)`. Manages the server membership list for members of a server group. Membership is initialized at boot time from a static list and later, when dead servers are detected, they are removed from the membership list. Note that the server with the largest host address is the coordinator. No negotiation is required to determine the coordinator. Included in servers.

- `LIVE`. Servers send liveness messages to each other in a ring topology to detect when a server fails. If no liveness message is received within the interval, then the member is suspected to have failed and the "suspect host dead event" is raised. This triggers the membership micro-protocol to determine if the server is really dead. Included in servers only.

- `SIMPLE AGREEMENT(SIM)`. Simple agreement will send "server is dead" messages to other servers if a "suspect host dead" event is triggered. All other servers simply accept this declaration of a defunct server even if they have information to the contrary. Included in servers only.

- `VIRTUAL_SYNCHRONY(VS)`. Virtual synchrony insures that membership change messages appear in the same order relative to data messages for all hosts. When a failure occurs, non-failing servers exchange information about the highest ordering message that has been received before the failure occurred. This allows servers to synchronize on what messages should have been received before the membership change occurred. Included in servers only.

5.1.7 Failure Semantics

Failure semantics specify what guarantees are given to the client about the execution of requests by the server.

- `UNIQUE`. Eliminates duplicate request or reply messages using sequence numbers. Ensures that a request is never executed more than once even if the call returns unsuccessfully. Included in servers only

5.1.8 Driver Protocol

The suite requires a driver protocol, `GRPC`, for all combinations of micro-protocols. Verifies incoming messages and maintains client and server state information. Required for clients and servers.

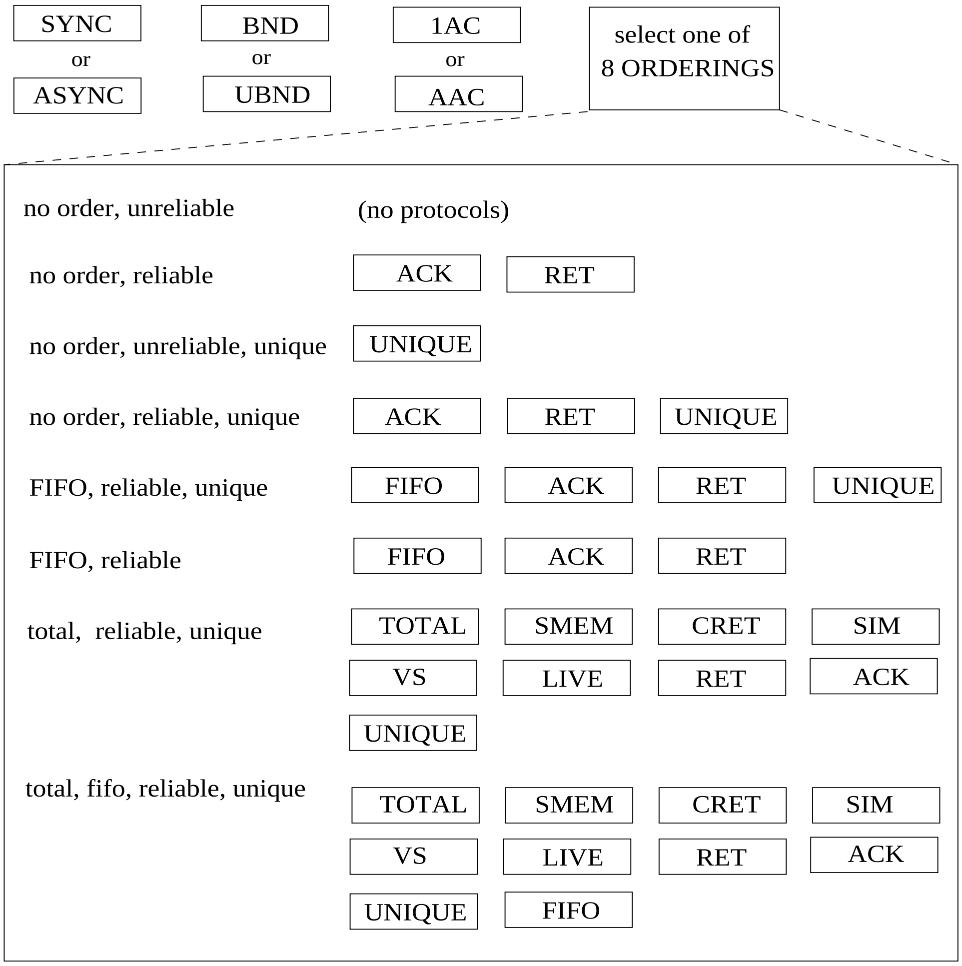

Figure 5.2: Group RPC configuration selections.

5.2 Combining Micro-Protocols

There are 64 possible GRPC configurations given the above collection of micro-protocols. The composite protocol may have synchronous or asynchronous call style, bounded or unbounded calls, one accept or all accept collation, and 8 selections of orderings. Figure 5.2 illustrates the possible selections of micro-protocols. All configurations require the `GRPC` and `CSMEM` micro-protocols.

The selection of call style, bounding of calls, and acceptance policies are independent choices and each only requires the inclusion of one micro-protocol implementing that property. Unique execution and `FIFO` ordering are each achieved through the inclusion of one micro-protocol. Reliable transmission of messages is accomplished through acknowledgment and retransmissions of messages, which requires the `ACK` and `RET` micro-protocols.

Total ordering is complex and requires several micro-protocols, because the servers must maintain there own membership to ensure totally ordered execution of client requests. As already noted, all servers receive request messages and one server acts as the coordinator, generating ordering messages that guide all servers to complete the requests in total order. All servers must receive all the request messages, so servers maintain their membership through the use of a liveness micro-protocol, `LIVE`. When a server is suspected of having failed, a simple agreement micro-protocol, `SIM`, is executed, which causes all servers to delete failed servers from membership lists. Virtual synchrony, `VS`, is used to ensure that the membership change occurs in the same point with respect to the request/reply message stream. Servers must communicate reliably or communication would halt if an ordering message was lost. This functionality is provided by `CRET`.

5.3 Measurements of Group RPC Configurations

Tests consisted of one or more clients sending a 4-byte integer to one or more servers, which respond with an integer. Each test makes 1000 RPC calls and was run 10 times. The round trip times are the average of the 10 test runs. To provide a baseline, a version of Sun RPC implemented using the standard *x*-kernel was also tested. Note, however, that Sun RPC is a peer-to-peer rather than group protocol, and, as a result, implements less functionality than `Group_RPC`.

All measurements were done on the experimental platform described in Chapter 4. In addition, tests requiring three or less hosts execute server and client processes on DecStation 5000/240s. For tests requiring more than three hosts, all server processes execute on DecStation 5000/240s and client processes execute on DecStation 5000/200s. Like the DecStation 2000/240s, DecStation 5000/200s are MIPS R3000 micro-processor based systems with separate off-chip 64 KB instruction and data caches, and 16 MB of memory. However, the DecStation 5000/200's processor clock rate is 25 MHz instead of 40 MHz.

The average roundtrip times for the various configurations are given in Table 5.1. All communication between hosts are point-to-point network messages. The relative ordering is what one would expect: normal Sun RPC using the *x*-kernel (BL) is fastest, and for the same micro-protocol configurations, increasing the number of servers and clients results in increased execution time. As noted, the *x*-kernel Sun RPC is included only for comparison. Such a protocol would naturally be used for simple client/server communication, but does not provide the multiple acceptance policies, group membership, multiple servers, or message ordering options needed for more complex applications.

In general, increasing the guarantees the communication service provides results in a slower roundtrip execution time. This is as expected, since the more guarantees that are given, the more expensive the algorithms required to implement the communication service. However, micro-protocols that increase message traffic degrade performance more than micro-protocols that only add computation time to clients or servers. For example, adding FIFO to configuration C8 (measured in configuration C9) results in a small increase in timing (0.02 msec) because it only adds sequence numbers to requests. On the other hand, the difference between configuration C7 and C8 is the addition of server membership and total ordering. The timing difference between these tests is appreciable (1.71 msec),

	System Configuration	Clients	Servers	avg	var
BL	x-kernel Sun RPC	one	one	4.38	0.00035
C1	GRPC,SYNC,1AC,CSMEM,UBND	one	one	6.30	0.018
		one	two	8.82	0.032
C2	GRPC,SYNC,AAC,CSMEM,UBND	one	two	8.85	0.052
C3	GRPC,ASYNC,FIFO,1AC,CSMEM,UBND	one	one	5.68	0.024
C4	GRPC,ASYNC,FIFO,1AC,CSMEM,BND	one	one	6.12	0.019
C5	GRPC,ASYNC,1AC,CSMEM,BND	one	one	5.58	0.012
C6	GRPC,ASYNC,AAC,CSMEM,BND, RET,ACK	one	one	8.49	0.026
		two	two	16.59	0.849
		two	three	22.71	0.008
C7	GRPC,ASYNC,FIFO,AAC,CSMEM,BND, UNIQUE,RET,ACK	one	one	8.91	0.043
		two	two	19.68	0.018
		two	three	23.76	0.003
C8	GRPC,ASYNC,AAC,CSMEM,BND, UNIQUE,RET,ACK,SMEM,LIVE,SIM CRET,TOTAL,VS	one	one	10.62	0.077
		two	two	35.22	0.230
		two	three	48.47	0.224
C9	GRPC,ASYNC,FIFO,AAC,CSMEM,BND, UNIQUE,RET,ACK,SMEM,LIVE,SIM CRET,TOTAL,VS	one	one	10.64	0.219
		two	two	44.19	2.40
		two	three	50.66	0.83

Table 5.1: Time for `Group_RPC` call (in msec)

because configuration C8 increases message traffic between servers. Similarly, the increase in the number of servers for configurations using total order results in big increases in running time, because the message traffic grows quadratically with the number of servers. For configurations C8 and C9, we can also see large increases (10 msec) with the addition of another server.

5.4 Detailed Analysis

Each configuration and its performance results are discussed below.

Configuration BL. x-kernel implementation of Sun RPC. Only supports a single client and server. Given as a baseline for comparison.

Configuration C1. Implements a group RPC service that provides a synchronous call interface and returns when the first response is received (i.e., a one accept policy). This variation could be used for applications that need to execute a request on any server before continuing but do not require that server responses be identical. The first performance figure is for one client and one server, which makes this configuration closest to BL.

However, this test runs slower than BL due to extra code that could handle multiple clients and servers. The second test uses two servers with the single client. As expected, two servers execute slower than one server. Even though only one server response is needed to complete the call, both servers generate responses, which means message traffic is at least double compared to the test of one server. As a result, network contention slows down the test.

Configuration C2. Identical to C1 but with an all accept policy, which causes the client to wait until responses from all servers are received. Hence, the call does not complete until the request has been executed on all servers. Such a configuration might be used, for example, in a simple replicated database, where the application must know that each group member has completed the request before continuing. The execution time for two servers is almost exactly the same time as C1 because both configurations generate the same amount of message traffic; although C1 only needs one reply, two replies are always generated. Network contention is the limiting factor in both configuration tests.

Configuration C3. Implements an asynchronous call style with FIFO ordering. The FIFO ordering micro-protocol ensures that the server executes all calls from a given client in FIFO order. Since the client executes with the asynchronous call style, multiple concurrent RPC requests can be issued, and may arrive at the server in any order. Multithreaded clients that need their actions serially executed in the server find this a useful configuration. The test of one client and server shows that the concurrent requests result in faster performance for clients. C3 executes faster than C1 even though both achieve serial execution of client requests. Specifically, C3 achieves this by the FIFO micro-protocol, while C1 achieves this by only requesting the next RPC call after the first call has completed.

Configuration C4. Adds bounded termination to C3, which makes this appropriate for applications that need to raise an exception when servers are not responding, so that the client will not appear "frozen" waiting for the RPC call to terminate. The addition of the bounded termination micro-protocol requires starting and resetting timers, which slightly increases the execution time over the unbounded configuration, C3.

Configuration C5. Same as C4 without FIFO ordering of client requests, which makes this suitable for applications that require detecting that servers are not responding but can execute client requests in any order. Runs slightly faster than the same one client, one server test of C4, since requests can be executed in any order. Servers can execute requests as soon as they are received and no FIFO micro-protocol code is executed.

Configuration C6. Reliable communication between clients and servers. All request and reply messages are acknowledged and if no acknowledgment is received, the message will be retransmitted. Such a configuration is suitable for reliable unordered communication between clients and servers, such as might be used by a reliable name service providing information about host utilization and resource availability. No requests or responses are lost but they may be executed in any order by all servers.

The first performance number is for execution with one client and one server. This runs slower than all previous configurations because it provides reliable communication, which adds more micro-protocol code and more message traffic for acknowledgments and retransmissions. The second test is executed with two clients and two servers, which essentially doubles the message traffic and execution time. The third test is executed with two clients and three servers, resulting in an expected proportional increase in message traffic.

Configuration C7. Adds unique execution to reliable communication. This is useful for applications that require no ordering of reliably delivery requests, but can only allow servers to execute the request only once. Such a facility is essential for non-idempotent operations, such as incrementing a value. The test of one client and one server of C7 compared with C6 reveals the cost of adding unique execution. To implement this, the server must record the identity of all client requests with the response message. If an identical request is received, the saved response message is retransmitted. The extra execution time results from checking every request to see if it is unique and saving messages. The test executed with two clients and two servers slightly more than doubles the message traffic and the execution time. The test executed with two clients and three servers result in a proportional increase in message traffic.

Configuration C8. Provides totally ordered execution of reliably transmitted request. Ordering is accomplished using a server that acts as the coordinator, sending ordering information about each request. Each non-coordinator server must receive the request message and an ordering message before it may execute the request. The servers maintain common membership by detecting server failures and then informing the rest of the group about the failure.

The execution time of C8 for one client and one server is 1.71 msec greater than the one server, one client test of C7, which provides an idea of the extra execution time resulting from the addition of five micro-protocols. The next test, executed with two clients and two servers, more than triples the execution time because message traffic increases quadratically with the number of servers and all servers must wait for total ordering messages before executing any requests. The final test was executed with two clients and three servers. Message traffic increases significantly, again resulting in much slower execution.

Configuration C9. Same as C8 but adds FIFO ordering of client requests, which results in total order that preserves FIFO ordering. These semantics are ideal for banking transactions that are executed on a cluster of servers for fault-tolerance. Each client transaction must be executed in the order that the client made the request, and all servers should execute all transactions in the same order for consistency.

The addition of FIFO increases the time only modestly from C8 for the single client, single server test, since no additional message traffic is introduced. When executed with two clients and two servers, the timing is quite a bit higher than the analogous configuration of C8, which was unexpected. We think this is due to the increased message traffic causing more messages to arrive out of order, and therefore, the servers having to wait to order the messages. The increase in variance is also large, which may indicate sensitivity to network traffic arrival rates. The final configuration is executed with two clients and three servers. Timing is close to C8, which indicates that performance is probably limited by the network.

5.5 Summary

In this chapter, a configurable group RPC service is described in which selected micro-protocols are combined to form a composite protocol that executes in the x-kernel. A variety of semantics are supported that cover the requirements of many different group communications applications. We also demonstrate that the services have reasonable performance, especially considering the preliminary nature of the prototype implementation. This chapter demonstrates the feasibility of our approach to designing and implementing modular communication protocols.

CHAPTER 6
PROTOCOLS FOR MOBILE COMPUTING

Mobile computing systems can benefit from configurable communication services in much the same way that fault-tolerant systems can benefit from the GRPC services described in the previous chapter. Here, we present the design of a micro-protocol suite intended for a range of mobile computing architectures and applications to illustrate the suitability of our approach for another type of distributed system. While the discussion of fault-tolerant systems centered around abstractions useful for supporting common structuring paradigms, our approach to mobile computing is based on supporting multiple hardware architectures, routing policies and qualities of service. Our specific focus is on building configurable services for base stations, mobile hosts and agents. Recall from Chapter 2 that base stations are gateways that connect the wired and wireless networks, where each base station administers connections to mobile hosts within its cell. An agent is a stationery process that acts as a proxy for a mobile host by maintaining connections to the applications.

6.1 Communication Requirements

The ability to vary the communication services for mobile computing is useful for several reasons, including to match the architecture, to allow different semantics, and for experimentation. Architecturally motivated variations are necessary because mobile hosts have a wide range of hardware capabilities. Some, such as the Apple Powerbook, have compute power that rivals desk top machines and are useful in a stand-alone capacity. These autonomous machines may be active participants in handoffs between base stations and are aware of their current connections. At the other end of the spectrum are machines such as the Xerox PARC TAB [AGSW93] that have very little storage capacity or processing power, which means that storage and processing are provided by resource rich machines in the stationary infrastructure. Such machines are often unaware of handoffs and therefore are passive participants. In addition, a communication service for mobile systems may have to accommodate an existing software architecture, including wireless protocols, protocols used to communicate between base stations, and routing software such as mobile IP.

A second reason communication requirements can differ is semantics, especially those related to quality of service. These guarantees may be given on a per connection basis or renegotiated when the host moves. The former guarantees a given quality of service even when the host moves to another cell, so that for example, a host with a high bandwidth connection retains that guarantee no matter what its location. If connection characteristics must be renegotiated, the negotiation may involve several rounds of messages to relevant applications. Renegotiated connections may only affect the mobile host that is moving or may also affect other preexisting connections. Yet another approach is to monitor bandwidth utilization of a mobile host, with unused allocation being given to other mobile hosts. Variations of such semantics can be customized to the application. For example, to facilitate multimedia display applications, a communication service can be selected that guarantees no changes to bandwidth allocations. However, mail reading applications can function with less strict guarantees, and a patient monitor application requires high reliability but less throughput. Some applications are tolerant of lossy transmissions during crossover into a new cell, making immediate handoffs less critical.

Finally, the ability to easily prototype experimental communication services is important in the field of mobile computing and is essential for rapid development of new distributed systems. Hardware, protocols for base station and host communication, handoff, and routing are all active areas of experimental system design. Sometimes this experimentation is to accommodate different hardware or protocols, but more often it is to test different ideas and philosophies of system design. For example, some designs promote the notion that mobile hosts should be unaware of their location; consequently, mobile hosts are not involved in handoffs in such systems. Another philosophy is to preserve the state of a mobile host in an agent process that mirrors the mobile host state, and to have this agent manage all interaction between the mobile host and applications. A system of protocols that make it easy to configure and implement different behaviors can only help facilitate this experimentation.

In this chapter, a micro-protocol suite that supports multiple variants of communication services for mobile computing is described. We focus our attention specifically on micro-protocols for base stations, mobile hosts, and agents. The base station and mobile host software is where control of mobility and cell boundary crossings resides; if these components are configurable, core mobility behavior can be changed to account for differences in architecture and semantics, or for experimental prototyping. Thus, core behavior can be divided further into two broad categories: handoff and quality of service. Handoffs are composed of three separate stages: detecting handoffs, handoff negotiation, and disconnection from the current base station. Variations of each stage are implemented by different micro-protocols that can be combined to create a complete handoff protocol. Quality of service micro-protocols are optional and can be included if desired.

6.2 Handoff Related Variations

Micro-protocols related to handoff are divided into orthogonal behaviors governing detection of when a handoff is needed or desirable, the actual handoff procedure, and disconnection. Micro-protocols can be combined to achieve different behaviors. For example, several detection mechanisms can be used with the same handoff protocol. To enhance configurability, we decouple behaviors; one set of micro-protocols make the handoff decision, while a different set governs how to execute the handoff and disconnect from the old base station. Each is addressed in turn below.

6.2.1 Handoff Detection

Handoff detection determines that a cell handoff is either desirable or necessary. There are two classes of approaches depending on whether the detection is done by the base station or mobile host. In addition, detection can be either performed with the assistance of a lower-level protocol that monitors the signal strength of base stations or solely by higher-level protocols. Finally, there are different strategies for preventing oscillation between the same two cells or doing handoffs between two equally reachable cells. Next, we outline several different detection approaches.

Using ICMP Messages

In this approach, base stations periodically transmit an ICMP (Internet Control Message Protocol) message requesting that mobile hosts in the area identify themselves by transmitting a response message. Mobile hosts then respond with their unique identifications, which enables the base station to detect the arrival of a new host. When such a host is detected, it becomes a candidate for a handoff from the old base station. Also, the lack of response allows detection of inactive hosts that have presumably left the cell, which is important for maintaining a correct list of active hosts for use in handoff requests. That is, if one cell has an inactive host and receives a handoff request there is little question that the host has moved from the area and can be released. Detecting an inactive host can also trigger an event that causes a message to be sent to applications or an agent indicating that no active connection is being maintained.

Using Host Beacons

Mobile hosts facilitate detection in this strategy by periodically transmitting a beacon message that informs base stations that the host is in the area. Except that the message is automatically generated by the mobile host rather than being sent in response to a query, this protocol is very similar to the ICMP protocol. Mobile hosts that are actively communicating and have sent a message within the beacon period do not need to send a separate message.

Monitor Based Detection

A lower-level protocol implemented in either hardware or software detects another base station that has better transmission and reception quality. When this happens, the detection protocol signals higher-level protocols to initiate a possible handoff. This is the simplest approach, although it requires lower-level support.

Lazy Detection

As an optimization, handoffs of inactive hosts can be implemented using a lazy strategy. In this case, a handoff is not performed if there has not been activity within some specified period, even if the host has moved. This can save communication overhead since connectivity is not maintained to hosts that are not transmitting. However, applications may not be able to initiate contact since current host location is not continually maintained.

Therefore, this micro-protocol would not be a good choice for those kinds of applications that require mobile hosts to be contacted at any time. An example of a good application for this kind of micro-protocol is a portable Web browser where activity is generally instigated by the mobile host. In this case, if there is activity then the host would perform handoffs; otherwise no handoffs are needed.

6.2.2 Handoff

The handoff micro-protocol governs the addition of a mobile host to different cells, and the update of location information within the rest of the system. In a handoff, a new base station attempts to "acquire" a mobile host by requesting a "release" from the old base station. Differences between handoff schemes include whether the mobile host is aware of the handoff, if the old base station participates in the negotiation, and if handoffs can be refused. It is also important that a single base station manage the sending and receiving of data for each mobile host at all times during the handoff process. Handoff schemes involve code in the base station, as well as in the mobile hosts and possibly agents.

The behavior for handoffs often starts with the new base station sending a request to the old base station to release the mobile host. The rest of the process determines under what conditions the old base station removes the mobile host from its active members list and releases the host. Some schemes always let the new base station have the new host, while others will not release the mobile host if the candidate host has communicated successfully within some specified time interval. In some schemes, the old base station is not known, so the agent for the mobile host must be contacted to coordinate the handoff, or the coordination is done by broadcasting the new connection to all base stations. In still other protocols, the new base station just assumes that the acquisition of the mobile host is successful unless some other base station responds otherwise. This is a negative acknowledgment style. Below, several specific schemes are described.

Negative Acknowledgment (NACK)

In the negative acknowledgment (NACK) approach, a host is acquired by a new base station only after a broadcast is made to all base stations and none responds with a "request denied" messages within the allowed time interval. After the handoff is completed, the base station broadcasts a notification message so all base stations are aware of the host's new base station. This type of broadcast-based protocol is typical of architectures that have fast control message transmission capability, such as the ATM Crosspoint system described in Chapter 2. Base stations that receive a handoff request will generally comply if they have had no activity from the host within a specified interval. Otherwise, it will contest the handoff by transmitting a "handoff request denied" message. Note that, unlike some handoff schemes, this protocol does not require any knowledge about the specific identity of the old base station.

It is possible for two base stations to attempt to add the same mobile host to their cells concurrently. To avoid this race condition, a random number is used to break ties. When the "handoff request" message is broadcast to all base stations, a newly generated random number is included in the message. If a base station receives a request that contains a higher number than it generated, it gives up its attempt to acquire the host and allows the other base station to win. Typically, startup communication is done this same way.

Mobile Host Initiated

The previous detection styles are all base station initiated. Handoffs can also be initiated by a mobile host when it crosses cell boundaries. This style is appropriate for mobile hosts that use a lower-level protocol to do detection, as discussed in the previous section. In this case, a mobile host can address a join message directly to the new base station, usually including the identifier of its current base station in the message. This results in negotiation between the old and new base stations about the handoff. For example, the old base station can choose to deny the request because it believes it has an on-going active connection with the mobile host. Another variation is to always grant control to the new base station and only notify the old one, in which case no negotiation is needed. This strategy is only appropriate for mobile hosts that have enough capability to store the state of current connections.

Agent Coordinated

In this approach, a mobile host's agent process functions as the final arbiter of which base station acquires the mobile host. A base station sends a message to the agent to request a handoff. If the agent has an active connection with the mobile host, it denies the handoff request; otherwise it allows the handoff to proceed. Base stations locate agent processes using a name service that maps mobile host identifiers to agent process addresses. Note that agent-coordinated handoff does not require that base stations communicate with each other.

6.2.3 Oscillation Prevention

In mobile systems, overlap of base station cells is needed to provide complete coverage for a geographic area. As a result, two base stations can simultaneously decide to add a host to their respective cell. This may even continue indefinitely if the host remains in this crossover area, shown in Figure 6.1. *Oscillation prevention* prevents a host from undergoing such constant handoffs. Most architectures do not address this issue, probably because current test situations do not involve large numbers of mobile hosts.

An oscillation prevention protocol can be included with any of the handoff protocols to prevent the algorithm from attempting to acquire the same mobile host repeatedly. This protocol counts the number of unsuccessful handoff attempts and if it passes a threshold, the handoff is not run again until a time interval has passed. While not preventing the problem entirely, this strategy will reduce extra handoffs and prevent a base station from constantly making unsuccessful handoff attempts. Oscillation protocol is installed in the mobile host if handoffs are initiated there or in the base station otherwise.

6.2.4 Disconnection

Disconnection covers how the old base station disposes of packets and state from a mobile host that has been handed off. Packets from a mobile host that are addressed to an

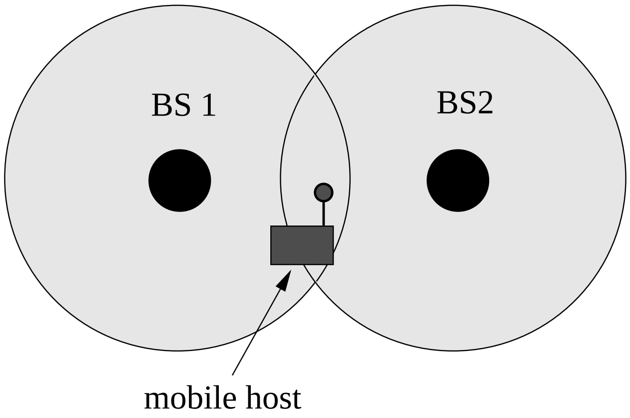

Figure 6.1: Mobile host in range of two base stations.

application are simple to deal with, since they can be sent to an application or agent using the stationary wired network. The difficult question is how to dispose of the undelivered packets at the old base station that are bound for the mobile hosts.

There are three approaches to handling leftover packets from a connection: drop them, forward them to the new base station for delivery, or quickly deliver as many as possible (i.e., to "drain" packets). Certainly, the simplest is to drop them, in which case, an end to end protocol must handle the retransmission of these packets if reliability is needed. While seemingly wasteful, it can be argued that the wired links are much faster than wireless medium so the retransmission delay of these packets is negligible. The second approach, forwarding the packets to the new base station, saves retransmission of the packets by the application. In this case, the new base station receives forwarded packets and delivers them in order, making the forwarding transparent to the mobile host. The final approach, to drain packets, is based on the argument that when a handoff is occurring, the mobile host is still reachable and the old base station should delivery the enqueued messages as soon as possible. Of course, this assumes that the old base station can still maintain contact with the mobile host, perhaps at degraded transmission quality. If a drain is unsuccessful and packets are undeliverable, then the behavior can revert to dumping or forwarding packets.

6.3 Example Mobility Micro-Protocols

This section contains micro-protocols for detecting handoff conditions, performing handoffs, and disconnecting. The particulars of each micro-protocol are explained, as is their relationship and compatibility with other micro-protocols. In Section 6.5 we show how to augment these protocols by adding micro-protocols implementing quality of service guarantees. Then in Section 6.7 we describe some sample communication services using the micro-protocols that have been presented.

Figure 6.2 summarizes the micro-protocols for detection, handoff, and disconnection, as well as illustrates the micro-protocol combinations that can be selected for a commu-

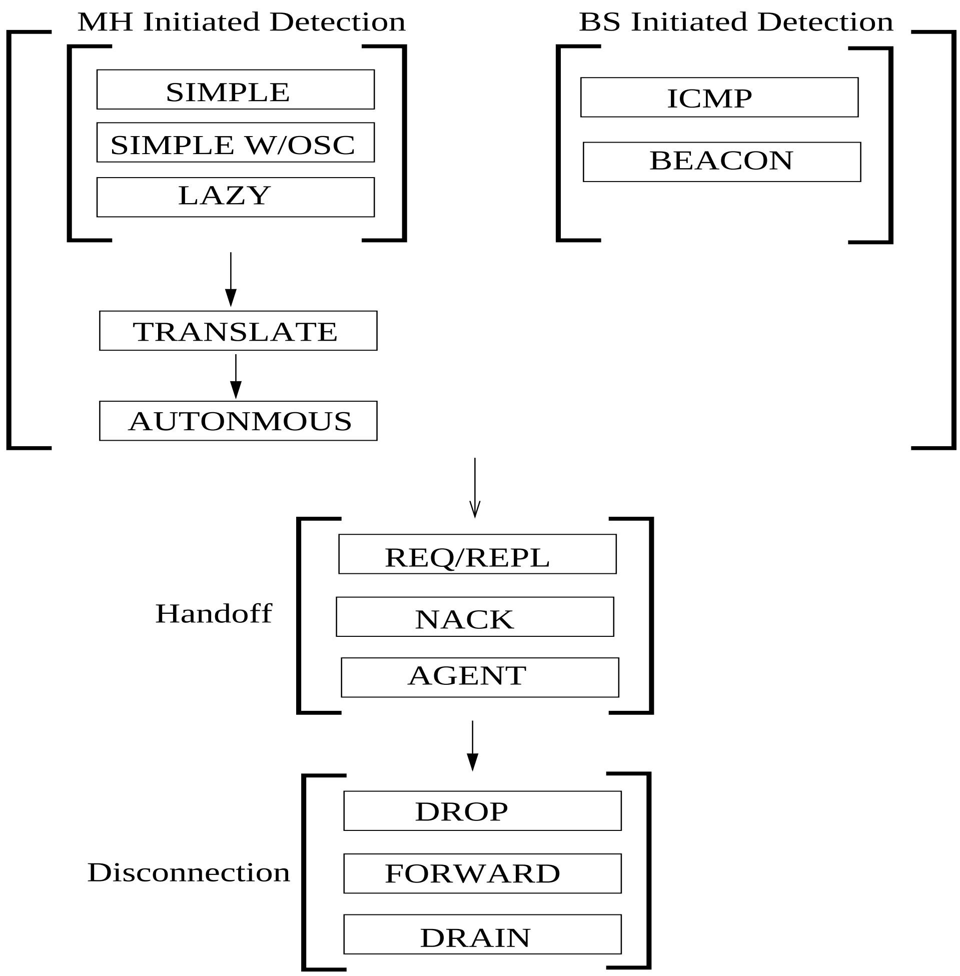

Figure 6.2: Overall micro-protocol structure.

nication service. The brackets indicate a choice and can be nested, as in the selection of detection micro-protocols where either a mobile host or base station initiated detection scheme can be used. An arrow indicates that events or messages are sent between two micro-protocols, so both micro-protocols must be included if either is selected. For example, if any micro-protocol from the set {SIMPLE, SIMPLE W/OSC, LAZY} is selected, then TRANSLATE and AUTONOMOUS must also be included since there are message and event arrows between these micro-protocols. However, if either ICMP or BEACON are selected, then TRANSLATE and AUTONOMOUS are not needed since there is no event or message communication.

6.3.1 Detection Micro-Protocols

This section presents five micro-protocols for detection of handoff conditions. As noted above, there are two classes of detection micro-protocols, those initiated by base stations and those initiated by mobile hosts. The former will be presented first (ICMP and BEACON), followed by the latter (SIMPLE, SIMPLE W/OSC, LAZY). Each micro-protocol is given a descriptive name followed by either "MOBILE HOST" or "BASE STATION" depending on whether the micro-protocol is to be executed in a composite protocol for mobile hosts or base stations.

```
micro-protocol ICMP MOBILE HOST{
  actions{
    Message_Inserted_Into_Bag(CP_Msg_t Msg)
        & Msg->type == ICMP_ID_REQUEST →
      sendMsg(ICMP_RESPONSE, myId);
  }
} end micro-protocol ICMP MOBILE HOST
```

Figure 6.3: ICMP based detection for mobile hosts

The ICMP MOBILE HOST micro-protocol (Figure 6.3) responds to ICMP requests that are broadcast by base stations to detect which hosts are in their cell. When such a `ICMP_ID_REQUEST` message arrives, a corresponding `ICMP_RESPONSE` message is sent with the host identification.

The ICMP BASE STATION micro-protocol (Figure 6.4) sends out the required ICMP requests once every detection interval. A repeating timer event is started when the micro-protocol is initialized and will continue to repeat until the micro-protocol has terminated. This timer causes a private event to be raised every interval that invokes a handler to send the request messages. Mobile hosts in the area are stored in a table of active hosts and marked with the timestamp for each interval that it responds to a request. A new host is detected when a response is received from a host not in the table; this triggers the `Join_New_MH` event. Hosts that have not responded within a certain interval are removed from the table, and the `MH_Inactive` event is triggered. When a host has been successfully acquired, the `Host_Joined(MH)` event is generated by a handoff micro-protocol. In this case, the host is added to the active table. This micro-protocol also provides access to the last activity timestamp of hosts, which can be used by quality of service micro-protocols.

```
micro-protocol ICMP BASE STATION{
  exports{
    proc    SetICMPInterval(int val);
    proc    int GetICMPInterval();
    proc    timestamp_t MemberLastTimestamp(member_t MH);
    event   Join_New_MH(member_t MH);
    event   MH_Inactive(member_t MH);
  }
  imports{
    proc    boolean LookupMemList(member_t MH);
  }
  private{
    event       ICMP_Timer();
    int         ICMPIntervalDEFAULT_ICMP_INTERVAL_BS;
    table_t     timestampMembers;
    timestamp_t curTimestamp;
  }
  initialize{
    setTimerEvent(ICMPTimer, REPEATING, ICMPInterval);
    curTimestamp = INIT;
  }
  actions{
    Message_Inserted_Into_Bag(CP_Msg_t Msg) & type == ICMP_RESPONSE  →
      if (LookupMemList(Msg->MH) == FALSE) {
        raiseEvent(Join_New_MH, ASYNC, Msg->MH);
      }
      else {
        entry = lookup(timestampMembers, MH) ;
        entry->timestamp = curTimestamp;
      }
    ICMP_Timer()  →
      for each entry(timestampMembers)) {
        if ((curTimestamp - entry->timestamp) > INACTIVE_THRESH ){
          raiseEvent(MH_Inactive, SYNC, entry->MH);
          deleteEntry(entry, timestampMembers);
        }
      }
      increment(curTimestamp);
    Host_Joined(MH)  →
      entry = tableInsert(timestampMembers, MH);
      entry->timestamp = curTimestamp;
  }
  ... code for SetICMPInterval, GetICMPInterval, MemberLastTimestamp...
} end micro-protocol ICMP BASE STATION
```

Figure 6.4: ICMP based detection for base stations

```
micro-protocol BEACON MOBILE HOST{
  exports{
    proc    SetBeaconInterval(int val);
    proc    int GetBeaconInterval();
  }
  imports{
  }
  private{
    event   Beacon_Timer();
    boolean msgSent=FALSE;
    Msg_t   beacon;
    int     beaconInterval=DEFAULT_BEACON_INTERVAL_MH;
  }
  initialize{
    setTimerEvent(beaconTimer, REPEATING, beaconInterval);
  }
  actions{
    Message_Pushed_From_CP(CP_Msg_t Msg)   →
      msgSent = TRUE;
    Beacon_Timer()  →
      if (!msgSent) {
        sendMsg(beacon);
      }
  }
  ... code for SetBeaconInterval, GetBeaconInterval...
} end micro-protocol BEACON MOBILE HOST
```

Figure 6.5: Beacon based detection for mobile hosts

The BEACON micro-protocols are similar to those using ICMP except that the messages are periodically generated automatically by a mobile host rather than in response to a query. BEACON MOBILE HOST (Figure 6.5) sets a timer event to trigger the sending of a beacon packet with the appropriate host identifier. Data messages are sufficient for informing the base station of the mobile host's presence, which means that the beacon is piggybacked on every outgoing message.

BEACON BASE STATION (Figure 6.6) has an interval timer and table of active hosts that is used to detect mobile hosts that have recently entered the cell. The base station adds and removes hosts from this table in the same manner as the ICMP BASE STATION micro-protocol. It also triggers the same `Join_New_MH` and `MH_Inactive` events.

```
micro-protocol BEACON BASE STATION{
  exports{
    proc    SetBeaconInterval(int val);
    proc    int GetBeaconInterval();
    proc    timestamp_t MemberLastTimestamp(member_t MH);
    event   Join_New_MH(member_t MH);
    event   MH_Inactive(member_t MH);
  }
  imports{
    proc    boolean LookupMemList(member_t MH);
  }
  private{
    event       Beacon_Timer();
    int         beaconIntervalDEFAULT_BEACON_INTERVAL_BS;
    table_t     timestampMembers;
    timestamp_t curTimestamp;
  }
  initialize{
    setTimerEvent(beaconTimer, REPEATING, beaconInterval);
    curTimestamp = INIT;
  }
  actions{
    Message_Inserted_Into_Bag(CP_Msg_t Msg)   →
      if (LookupMemList(Msg->MH) == FALSE) {
        raiseEvent(Join_New_MH, ASYNC, Msg->MH);
      }
      else {
        entry = lookup(timestampMembers, MH) ;
        entry->timestamp = curTimestamp;
      {
    Beacon_Timer()  →
      for each entry(timestampMembers)) {
        if ((curTimestamp - entry->timestamp) > INACTIVE_THRESH ){
          raiseEvent(MH_Inactive, SYNC, entry->MH);
          deleteEntry(entry, timestampMembers);
        }
      }
      increment(curTimestamp);
    Host_Joined(MH)  →
      entry = tableInsert(timestampMembers, MH);
      entry->timestamp = curTimestamp;
  }
  ... code for SetBeaconInterval, GetBeaconInterval, MemberLastTimestamp...
} end micro-protocol BEACON BASE STATION
```

Figure 6.6: Beacon based detection for Base stations

```
micro-protocol SIMPLE MOBILE HOST{
  exports{
    event   Join_New_BS(bs_t oldBS, newBs);
  }
  imports{
    bs_t curBS /* global variable */
  }
  private{
  }
  initialize{
  }
  actions{
    Message_Popped_To_CP(CP_Msg_t Msg) & Msg->type == BETTER_BS  →
      /* Message from lower-level protocol */
      raiseEvent(joinNewBS, ASYNC, MH, curBS, Msg->newBS);
  }
} end micro-protocol SIMPLE MOBILE HOST
```

Figure 6.7: Simple detection micro-protocol for mobile hosts

SIMPLE MOBILE HOST is the first of the mobile host initiated detection micro-protocols that rely on a lower-level protocol to determine if there is a base station in the area with better signal quality. For all these micro-protocols, only the mobile mobile host micro-protocol is described. The base station component is trivial; either it does nothing because a lower-level protocol transmits the identification packets as part of its wireless channel management or it just transmits identification packets periodically. When any of the mobile host initiated micro-protocols are used, the AUTONOMOUS MOBILE HOST micro-protocol must also be included.

The code for SIMPLE MOBILE HOST (Figure 6.7) relies on a lower-level protocol to send up a message indicating that a better or new base station has been detected. In response, the micro-protocol sends a wireless message to the new base station to request a handoff (join) to the cell.

SIMPLE W/OSC PREVENTION MOBILE HOST in Figure 6.8 is similar to SIMPLE MOBILE HOST but with the addition of oscillation prevention. The micro-protocol keeps a timestamp of the last cell change and the base stations involved to prevent an attempt to immediately switch back to the previous base station.

```
micro-protocol SIMPLE W/ OSC MOBILE HOST {
  exports{
    event   Join_New_BS(bs_t oldBS, newBs);
  }
  imports{
    event   Join_Complete(bs_t newBS);
    bs_t    curBS; /* global variable */
    boolean handoffEnabled; /* global variable set by QoS */
  }
  private{
    timestamp_t lastChangeTimestamp;
  }
  initialize{
    lastChangeTimestamp = INIT;
  }
  actions{
    Message_Popped_To_CP(CP_Msg_t Msg) & Msg->type == BETTER_BS  →
      /* Message from lower-level protocol */
      if (Msg->newBS == lastBS)
        if (((currentTimestamp() - lastChangeTimestamp) > TRESH) &&
          handoffenabled) {
          raiseEvent(joinNewBS, ASYNC, MH, curBS, Msg->newBS);
        }
    Join_Complete(bs_t newBS)  →
      setTimestamp(lastChangeTimestamp);
  }
} end micro-protocol SIMPLE W/ OSC MOBILE HOST
```

Figure 6.8: Simple detection with oscillation prevention for mobile hosts

LAZY MOBILE HOST (Figure 6.9) is similar to SIMPLE MOBILE HOST but does not request a handoff unless there has been activity in the last `ACTIVITY_THRESH` time period. Note that the activity threshold interval records both sending and receiving, so even if the host is just receiving a stream of data, handoffs will still be performed.

6.3.2 Handoff Protocols

Handoff micro-protocols govern how the handoff is negotiated after it has been detected by a detection micro-protocol. The first micro-protocol, AUTONOMOUS MOBILE HOST (Figure 6.10), is required for mobile hosts that are using any of the mobile host initiated detection micro-protocols. In this strategy, the event of handoff detection in the mobile host is translated into a message that is sent to the new base station. The response is then translated into a `Join_Complete` event, or the join may time out. The micro-protocol also controls the sending of data to base stations by enabling or disabling uplink capability with the `mode` variable. Since many architectures prohibit data transmissions when a handoff

```
micro-protocol LAZY MOBILE HOST{
  exports{
    event   Join_New_BS(bs_t oldBS, newBs);
  }
  imports{
    event   Join_Complete(bs_t newBS);
    bs_t curBS; /* global variable */
    boolean handoffEnabled; /* global variable set by QoS */
  }
  private{
    timestamp_t activityTimestamp;
  }
  initialize{
    activityTimestamp = INIT;
  }
  actions{
    Message_Popped_To_CP(CP_Msg_t Msg) & Msg->type == BETTER_BS  →
      /* Message from lower-level protocol */
      if ( (curTimestamp() - activityTimer) > ACTIVITY_THRESH) {
          raiseEvent(joinNewBS, ASYNC, MH, curBS, Msg->newBS);
        }
    Message_Popped_To_CP(CP_Msg_t Msg) & Msg->type != BETTER_BS  →
      setTimestamp(activityTimestamp);
    Message_Pushed_From_CP(CP_Msg_t Msg) & Msg->type != BETTER_BS  →
      setTimestamp(activityTimestamp);
  }
} end micro-protocol LAZY MOBILE HOST
```

Figure 6.9: Lazy detection for mobile hosts

is in progress, the `mode` variable is monitored by the send micro-protocol and used as a signal that sending should be disabled. This could also have been implemented using an event, with the send module changing the transmission mode when the event is triggered.

REQUEST/REPLY BASE STATION (Figure 6.11) is executed by the new base station during a handoff. The base station responds to the `Join_New_MH` event by checking if the host is already in the active list, which would indicate that it had recently been in the cell and no handoff is needed. Otherwise, a message is sent to the old base station to request a release of this host. If granted, the old base station informs applications interacting with the host that it has moved and the identity of the new base station. Then the old base station initiates a disconnect micro-protocol, which disposes of packets that were addressed to the host. When disconnection is complete, a `RELEASE_GRANTED` message is sent to the new base station. A table named `MHRequestingJoin` is used to match up `RELEASE_GRANTED` messages with mobile hosts that have started the handoff procedure.

An alternative strategy is implemented by NACK BASE STATION (Figure 6.12), which broadcasts a handoff request to all base stations. The micro-protocol responds to the `Join_New_MH` message by setting a timer and then broadcasting a `RELEASE_REQUESTED` message to all base stations. If a `RELEASE_DENIED` message is received before the timer event is triggered, the handoff is aborted. Otherwise, the handoff is completed and a HANDOFF message is sent to all base stations. If the `RELEASE_DENIED` message is also broadcast, then the HANDOFF message can be omitted since all base stations can infer the result of the request.

```
micro-protocol AUTONOMOUS MOBILE HOST{
  exports{
    event   Join_Complete(bs_t newBS);
  }
  imports{
    event   Join_New_BS(bs_t oldBS, newBs);
    event   QOS_Handoff( QoS_t *QoS);
    mode_t  mode; /* controlling sending */
  }
  private{
    timer_t requestJoinTimerHandle;
    bs_t    joiningBS; /* global variable */
  }
  initialize{
  }
  actions{
    Join_New_BS(bs_t newBS)  →
      /* do not transmit any messages while join in progress */
      mode = NO_UPLINK;
      raiseEvent(QOS_Handoff, SYNC, QoS);
      sendMsg(JOIN_REQUEST, newBS, curBS, *QoS);
      requestJoinTimerHandle =
        setTimerEvent(requestJoinTimer, ONCE, JOIN_INTERVAL, newBS);
      joiningBS = newBS;
    requestJoinTimer  →
      /* Abort joining attempt */
      mode = UPLINK;
    Message_Inserted_Into_Bag(CP_Msg_t Msg) & Msg->type = JOIN_REPLY_YES →
      /* Joining accepted */
      if ( Msg->BS == joiningBS) {
        cancelTimer(requestJoinTimerHandle);
        prevBS = curBS;
        curBS = Msg->BS;
        raiseEvent(Join_Complete, ASYNC, Msg->BS);
      }
    Message_Inserted_Into_Bag(CP_Msg_t Msg) &
        Msg->type = JOIN_REPLY_DENIED →
      /* Joining denied!  */
      cancelTimer(requestJoinTimerHandle);
      mode = UPLINK;
  }
} end micro-protocol AUTONOMOUS MOBILE HOST
```

Figure 6.10: Autonomous mobile host handoff for mobile hosts

```
micro-protocol REQUEST/REPLY BASE STATION {
  exports{
    event   Start_Disconnect(member_t MH, bs_t oldBS);
    event   Host_Joined(member_t MH);
    event   Host_Released(member_t MH);
    event   QOS_Handoff(member_t MH, QoS_t *QoS);
    event   QOS_Info_Request(Member_t MH, QoS_t *QoS);
  }
  imports{
    event   End_Disconnect(member_t MH, bs_t oldBS);
    event   Join_New_MH(member_t MH, bs_t oldBS);
  }
  private{
    table_t MHRequestingJoin;
  }
  initialize{
    initTable(MHRequestingJoin);
  }
  actions{
    Join_New_MH(member_t MH, bs_t oldBS)  →
      if (LookupMemList(MH) == TRUE) {
        /* already in list of active hosts */
        sendMsg(JOIN_REPLY_YES, MyID);
      }
      else {
        tableInsert(MHRequestingJoin, MH, oldBS);
        sendControlMsg(RELEASE_REQUESTED, MH, oldBS);
      }
    Message_Inserted_Into_Bag(CP_Msg_t Msg) & Msg->type = RELEASE_GRANTED →
      if ( (entry = tableLookup(MHRequestingJoin, Msg->MH))) {
        tableDelete(MHRequestingJoin, entry);
        raiseEvent(Host_Joined, SYNC, entry->MH, Msg->QoS);
      }
      else {
        sendControlMsg(ERROR, Msg->BS, Msg);
      }
    Message_Inserted_Into_Bag(CP_Msg_t Msg)
        & Msg->type = RELEASE_REQUESTED →
      for all server connected {
        sendControlMsg(HANDOFF, myId, Msg->BS);
      }
      raiseEvent(Start_Disconnect, ASYNC, Msg->MH, Msg->newBS);
    End_Disconnect(member_t MH, bs_t newBS)  →
      raiseEvent(QOS_Handoff, SYNC, MH, QoS);
      sendControlMsg( RELEASE_GRANTED, newBS, *QoS);
      raiseEvent(Host_Released, ASYNC, MH);
  }
} end micro-protocol REQUEST/REPLY BASE STATION
```

Figure 6.11: Request/reply handoff for base stations

```
micro-protocol NACK BASE STATION {
  exports{
    event   Start_Disconnect(member_t MH, bs_t oldBS);
    event   Host_Joined(member_t MH);
    event   Host_Released(member_t MH);
  }
  imports{
    event   Join_New_MH(member_t MH);
    proc    timestamp_t MemberLastTimestamp(member_t MH);
  }
  private{
    table_t MHRequestingJoin;
    timer_t joinTimerHandle;
    event   joinTimer(member_t MH);
  }
  initialize{
    initTable(MHRequestingJoin);
  }
  actions{
    Join_New_MH(member_t MH) →
      entry = tableInsert(MHRequestingJoin, MH);
      joinTimerHandle = setTimerEvent(JoinTimer, ONCE, JOIN_TIME, MH);
      addToEntry(entry, joinTimerHandle);
      /* message is broadcast to all base stations */
      sendControlMsg(RELEASE_REQUESTED, item->MH, item->oldBS);
    Join_Timer(member_t MH) →
      /* if no BS responds can add the host to my cell */
      if ( (entry = tableLookup(MHRequestingJoin, MH)) {
        raiseEvent(host_Joined, SYNC, MH);
        tableDelete(MHRequestingJoin, MH);
        /* all base stations and routers informed */
        sendControlMsg(HANDOFF, MH, myId);
      }
    Message_Inserted_Into_Bag(CP_Msg_t Msg) & Msg->type = RELEASE_DENIED →
      if ( (entry = tableLookup(MHRequestingJoin, Msg->MH))) {
        cancelTimer(entry->joinTimerHandle);
        tableDelete(MHRequestingJoin, entry);
        raiseEvent(Host_Denied, entry->MH);
      }
      else {
        sendControlMsg(ERROR, Msg->BS, Msg);
      }
    Message_Inserted_Into_Bag(CP_Msg_t Msg) &
        Msg->type = RELEASE_REQUESTED →
      if (LookupMemList(Msg->MH)) {
        if ((curTimestamp - MemberLastTimestamp(Msg->MH)) >
            INACTIVE_THRESH) {
          raiseEvent(Host_Released, ASYNC, Msg->MH);
          raiseEvent(Start_Disconnect, AYNC, Msg->MH, Msg->oldBS);
        }
        else {
          sendControlMsg(RELEASE_DENIED, Msg->MH, myId);
        }
      }
} end micro-protocol NACK BASE STATION
```

Figure 6.12: NACK handoff micro-protocol for base stations

```
micro-protocol AGENT COORDINATED BASE STATION {
  exports{
    event   Start_Disconnect(member_t MH, bs_t oldBS);
    event   Host_Joined(member_t MH);
    event   Host_Released(member_t MH);
    event   QOS_Handoff(Member_t MH, QoS_t *QoS);
    event   QOS_Info_Request(Member_t MH, QoS_t *QoS);
  }
  imports{
    event   Join_New_MH(member_t MH);
    proc    getHostAgent(member_t MH);
  }
  actions{
    Join_New_MH(member_t MH)  →
      agent = getHostAgent(MH);
      /* Inform MH agent, it forward request to old base station */
      sendControlMsg(RELEASE_REQUESTED, agent, MH, MyID);
    Message_Inserted_Into_Bag(CP_Msg_t Msg) &
        Msg->type = RELEASE_REQUESTED →
      /* Release is always OK send back QoS response */
      raiseEvent(QOS_Handoff, SYNC, Msg->MH, QoS);
      sendControlMsg(RELEASE_GRANTED, Msg->BS, MyId, *QoS);
      raiseEvent(Host_Released, ASYNC, MH);
      raiseEvent(Start_Disconnect, ASYNC, Msg->MH, Msg->newBS);
    Message_Inserted_Into_Bag(CP_Msg_t Msg) & Msg->type = RELEASE_GRANTED →
      raiseEvent(Host_Joined, ASYNC, MH, Msg->QoS);

} end micro-protocol AGENT COORDINATED BASE STATION
```

Figure 6.13: Agent Coordinated handoff for base stations

The final handoff style, implemented in the AGENT COORDINATED BASE STATION (Figure 6.13), allows agents to coordinate handoff requests. Since agents already coordinate the data going to the host, this simply involves informing the base stations, so it knows where to forward new packets. The old base station cannot refuse in this approach, so the agent just sends it notification to allow it to deal with extra packets and update its active table.

```
micro-protocol TRANSLATE BASE STATION {
  exports{
    event   Join_New_NM(bs_t oldBS, newBs);
  }
  imports{
    event   Host_Joined(member_t MH);
    event   Host_Denied(member_t MH);
  }
  actions{
    Message_Popped_To_CP(CP_Msg_t Msg) & Msg->type == JOIN_REQUEST  →
      /* Set timer and if not response in time then denied */
      raiseEvent(Join_New_MH, ASYNC, Msg->MH, Msg->curBS, Msg->newBS,
        Msg->QoS);
    Host_Joined(member_t MH) →
      sendMsg(JOIN_REPLY_YES, MH, MyId);
    Host_Denied(member_t MH) →
      sendMsg(JOIN_REPLY_YES, MH, MyId);
  }
} end micro-protocol TRANSLATE BASE STATION
```

Figure 6.14: Translate messages into events for base stations

All the base station handoff micro-protocols described above execute handlers when the `Join_New_NM` event is triggered. However, some of the detection micro-protocols — in particular SIMPLE, SIMPLE W/ OSC, and LAZY — all send a message when a handoff is triggered since the detection is done on the mobile host rather than the base station. Hence, to use the same handoff micro-protocols, this message must be translated into an event. The TRANSLATE micro-protocol (Figure 6.14) performs this function. It also translates the completion of the handoff micro-protocol from an event into a message that is sent to the mobile host.

6.3.3 Disconnection

Separate and orthogonal from handoff is the decision about what to do with packets left in the old base station during and after a handoff. This section contains three micro-protocols for disconnection: DROP, DRAIN, and FORWARD. All three are executed on the old base station and start to process packets of the mobile host to be handed off when the `Start_Disconnect` event is triggered. The `End_Disconnect` event is raised when the disconnection has completed. The disconnection micro-protocol is intimately connected with the send micro-protocol that controls the sending of packets and manages queues of outgoing messages. We omit the details of send, but essential features are mentioned.

```
micro-protocol DROP BASE STATION {
  exports{
    event   End_Disconnect(member_t MH, bs_t newBS);
  }
  imports{
    event   Start_Disconnect(member_t MH, bs_t newBS);
    proc      DropPackets(member_t MH);
  }
  actions{
    Start_Disconnect(member_t MH, bs_t newBS)  →
      /* Throw away packets that can not be delivered */
      DropPackets(MH);
      raiseEvent(End_Disconnect, MH, newBS);
  }
} end micro-protocol DROP BASE STATION
```

Figure 6.15: Drop packet disconnection scheme for base stations

DROP BASE STATION (Figure 6.15) discards packets of a host that has left the cell. The `DropPackets` procedure is imported from the send micro-protocol and deletes the host's packets from all queues. As noted above, if reliable transmission is needed then a higher-level protocol must be providing the guarantees.

DRAIN BASE STATION (Figure 6.16) takes the view that the host may still be reachable even if a handoff is occurring. The micro-protocol takes advantage of this by adjusting the priority of any remaining host packets and quickly transmitting them as the handoff occurs. The `Top_Priority_And_Send_All` event prompts the send micro-protocol to set the packets for quick delivery and if they are not acknowledged, to discard them. The event is raised synchronously, so that the micro-protocol will block until the send of the packets is completed or the packets are dumped.

FORWARD BASE STATION (Figure 6.17) removes packets from a departed host from the sending queues of the old base station and forwards them to the new one. The new base station then inserts them in the correct sending order, and adds them to the queues of outgoing messages for the mobile host. When all the packets have been forwarded, the `End_Disconnect` event is raised.

```
micro-protocol DRAIN BASE STATION {
  exports{
    event   End_Disconnect(member_t MH, bs_t newBS);
    event   Top_Priority_And_Send_ALL(member_t MH);
  }
  imports{
    event   Start_Disconnect(member_t MH, bs_t newBS);
  }
  actions{
    Start_Disconnect(member_t MH, bs_t newBS)  →
      /* Throw away packets that can not be delivered */
      raiseEvent(Top_Priority_And_Send_All, SYNC,MH);
      raiseEvent(End_Disconnect, SYNC, MH, newBS);
      }
} end micro-protocol DRAIN BASE STATION
```

Figure 6.16: Drain disconnection scheme for base stations

6.4 Variations of Quality of Service

Quality of service (QoS) micro-protocols can be separated from mobility micro-protocols since they are not directly involved in handoffs and other mechanics of routing. This separation is important for creating communication services with differing quality of service policies, but the same basic system architecture for routing and handoffs. Recall from Chapter 2 that InfoPad is the only current system that specifically addresses quality of service, largely because it is specifically designed for multimedia applications.

QoS for mobile systems is complicated because the negotiation between applications, mobile hosts, and base stations cannot necessarily be done once, as in stationary distributed systems. Mobile hosts move, of course, which means that a new base station must provide these resources when a handoff occurs. If the cell is already being heavily used, the new arrival with its connections may be be too much for the new base station to guarantee. The overall result is that quality of service must be dynamically renegotiated many times over the lifetime of a connection.

To aid the negotiation, QoS attributes are part of the connection parameters between a mobile host and an application, and migrate with the connection as the host moves. The connection contains a description of performance parameters, such as throughput, jitter, latency, packet retransmission limits and requirements, and perhaps transmission priority. These descriptions are passed during a handoff to the new base station along with the mobile host's connections to applications. Agent-based architectures cache the connection parameters in the agent, so in this case base stations contact the agent directly for performance parameters. These parameters are modified by QoS algorithms to realize different qualities of service.

We have identified four classes of properties related to QoS in mobile systems: scope, authority, locality of scheduling, and information types. Each constitutes an individual orthogonal aspect independent of other selections.

```
micro-protocol FORWARD BASE STATION {
  exports{
    event   End_Disconnect(member_t MH, bs_t newBS);
  }
  imports{
    event   Start_Disconnect(member_t MH, bs_t newBS);
    proc      getPacketFromQueues(member_t MH);
  }
  actions{
    Start_Disconnect(member_t MH, bs_t newBS)  →
      /* Forward packets to new base station for delivery */
      while (packet = getPacketFromQueues(MH)) {
        msg = createForwardPacket(packet, newBS);
        sendMsg(FORWARD, newBS, msg);
      }
      raiseEvent(End_Disconnect, MH, newBS);
  }
} end micro-protocol FORWARD BASE STATION
```

Figure 6.17: Forward packets disconnection schemes for base stations

Scope. An algorithm that governs QoS can either use information from one cell or information from neighboring cells. An algorithm of single cell scope would adjust priorities and allocations only in a single base station. However, an algorithm of multiple cell scope could use information from neighboring cells to do, for example, load balancing. That is, if a host is reachable by two different base stations, the algorithm could assign the mobile host to the more lightly loaded cell. This idea can be generalized to multiple overlapping cells.

Authority. QoS parameters are maintained by the *QoS authority.* Depending on the architecture and the algorithm, there are several choices of QoS authority. A common choice is the current base station, which caches all connection data for hosts in its cell. When using the base station authority, QoS information is either passed automatically as part of the handoff or requested from the old base station, as described above. Note that this information may only be a reflection of what the base station allocated to the connection, not necessarily what the host originally requested. A second option for mobile hosts that are autonomous is for the mobile host itself to be the authority. In this scheme, when a mobile host establishes a connection, it passes this information along to the new base station. A third possibility is the applications, in which case new base stations request QoS information from the application for each connection associated with a mobile host that enters its cell. In general for all three options, applications participate in allocation decisions, because they are knowledgeable about what parameters they require and how requirements can be adjusted. For example, video applications could respond to a reduced throughput allocation by slowing the frame rate or reducing the resolution.

Locality of Scheduling. The scheduling of resources (allocation) within a single cell by a base station can be local or global depending on whether the algorithm affects only one host or all hosts. Local scheduling allocates a share of resources to a mobile host and its applications that may be adjusted as requirements change or more resources become available. However, using local scheduling, an increase in the allocation of one host does not affect another host, i.e., regardless of later mobile host activity or arrivals, a host would be assured its allocation. Local scheduling also implies that late comers may not be able to acquire enough resources if all have already been allocated. On the other hand, global scheduling considers all the resources within the cell when making decisions, and can make adjustments to existing allocations. Thus, global scheduling algorithms may reduce the resources allocated to a mobile host to accommodate a new arriving host or a current host that starts running additional applications that require additional resources.

Information Types. Two types of information can be used for resource scheduling: request and usage. Request-based scheduling uses only requirements stated by applications and mobile hosts. In contrast, usage-based monitors actual use instead of or in addition to request-based information, which can result in better overall service to all hosts in the cell. If a guarantee of service to accommodate bursty traffic has been made, then the allocation must be request-based, and an application may end up acquiring larger allocations than actual usage patterns would dictate.

6.5 Example QoS Micro-Protocols

In this section, we describe generic high-level micro-protocols for QoS. Instead of a specific algorithm, the micro-protocols that are presented provide structure for adding QoS into a communication service for mobile systems. In this sense, the QoS micro-protocol is treated as a "black box", and we concentrate on describing the connections and relationships to other modules in the micro-protocol suite. The specific micro-protocols presented are to manage base station QoS and transmission QoS attributes by a base station or a mobile host during handoffs. The micro-protocols for managing sending of messages and usage monitoring are essential to QoS, and are described in Section 6.6.

The most complex QoS management occurs in the base station micro-protocol that allocates resources to mobile hosts and applications, negotiates QoS attributes, and dynamically adjusts QoS allocations. A generic micro-protocol, QOS BASE STATION, is given in Figure 6.18. It covers mobile hosts joining a cell, hosts leaving the cell, negotiating with applications for QoS attributes, and handling performance panics when QoS attributes are not being met.

In this micro-protocol, a number of situations are handled. When a host is released from the cell, the `Host_Released(MH)` event is raised and all allocations associated with the host can be given to other hosts as needed. The micro-protocol stores QoS attribute information of unmet requests to use in this situation. When hosts join a cell, the initial QoS attributes are assigned after negotiation and possible adjustments are made to allocations of other hosts. The negotiation may involve the monitor micro-protocol. The `LookupQoS` procedure determines what attribute values have been assigned to a host, so that this information can be passed along with the handoff if the QoS authority is the current base station. After QoS attributes are established, the QoS authority is informed.

When a host joins a cell, its QoS requirements may be passed as part of the handoff information; for example, the HANDOFF QoS BASE STATION micro-protocol (Figure 6.19) sends QoS attributes during a handoff. Other schemes where the QoS authority is the mobile host would use the HANDOFF QoS MOBILE HOST (Figure 6.20), which looks up QoS information when a handoff is occurring. In this case, the information would be included in the handoff message to the new base station. If the QoS authority is the application or agent, then the base station directly requests this information from the authority.

The QoS attributes received by the base station during a handoff are a temporary resource allocation that is used until the base station can negotiate. If these attributes acceptable — i.e., they can be met — then they are used permanently; otherwise, the base station negotiates to obtain achievable values. This negotiation can involve applications, modifying allocations for current mobile hosts, modifying other connections for the same mobile host, or use of monitor information to downgrade connections allocated but not being used. After the attributes are established, the QoS authority is informed of the new parameter values.

Anytime during the lifetime of a connection, an application can signal a desire for an adjustment to the QoS attributes by sending a `QOS_New_Request` message to the base station. The message can come directly from an application or from an agent on behalf of an application. This request will be met if possible, and a response sent back along with a message to the QoS authority about the change in parameters.

In certain situations, the monitor or send micro-protocols may alert the QoS micro-protocol that certain QoS attributes are not being maintained. This can occur either because a mobile host or application is not adhering to the negotiated values or because the resources have been over-scheduled. When this occurs, the QoS micro-protocol can modify QoS parameters and start a new negotiation phase or ignore the panic.

```
micro-protocol QOS BASE STATION {
  exports{
    proc     LookupQoS(member_t MH, QoS_t *QoS);
    event    QoS_Modified(member_t MH, QoS_t *QoS);
    event    QoS_Added(member_t MH, QoS_t *QoS);
  }
  imports{
    event    Host_Joined(member_t MH, QoS_t *QoS);
    event    Host_Released(member_t MH);
    event    QoS_Panic(Panic_Status_t status);
    proc     UsageMonitorData(member_t MH, use_t *Usage);
  }
  private{
    table_t TableOfAllocations;
  }
  initialize{
    ClearTableOfAllocations();
  }
  actions{
    Host_Released(MH) →
      Remove mobile host from TableOfAllocations, cleanup allocations given to host,
      perhaps increase other connection allocations
    Host_Joined(MH,QoS) →
      if (QoS == NULL) {
        /* no QoS information with the handoff, start with default */
        InsertTable(TableOfAlloactions, DEFAULT_QOS) ;
        Other schemes can request QoS information from QoS Authority
        for initial allocation
      }
      else {
        If the initial QoS can be accommodated then insert into table.
        Otherwise apply negotiation algorithm to attain new QoS attributes
        employing any of the following methods.
          1. Negotiate with application to reduce requirements
          2. Change the allocation of other MH's and inform hosts and applications
          3. Change allocation of the other connections of this MH
          4. Use monitor information to downgrade connection not used to allocation
        Inform QoS Authority of final QoS parameters
        raiseEvent(QoS_Added, ASYNC, MH, QoS);
      }
    Message_Popped_To_CP(CP_Msg_t Msg) & Msg->type == QOS_New_Request  →
      /* Application wants to request different QoS requirements */
      Apply above algorithm with new QoS from Msg->QoS
      Send final QoS parameter message back to application
      Inform QoS Authority of final QoS parameters
      raiseEvent(QoS_Modified, ASYNC, MH, QoS);
    QoS_Panic(status) →
      Send unable to meet QoS requirements, missed deadlines, etc.
      Consult with monitor to isolate difficulty and modify QoS attributes
      Possibly send additional QoS messages to applications
  }
  ... code for lookupQoS, calculateQoS
} end micro-protocol QOS BASE STATION
```

Figure 6.18: Quality of service management

```
micro-protocol HANDOFF QOS BASE STATION {
  imports{
    event   QOS_Handoff(member_t MH, QoS_t *QoS);
    proc      LookupQoS(member_t MH, QoS_t *QoS);
  }
  actions{
    QOS_Handoff(member_t MH, QoS_t *QoS) →
      LookupQoS(MH, QoS);
    Message_Popped_To_CP(CP_Msg_t Msg) & Msg->type == QOS_Info_Request  →
      LookupQoS(MH, QoS); /* get current QoS characteristics */
      sendControlMsg(QOS_Info, Msg->sender, QoS);
  }
} end micro-protocol HANDOFF QOS BASE STATION
```

Figure 6.19: QoS information provided by base stations

```
micro-protocol HANDOFF QOS MOBILE HOST {
  imports{
    event   QOS_Handoff( QoS_t *QoS);
    proc      LookupQoS(QoS_t *QoS);
  }
  actions{
    QOS_Handoff(QoS_t *QoS) →
      /* Fill in QoS structure */
      LookupQoS(QoS);
  }
} end micro-protocol HANDOFF QOS MOBILE HOST
```

Figure 6.20: QoS information provided by a mobile host

6.6 Supporting Micro-Protocols

The QoS micro-protocols are intimately connected to the send and monitor micro-protocols. The first is responsible for translating the QoS information to priorities for scheduling transmission of messages to mobile hosts. In particular, it multiplexes the shared resource of wireless bandwidth among all the mobile hosts in the cell. Some wireless transmission protocols assign each mobile host separate channels that can simplify scheduling, but even then, multiple applications using the same channel must be scheduled.

The monitor micro-protocol is essential for implementing usage-based QoS policies, as noted above. Unlike the send micro-protocol, monitor is a passive observer. It collects information about the aggregate allocation of resources in the cell, the usage patterns for each mobile host, and whether real-time deadlines for jitter and throughput are being met. The monitoring also determines which hosts are not using their allocated resources, so the QoS module can reassign them to other connections or adjust sending priorities. The monitor module supports global scheduling by answering queries from its peers in

Inspiration	Detect	Handoff	Disconnect
Crosspoint	ICMP	NACK	FORWARD
PARC TAB	BEACON	AGENT	DROP
InfoPad	SIMPLE, TRANSLATE, AUTONOMOUS	REQ/REP	DRAIN
DataMan	LAZY, TRANSLATE, AUTONOMOUS	REQ/REP	FORWARD

Table 6.1: Existing mobile system inspired configurations.

neighboring cells to implement load balancing.

For all micro-protocol suites, a membership micro-protocol is needed to add and remove hosts from the active table. In addition, verification micro-protocols are included to check the format of a message prior to being added to the shared bag of messages. Some verification micro-protocols also drop messages from hosts not in the table unless it is a control message requesting addition to the cell.

6.7 Example Configurations

Combining the micro-protocols described in this chapter together can result in a variety of communication services for mobile systems. All together there are 5 choices for detection, 3 choices for handoffs and 3 choices for disconnection, resulting in 45 possible composite protocols. The selection of QoS micro-protocols is orthogonal and adds another dimension of behavior that can be added to any of the 45 combinations. As an example of some of the possible choices, several configurations inspired by systems described in Chapter 2 are shown in Table 6.1. The table lists the detection, handoff, and disconnection micro-protocols that would be combined to create a system with mobility behavior similar to the named systems. In addition, a system such as InfoPad that includes QoS guarantees would include some variation of the QoS and supporting micro-protocols.

6.8 Conclusions

In this chapter, we have seen how mobile systems can benefit from configurable communication services. Micro-protocols for detection of handoffs, handoffs, and disconnection can be selected and configured to match a variety of differences in the architecture and semantics, or to facilitate rapid protocol development. In addition, the micro-protocol approach allows for incremental system construction, which is essential in an experimental field. We also showed several different configurations using this one collection of micro-protocols that result in semantics similar to existing systems. Finally, quality of service issues and policies are separated from those concerned directly with mobility in this approach, thereby simplifying system design and construction.

CHAPTER 7
EVALUATION

The goals of this research as stated in Section 3.1 are broad both in scope and in character, which makes conclusive statements and quantitative measurements against objective standards difficult. Furthermore, limitations in the programming and execution environments constrained experimentation in several important areas. Nevertheless, our experience allows us to reach several tentative conclusions about the effectiveness of our programming model for constructing communication services, and the viability of the prototype implementation.

The next section presents our overall assessment of this research. Following is an explanation of the limitations imposed by the current implementation and programming environment, and how they might be overcome. We conclude by discussing additional issues that arise when supporting mobile systems with real-time requirements and evaluating related work.

7.1 General Assessment

7.1.1 Overview

Our *x*-kernel-based prototype implementation is an effective realization of the composite protocol model and achieved the majority of the objectives described in Section 3.1. The model has been used to design four widely-differing communications services — group RPC, membership, atomic multicast [GBB+95], and mobile communication — with the first three implemented by three different people. Based on these experiences, we have reached several conclusions. First, the system is indeed configurable, with construction of a new service being no more complicated than re-linking the framework with different micro-protocol object files. Second, our performance measurements suggest that the event-driven, data-centered approach is not only a useful design tool, but a viable implementation technique as well. Third, the prototype successfully interfaced with existing *x*-kernel protocols without modification, which makes it possible to use in a variety of settings and for a variety of applications.

Finally, and perhaps most importantly, the composite protocol model, like the *x*-kernel itself, simplifies development and debugging by encouraging protocol designers to decompose protocols into more manageable pieces. In order to implement and test each component separately, designers are forced to minimize the amount of external state and interaction required between components, which helps identify more precisely the distinct semantic categories contained in their protocol specifications. As a result, communications services built from micro-protocol suites really do seem to encapsulate specific semantic properties much better than existing alternative implementation forms.

We now examine in greater detail how well the goals of efficiency, reusability, ease of debugging and maintenance, and explicit dependencies have been achieved.

7.1.2 Efficiency

Efficiency can be measured in many different ways. In Chapter 5, we presented measurements of protocol performance for the group RPC micro-protocol suite. Given the limitation of the environment — a relatively old and slow hardware platform, and execution as a user-level task on top of Mach MK82 — the execution times were encouraging and lead us to believe that micro-protocol suites constructed using the event-driven model can be competitive with other system architectures.

Another approach to ascertaining performance would be to make comparisons with similar existing systems. However, this introduces a number of problems. For example, we must first determine which metrics would be appropriate to compare. In addition, for the results to be fair, the other systems must have similar design goals. Unfortunately, as discussed in Chapter 2, there are few systems that provide similar functionality, and those that do run on hardware or OS platforms that are sufficiently different to prevent meaningful comparisons.

Finally, if further experimentation convinces us that performance of the existing prototype is inadequate, we are aware of several areas where enhancements and optimizations are possible. These are discussed in Section 7.2.

7.1.3 Resuability

Our initial experimentation has focused on demonstrating the wide range of communications services that can be constructed using the composite protocol model. Consequently, we have had little opportunity to experiment with reusing micro-protocols in different protocol suites. However, from studying the structure of the group RPC suite (Section 5.1), we believe that many of these micro-protocols could be used in other suites, particularly multicast. Alternatively, by providing additional semantic choices using additional micro-protocols, the group RPC suite could be generalized to create a reliable group communication suite.

7.1.4 Ease of Debugging and Maintenance

Building a high-level communication service is a difficult task. One of the greatest benefits of the prototype has been the opportunity to build and test elements of the communication service individually. Each semantic behavior, coded as a separate micro-protocol, can be tested either stand-alone or in peer-to-peer communication using the existing x-kernel protocol graph. Because each micro-protocol is relatively small, tracking down bugs is comparatively easy.

In addition, the framework provides a number of built-in debugging aids. One of the most useful is an event-level tracing facility that reports all event triggers along with their arguments, and the sequence of event handler execution. This service is entirely provided by the framework, so no special code is required in the micro-protocol. Similarly, message creation and destruction, and changes to message attribute values can be tracked. It is also possible to trace thread execution using debugging facilities in the x-kernel.

The composite protocol model, with its clearly expressed dependencies, also aids in program maintenance. First, changes are generally confined to one micro-protocol at a time, greatly reducing the amount of code that must be examined to find an error. Second, if the error is due to an unexpected interaction with other micro-protocols, it can easily be checked by tracing event execution and message attribute values. Contrast this to the normal protocol construction and debugging procedures, where no such corresponding high-level interactions can be identified, much less traced.

7.1.5 Explicit Dependencies

The micro-protocol structure makes dependencies explicit and clearly visible. Events and message attributes — the external interface to a micro-protocol — are defined at the beginning of the code. When a micro-protocol imports an event, it clearly indicates that it is relying on some other micro-protocol to generate that event.

By making these elements part of the language definition, it is straightforward for a language translator (or smart linker) to verify that these expectations are in fact being met. Unfortunately, in our prototype implementation micro-protocols must be hand-translated into the implementation language (C), so only limited checks are performed. Section 7.2 discusses this in more detail.

One important semantic aspect not captured by the interface definition is event ordering, i.e., that event A must be processed before events B or C. Also not expressed is the relationship between events and changes to message attributes. Concisely expressing these constraints in the interface would make it easier to understand the behavior of the micro-protocol, and potentially permit some level of automated verification, either at translation time or at runtime.

7.2 Programming Issues

7.2.1 Synchronous and Asynchronous Event Execution

The framework implementation has combined two orthogonal characteristics, call semantics and execution semantics, into synchronous or asynchronous styles of event execution. Recall that call semantics can either be blocking (i.e., calls do not terminate until all handlers terminate) or non-blocking (i.e., calls return immediately). Similarly, event handlers can execute either sequentially or in parallel. In the framework, raising an event with the synchronous parameter results in a blocking call and sequential execution, while raising an event with the asynchronous parameter results in a non-blocking call and parallel execution. Figure 7.1 shows the four possible combinations and the two choices that are supported by the framework, labeled ASYNC and SYNC.

Given that only two combinations are supported, the natural question is whether the other two might be useful. Blocking call style with parallel execution would be useful for a micro-protocol that can only proceed when all handlers complete execution but the handlers do not have to execute sequentially. For example, the framework event `Message_Ready_To_Be_Sent` must be handled by all micro-protocols before actually sending the message, but the micro-protocols need not execute sequentially. The last combination,

Call Style	*Handler Execution*: Sequential	Parallel
Blocking	SYNC	
Non-Blocking		ASYNC

Figure 7.1: Possible combinations.

non-blocking sequential style, could be used when there are dependencies between micro-protocols, but no need for notification when the handlers complete. To allow variation conveniently, perhaps call and execution semantics should be specified independently.

7.2.2 Call Depth

In the course of constructing and experimenting with the prototype, we learned some surprising lessons about the interactions of events and handler execution with the procedure call optimization. In particular, when a single thread is used in this way, asynchronous event execution is sometimes required to make computational progress, and repeated synchronous event execution can cause stack overflow. The basic problem is that event handlers can raise other events that cause nested event handlers to execute, which results in the thread executing to a great depth without returning. In other words, the execution takes on a depth-first execution style, executing all nested events first before completing the execution of handlers in the outermost events. Favoring nested events in this way can prevent handlers associated with other triggered events from running.

Stack overflow is a straightforward problem that results from long execution chains of nested events handlers. For example, consider the following scenario. A message arrives from the network, which causes all event handlers registered for the message arrival event to be scheduled for execution. One of these handlers reacts to this arrival by delivering messages to the user, for instance, if the message was the missing predecessor in a context graph. This in turn may cause a cascade of new messages from the user and more handlers being scheduled. Since all these activities are executed with the same thread, stack overflow may result.

The need for asynchronous event execution to make progress is more subtle but arises from the same depth-first execution scenario. Specifically, if the triggering of an event results only in a hander being added to the ready queue (to be executed later using fair scheduling), then all handlers will eventually execute. However, with the procedure call optimization, handlers that appear later in the list may be indefinitely postponed. Also, since the single call thread can continue to execute nested events and their corresponding handlers, new messages that arrive may not even be processed in a timely manner. Our experimentation with the group RPC micro-protocol suite demonstrated this problem; often a message was sent requesting a retransmission because no thread had run to retrieve arriving messages. In other words, message retransmissions were requested for messages the composite protocol possessed but was unaware of! The use of true asynchronous event execution with multiple threads solves these problems by servicing event handlers fairly, thereby allowing handlers associated with incoming messages to execute.

The call depth optimization discussed in Section 4.2.4 was introduced as a way to, in essence, force asynchrony. Recall that when using this optimization, a new thread is created when the call depth exceeds a threshold set by the user. After implementing call depth control, we noticed that GRPC made many fewer retransmission requests. Changing the call depth threshold significantly affected the running time, and in fact, we tuned this value to achieve the best results from our micro-protocol suite. If the call depth was set fairly shallow, 5 to 10, event execution caused more C-thread context switches, but fewer extra retransmission requests were sent. When the call depth was set fairly deep, 50 to 100, event execution was more rapid, but far more control message traffic was generated, which resulted in significant performance degradation as the network became severely overloaded. The best results for GRPC were achieved with a call depth setting around 30, which achieved the best tradeoff between the number of retransmissions requests and C-thread context switches.

Call depth monitoring is a simple fix that solved the immediate problem, but perhaps a priority scheduling scheme is a better long-term solution. The need for this type of scheduling control is discussed in Section 7.4.

7.2.3 Ordering Handler Execution

The current way in which synchronous event execution is realized allows the programmer to assume a given ordering for its associated handlers. Specifically, execution order is the order in which the handlers were installed. Knowing the execution order can be used to advantage, for example, when a second handler depends on data modified by the first. If ordered execution is not guaranteed, then the first event handler would have to explicitly trigger the second using an intermediate event. This issue is irrelevant for asynchronous event execution since in this case, handlers are executed concurrently with no guaranteed order.

The primary advantage of ordering handler execution is that the micro-protocol code can be simplified since intermediate events do not have to be used. Without the need to include minor events that capture only small state changes in the micro-protocols, the protocol writer can concentrate on structuring the code to support cleanly the major events that drive execution of the communication service. Also, the intermediate events introduce an extra layer of indirection that could potentially degrade performance.

On the other hand, ordering handler execution in this way has the major disadvantage of violating the modularity of micro-protocols. That is, ordering adds hidden dependencies that are not captured clearly in the specification of the micro-protocol as captured in the list of imported and exported events. This makes it difficult to reconfigure micro-protocol suites, since when a new micro-protocol is added or replaced, the execution order must be adjusted to accommodate this change. As the number of micro-protocols increases, this situation worsens due to all the possible configurations and corresponding orderings. In addition, using a micro-protocol from one suite in another becomes almost impossible. Since intermediate events can always be used to force handler dependencies, ordered handler execution adds no additional functionality, so its use is primarily a question of programming style.

While ordering of synchronous event execution is supported in the framework to enhance flexibility, on balance, our feeling is that its use should be avoided. By doing so, the protocol writer preserves the explicitness of dependencies, thereby enhancing the overall configurability of the communication service. We note that the RPC micro-protocol suite was written without relying on implicit ordering, and as a result, uses a more concurrent style of programming.

7.2.4 Event Scheduling

Another problem that was encountered during experimentation was the lack of control over event scheduling. For example, when two messages arrive and the corresponding events are raised, it would be useful to have a mechanism to control which is handled first. This could be used to ensure that the messages are processed in sequence number order (transmission order), thereby avoiding unnecessary retransmission requests. However, such control requires scheduling support from the underlying system, which is unfortunately lacking in our version of Mach.

Scheduling control is also essential for correctness in a multiprocessor environment. If two threads are popping up ordered messages to an application, it is essential that they execute in the order that these pops were issued. Otherwise, the messages could have been ordered by the composite protocol and the pops executed in correct order, but the messages actually arrive at the application in the wrong order. This kind of scenario is again difficult to prevent without support from the underlying system.

7.2.5 Programming Language Support

As mentioned in Chapter 3, there is currently no translator for the Protocol Description Language (PDL) used for examples in this dissertation. This leads to several limitations in our prototype, which can be grouped into three categories: obscuring the model, no static checking of PDL language rules, and no opportunity for optimizations.

The first is perhaps the most serious, since the key elements to our paradigm — events, handlers, messages, and message attributes — disappear when a protocol is coded in C, becoming ordinary function calls and definitions. This makes the essential elements of a micro-protocol's interface and behavior much less visible and complicates debugging.

The second problem is that language checks cannot be done without a translator. For example, we had to rely on C visibility rules that make functions and global variables public by default, where the opposite is true in PDL. Similarly, there are no shared global variables in PDL since this would create implicit dependencies between micro-protocols, while C's file-level scoping makes it impossible to enforce this restriction if a micro-protocol is split into multiple source files. It is also impossible to perform a number of module-specific checks, which could have facilitated adding a new micro-protocol and configuring new communication services. For example, a translator could issue a warning when events are raised but not handled, or when handlers are declared but no micro-protocol generates the corresponding event. This type of feedback can simplify the process of detecting incompatibilities between micro-protocols, and therefore, make configuration of micro-protocol suites more automatic.

Finally, a translator would provide an opportunity to optimize event execution with efficient evaluation of guards on handlers. Without this feature, each handler must evaluate the guard itself and exit if it evaluates to false. This would be better done before handlers are invoked to eliminate the need to process the event and execute the handler in all cases. Guard evaluation also provides other opportunities for optimization, such as evaluating a guard only once if several handlers have identical conditions. Event execution can also be optimized using techniques such as in-lining code to remove event handler execution overhead completely.

On a related issue, micro-protocols are actually objects, so an implementation in an object-oriented language would naturally provide benefits. As noted in Section 3.1, a C++ prototype using a simulated network has been developed in which micro-protocol, events, and messages are classes. Further explorations in this direction would be useful. For example, micro-protocols could be structured into specialized classes, such as ordering micro-protocols that represent orthogonal behaviors. This may help the user identify which micro-protocols may be substituted for one another.

7.3 Experimentation Issues

7.3.1 Performance Profiling

Performance profiling posed a number of challenges given the experimentation environment. Round trip message tests are the typical way to measure performance of network protocols. However, since our composite protocol is a mix of framework and user-supplied micro-protocol code, we would also like to measure the cost of different framework functions. Unfortunately, due to the lack of profiling tools on Mach, we developed only vague ideas of what percentage of execution time is involved in framework procedures versus micro-protocol specific code. This hampered our ability to improve runtime performance, which naturally should be based on reliable profiles of individual routines.

Also, as noted in Section 7.1.2, we would like to compare our performance with other comparable protocol suites. However, this is difficult because other suites are not based on the same hardware or operating system. Moreover, the DecStations used for testing are processors that run at 40 MHz and 25 MHz, which is very slow by modern standards. More modern hardware and operating systems platform would be desirable, and might provide a common basis for comparison with other approaches.

7.3.2 Testing

Running tests of composite protocols to gather data is the same as running any other *x*-kernel protocol suite, although testing all the different combinations of micro-protocols clearly involves more effort. For testing of individual modules, the configurability of the approach makes things easier because each can be tested incrementally. Tests are started with a weaker set of semantics requiring fewer micro-protocols, and once those are completed, stronger semantics and additional micro-protocols can be added. For example, acknowledgment and retransmission micro-protocols are very easily verified individually so these might be tested first. Then ordering micro-protocols might be added. This also has the additional benefit that the composite protocol provides a working system throughout and thus can serve as a testbed for the new micro-protocols.

Another advantage of our composite protocol approach is that the framework provides a convenient place to implement facilities needed for event-based testing. In particular, the framework can report all event trigger occurrences and handler executions, which provides valuable information for debugging micro-protocol suites. In addition, the framework allows specification of what level of debugging messages are printed, or if only events specific to one micro-protocol are reported. The framework also has a command line interface to a test program that can trigger events to simulate event generation. This is useful for testing individual micro-protocols.

7.3.3 Use of the *x*-kernel

Using the *x*-kernel had both positive and negative aspects. On the positive side, the *x*-kernel is specifically designed for network programming and experimentation with protocols, which allowed us to focus on our model and not deal with such details as integration with network device drivers. We were also able to exploit its other facilities such as the efficient message tool and novel thread architecture, both of which simplified our implementation effort.

Another advantage was that the standardized protocol interface simplified the integration of the composite protocol with the *x*-kernel protocol graph. Specifically, the composite protocol only needed to support a few operations that make up the *x*-kernel uniform protocol interface. Moreover, when a message is brought into the composite protocol via a pop or push operation, the framework can control further execution, which allows the composite protocol to enforce its own execution model within the composite protocol. That is, once the composite protocol handles messages that cross protocol boundaries, the framework can completely control the form of messages and handling of these messages by micro-protocols. We were also able to augment the *x*-kernel messages easily with attributes to create CP messages.

On the negative side, it was sometimes difficult to isolate the micro-protocol programmer from all the details associated with writing protocols for the *x*-kernel. For example, one of the goals of the implementation was to create a composite protocol that could be completely independent of the protocol immediately below it in the protocol stack. However, in our prototype this is only partially true; we have managed to localize the layer-dependent code to one user-modifiable file, but not eliminate it completely. Most of the difficulties are associated with specifying the participant addresses used to open up *x*-kernel sessions for the lower-level protocol. Perhaps a better environment would have provided a generalized mechanism for specifying the participant addresses in a protocol-independent fashion.

Another shortcoming of using the *x*-kernel is the lack of a multiprocessor implementation. While there are two multiprocessor versions of the *x*-kernel [NYKT94, Bjo93], they are not widely available and are not the standard distribution. In addition, few existing x-kernel protocols are written for multiprocessors, so for compatibility, the framework is restricted to sequential execution. However, our model is specifically designed to allow micro-protocols to execute in parallel when asynchronous event execution is used. Thus, the availability of a multiprocessor version of the x-kernel and appropriate hardware would have allowed for optimization and an assessment of parallel asynchronous handler execution.

7.4 Mobility and Real-Time

An issue that arose in the context of mobile computing concerns support for real-time deadlines and control of real-time aspects of scheduling. For example, the Crosspoint architecture uses a negative acknowledgment scheme to add new mobile hosts to a base station; if the new base station does not receive a NACK message, then it will add the host to its cell. There are two potential problems: for timeouts to work, timer events on the sender and receiver side need to execute fairly close to real time, and incoming messages must be serviced shortly after their arrival. The second problem arises because, as discussed in Section 7.2.4 above, the message may be held by a thread that is suspended and destined not to execute within the required time. A similar problem is that occurences of repeating events are not guaranteed to execute in the order they were triggered.

Another useful facility would be some mechanism for informing micro-protocols when tasks are not executed on time and deadlines are missed. For example, generating an exception for missed deadlines would allow micro-protocols to take some corrective action, such as increasing the period between task execution or adapting its behavior to a more loaded system. Since no information about missed deadlines is available, the system currently has no opportunity to correct behavior. A similar opportunity is to discard execution instances of repeating timer events that have missed their deadline, since such events are often of no use.

7.5 Related Work

A number of other papers have addressed areas related to this work. Several are in the area of fault-tolerance, where researchers have explored use of modularization or system customization. Examples include the ANSA system [OOW91] and the work on multicast reported in [Gol92]. In contrast to these, our approach is more general and provides more flexibility for the protocol designer. Also in the area of fault-tolerance, [Bla91] explores orthogonal properties of transactions. Such characterizations are complementary to our work since they suggest applications that might be suitable for implementation using our model.

Another area of related work concerns development of system support for constructing modular protocols. The x-kernel itself is, of course, one such system. Our work is an extension of the x-kernel model, with the goal of supporting finer-grain protocol objects that require richer facilities for communication and data sharing, while retaining the programming and configurability advantages of the x-kernel. Many of our goals related to system customization, code reuse, and protocol configurability are adopted from the x-kernel.

Other x-kernel related work has explored the use of finer-grain protocol objects [OP92], but the emphasis there is on syntactic decomposition of higher-level protocols within a hierarchical framework. This work, however, does lend credence to the claim that such fine-grain modularity can be introduced without sacrificing performance. System V Streams [Rit84] also supports modularization of protocols, but its model is also hierarchical and relatively coarse-grained. Horus [vRHB94] supports stack-line configurations of coarse-grained protocols.

Somewhat closer to our work is the ADAPTIVE system [SBS93], which is also designed to support flexible combinations of protocol objects. The goal of the system is to support efficient construction of transport services with different QoS characteristics, especially for multimedia applications using high-performance networks. In contrast with our work, the designers of ADAPTIVE emphasize runtime reconfiguration, automatic generation of *sessions*—i.e., instances of protocol objects—from high-level specifications, and support for alternative process architectures and parallel execution. Moreover, the type of protocol objects supported appear relatively coarse-grained when compared to our objects—multicast rather than individual properties of multicast, for instance—and more oriented toward hierarchical composition and limited data sharing.

Several other efforts have also concentrated on supporting parallel execution of modular protocols, including [GNI92, LAKS93]. While similar to our work in the sense of decomposing protocols along semantic lines, these efforts differ in their emphasis on using parallel execution to improve throughput and latency for high-performance scientific applications. They also retain a single-level composition model, which we believe does not offer enough flexibility for high-level protocols.

Protocol languages can be used to specify and validate protocols. The Language of Temporal Ordering Specifications (LOTOS) is based on the Calculus of Communicating Systems (CCS) [Bri87]. Lotos provides a high-level abstraction through the use of specification algebras that allow a designer to reason formally about protocol behavior. Estelle is another formal description language designed to program reactive systems [BD87]. The language execution model is based on an extended finite state machine, where protocols are described as a set of modules that contains responses to events and affect the environment through output events. In contrast to our work, these efforts concentrate on automatic validation of protocols.

Finally, as noted in Chapter 2, recent work on new generation operating systems has emphasized similar customization goals, but in a more general context [BCE+95, HPM93, MMO+94]. These projects attempt to increase the ability of users to configure different types of services, but for many aspects of operating system functionality rather than just network protocols. However, the configurability they provide is typically more coarse-grained than our approach, which emphasizes choice among specific semantic properties of high-level protocols.

7.6 Summary of Contributions

This dissertation makes a number of contributions to the study of communication services for distributed systems. The primary contribution is a new model for constructing configurable communication services that can be customized to meet the specific requirements of a distributed application. The approach is novel because communication services are decomposed into distinct semantic properties, each implemented by a fine-grained micro-protocol. Micro-protocols have well-defined interfaces and interact according to an event-driven paradigm.

Another significant contribution is an x-kernel based implementation that supports our model. The implementation extends the standard hierarchical model of the x-kernel with a composite protocol in which micro-protocol objects are composed with a standard runtime system. Using this implementation, we constructed a GRPC micro-protocol suite that can be configured to provide many customized variations of a group RPC service. The GRPC suite also provides a measure of the implementation cost of the event-driven model and an assessment of the incremental costs of communication properties for common group communication paradigms.

Finally, we demonstrated the widespread applicability of the approach by designing a suite of micro-protocols for mobile computing. We discovered general semantic properties for mobile computing, and designed micro-protocols for negotiation of quality of service, and detection, handoff, and disconnection, These can be combined to accommodate a variety of mobile computing architectures and applications.

REFERENCES

[AB93] A. Acharya and B. R. Badrinath. Delivering multicast messages in networks with mobile hosts. In *Proceedings of the 13th IEEE Symp. on Distributed Computing Systems*, pages 292–299. IEEE, May 1993.

[ABI93] A. Acharya, B. R. Badrinath, and T. Imielinski. Checkpointing distributed applications on mobile computers. Technical report, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, 1993.

[ABSK95] E. Amir, H. Balakrishnan, S. Seshan, and R. Katz. Efficient TCP over networks with wireless links. In *Proceedings of the HotOS-V Workshop*, Orcus Island, May 1995.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd IEEE Symp. on Fault-Tolerant Computing*, pages 76–84, Boston, July 1992.

[AGKK91] J. Auerbach, M. Gopal, M. Kaplan, and S. Kutten. Multicast group membership management in high speed wide area networks. In *Proceedings of the 11th IEEE Symp. on Distributed Computing Systems*, page 231, Arlington, TX, May 20-24 1991.

[AGSW93] N. Adams, R. Gold, B. Schilit, and R. Want. An infrared network for mobile computers. In *Proceedings of the 1st USENIX Mobile and Location-Independent Computing Symp.*, pages 41–51, August 1993.

[AIB] A. Acharya, T. Imielinski, and B. R. Badrinath. DATAMAN project: Towards a Mosaic-like location-dependent information service for mobile clients. Technical Report DCS-TR-320, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

[ALB88] E. Arthurs, T.T. Lee, and R. Boorstyn. The architecture of a multicast broadband packet switch. Technical report, Bell Communications Research, Morristown, NJ 07960, 1988.

[AP93] M. B. Abbott and L. L. Peterson. Increasing network throughput by integrating protocol layers. *IEEE/ACM Trans. on Networking*, 1(5), October 1993.

[ATK91] A. L. Ananda, B. H. Tay, and E.K. Koh. ASTRA — An asynchronous remote procedure call facility. In *Proceedings of the 8th IEEE Symp. on Distributed Computing Systems*, pages 172–179, Arlington, Texas, May 1991.

[BA89] N. E. Belkeir and M. Ahamad. Low cost algorithms for message delivery in dynamic multicast groups. In *Proceedings of the 9th IEEE Symp. on Distributed Computing Systems*, pages 110–119, Newport Beach, June 1989. IEEE.

[BAI93a] B. R. Badrinath, A. Acharya, and T. Imielinski. Impact of mobility on distributed computations. *ACM Op. Syst. Review*, 27(2):15–20, April 1993.

[BAI93b] B. R. Badrinath, A. Acharya, and T. Imielinski. Structuring distributed algorithms for mobile hosts. Technical Report DCS–TR–298, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903, 1993.

[BALL90] B. Bershad, T. Anderson, E. Lazokska, and H. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 6(1):37–55, February 1990.

[BB95] A. Bakre and B. R. Bandrinath. Handoff and system support for Indirect TCP/IP. In *Proceedings of the 2nd USENIX Mobile and Location-Independent Computing Symp.*, pages 11–24, April 1995.

[BBIM93] B. R. Badrinath, A. Bakre, T. Imielinski, and R. Marantz. Handling mobile clients: A case for indirect interaction. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*. IEEE, October 1993.

[BC91] K. Birman and R. Cooper. The ISIS project: Real experience with a fault-tolerant programming system. *ACM Op. Syst. Review*, 25(2):103—107, April 1991.

[BCE+95] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer. SPIN - an extensible microkernel for application-specific operating system services. *ACM Op. Syst. Review*, 29(1):74—77, January 1995.

[BCG91] K. Birman, R. Cooper, and B. Gleeson. Programming with process groups: Group and multicast semantics. Technical Report 91-1185, Department of Computer Science, Cornell University, January 1991.

[BD87] S. Budkowski and P. Dembinski. An introduction to Estelle: a specfication language for distributed systems. *Computer Networks and ISDN Systems*, 14:3–23, 1987.

[Bir85] K. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the Tenth ACM Symp. on Operating System Principles*, pages 79–86, Orcas Island, WA, December 1985.

[Bjo93] M. Bjorkman. The xx-kernel: An execution environment for parallel execution of communication protocols. Technical report, Uppsala University, June 1993.

[BKPV95] B. Bakshi, P. Krishna, D. K. Pradhan, and N. H. Vaiyda. Performance of TCP over wireless. Technical Report 95-049, Department of Computer Science, Texas A&M University, December 1995.

[Bla91] A. Black. Understanding transations in an operating system context. *ACM Op. Syst. Review*, 20(1):73–76, January 1991.

[BM89] K. Birman and K. Marzullo. The role of order in distributed programs. Technical Report 89-1001, Department of Computer Science, Cornell University, 1989.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984.

[Bri87] E. Brinksman. An introduction to Lotos. In *Proceedings of 7th IFIP WG 6.1 International Workshop on Protocol Specification, Testing, and Verification*, 1987.

[BSAK95] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the 1st ACM Conference on Mobile Computing and Networking*, November 1995.

[BSS91a] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. Technical Report 91-1192, Department of Computer Science, Cornell University, February 1991.

[BSS91b] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, August 1991.

[Car82] W. C. Carter. A time for reflection. In *Proceedings of the 12th IEEE Symp. on Fault-Tolerant Computing*, 1982.

[Car92] K. G. Carlberg. A routing architecture that supports mobile end system. In *Proceedings of IEEE MILCOM 1992*, October 1992.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th IEEE Symp. on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, June 1985.

[CFL94a] P. Cao, E. W. Felten, and K. Li. Application-controlled file caching policies. In *Proceedings of USENIX Summer 1994 Technical Conference*, 1994.

[CFL94b] P. Cao, E. W. Felten, and K. Li. Implementation and performance of application-controlled file cache. In *Proceedings of the First Operating Systems Design and Implementation Symp.*, 1994.

[CGR88] R. F. Cmelik, N. H. Gehani, and W. D. Roome. Fault Tolerant Concurrent C: A tool for writing fault tolerant distributed programs. In *Proceedings* *of the 18th IEEE Symp. on Fault-Tolerant Computing*, pages 55–61, Tokyo, June 1988.

[Che86] D. R. Cheriton. VMTP: A transport protocol for the next generation of communication systems. In *Proceedings of SIGCOMM'86*, pages 406–415, August 1986.

[CLR95] D. E. Comer, J. C. Lin, and V. F. Russo. An architecture for a campus-scale wireless mobile internet. Technical Report CSD-TR95-058, Purdue University, Department of Computer Science, 1995.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, August 1984.

[Coo85] E. C. Cooper. Replicated distributed programs. In *Proceedings of the Tenth ACM Symp. on Operating Systems Principles*, pages 63–78, Orcas Island, WA, 1985.

[Coo90] E. C. Cooper. Programming language support for multicast communication in distributed systems. In *Proceedings of the 10th IEEE Symp. on Distributed Computing Systems*, pages 450–457, Paris, France, 1990.

[Cou81] Courier. Courier: The remote procedure call protocol. Technical Report XSIS 038112, Xerox System Integration Standard, Stamford, CT, December 1981.

[CPR92] D. Cohen, J. Postel, and R. Rom. IP address and routing in a local wireless network. *IEEE INFOCOM*, 1992.

[CR94] D. E. Comer and V. F. Russo. Using ATM for a campus-wide wireless internetwork. In *Proceedings of the 1994 IEEE Workshop on Mobile Computing*, 1994.

[Cri89] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. Technical Report Research Report RJ 7203, IBM Almaden Research Center, December 1989.

[DC90] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Trans. Comput. Syst.*, 8(2):85, May 1990.

[Dee94] S. Deering. Internet multicasting. In *ARPA HPCC 94 Symp.* Advanced Research Projects Agency Computing Systems Technology Office, March 1994.

[DEF+94] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. An architecture for wide-area multicast routing. In *Proceedings, 1994 SIGCOMM Conference*, pages 126–135, London, UK, August 31st - September 2nd 1994.

[Dou87] F. Douglis. Process migration in the Sprite operating system. *Report No UCB/CSD 87/343*, [2] 1987.

[Dou89] F. Douglis. Experience with process migration in Sprite. In *Workshop on Experiences with Distributed and Multiprocessor Systems*, pages 59–72, Fort Lauderdale, Florida, October 5-6 1989.

[EKO95] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symp. on Operating Systems Principles*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[Fon94] H. J. F. Fonseca. Support environments for the modularization, implementation and execution of communication protocols. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, June 1994. In Portuguese.

[GBB+95] D. Guedes, D. Bakken, N. Bhatti, M. Hiltunen, and R. D. Schlichting. A customized communication subsystem for FT-Linda. In *Proceedings of the 13th Brazilian Symposium on Computer Networks*, pages 319–338, Belo Horizonte, MG, Brazil, May 1995.

[GL92] R. A. Golding and D. D. E. Long. Quorum-oriented multicast protocols for data replication. In *Proceedings of the IEEE International Conference on Data Engineering*, page 490, Tempe, AZ, February 1992.

[GL93] R. A. Golding and D. D. E. Long. Using an object-oriented framework to construct wide-area group communication mechanisms. Technical Report UCSC-CLR-93-11, University of California, Santa Cruz, March 1993.

[GMK88] H. Garcia-Molina and B. Kogan. An implementation of reliable broadcast using an unreliable broadcast facility. In *Proceedings of the Seventh Symp. on Reliable Distributed Systems*, pages 101–111, Columbus, OH, October 1988.

[GMS89] H. Garcia-Molina and A. Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th IEEE Symp. on Distributed Computing Systems*, pages 354–361, Newport Beach, CA, June 1989.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Trans. Comput. Syst.*, 9(3):242–271, August 1991.

[GNI92] M. Goldberg, G. Neufeld, and M. Ito. The parallel protocol framework. Technical Report 92-16, Dept. of Computer Science, University of British Columbia, Vancouver, British Columbia, August 1992.

[Gol92] R. A. Golding. *Weak-Consistency Group Communication and Membership.* PhD thesis, Dept of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, 1992.

[Hed88] C. Hedrick. Routing information protocol; RFC 1058. *Internet Request for Comments*, June 1988.

[HH93] L. B. Huston and P. Honeyman. Disconnected operation for AFS. In *Proceedings of the 1st USENIX Mobile and Location-Independent Computing Symp.*, pages 1–10, Cambridge, MA, August 2-3 1993. USENIX.

[Hil96] M. A. Hiltunen. *Configurable Distributed Fault-Tolerant Services*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, July 1996.

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

[HPM93] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A flexible base for distributed programming. In *Proceedings of the 14th ACM Symp. on Operating System Principles*, pages 69–79, Asheville, NC, December 1993.

[HS95a] M. Hiltunen and R. D. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th IEEE Symp. on Distributed Computing Systems*, Vancouver, BC, May 1995.

[HS95b] M. Hiltunen and R. D. Schlichting. Properties of membership services. In *Proceedings of the Second IEEE Symp. on Autonomous Decentralized Systems*, pages 200–207, Phoenix, AZ, April 1995.

[Hug88] L. Hughes. LAN gateway designs for multicast communication. In *Proceedings of the 13th Conference on Local Computer Networks*, pages 82–91, Minneapolis, Minn., October 10-12 1988. IEEE Computer Society.

[IB93] T. Imielinksi and B. R. Badrinath. Data management for mobile computing. *SIGMOD Record*, 22(1):34, 1993.

[IBar] T. Imielinski and B. R. Badrinath. Mobile wireless computing challenges in data management. *Communications of the ACM*, To appear.

[IDJ91] J. Ioannidis, D. Duchamp, and G. Q. Maguire Jr. IP-based protocols for mobile internetworking. In *Proceedings of ACM SIGCOMM 1991*, September 1991.

[IJ93] J. Ioannidis and G. Q. Maguire Jr. The design and implementation of a mobile internetworking architecture. In *Proceedings of 1993 Winter USENIX*, pages 489–500, January 1993.

[Kis90] J. J. Kistler. Transparent disconnected operation for fault-tolerance. In *IEEE-CS/TC-OS Workshop on the Management of Replicated Data*, Houston, TX, November 1990.

[KMS+93] K. Keeton, B. A. Mah, S. Seshan, R. H. Katz, and D. Ferrari. Providing connection-oriented network services to mobile hosts. In *Proceedings of the 1st USENIX Mobile and Location-Independent Computing Symp.*, pages 83–102, Cambridge, MA, August 2-3 1993. USENIX.

[KS91] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda File System. In *Proceedings Thirteenth ACM Symp. on Operating System Principles*, page 213, Pacific Grove, CA, October 1991.

[KTHB89] M. F. Kaashoek, A. Tanenbaum, S. F. Hummel, and H. Bal. An efficient reliable broadcast protocol. *ACM Op. Syst. Review*, 23(4):5–19, October 1989.

[LAKS93] B. Lindgren, M. Ammar, B. Krupczak, and K. Schwan. Parallel and configurable protocols: Experiences with a prototype and an architectural framework. In *Proceedings of International Conference on Network Protocols*, March 1993.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam81] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

[Lap92] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, Vienna, 1992.

[LBSR95] M. T. Le, F. Burghardt, S. Seshan, and J. Rabaey. InfoNet: The networking infrastructure of InfoPad. In *Proceedings of COMPCON*, San Francisco, California, March 1995.

[LG85] K. J. Lin and J. D. Gannon. Atomic remote procedure call. *IEEE Trans. on Software Engineering*, SE-11(10):1126–1135, October 1985.

[LSBR94] M. T. Le, S. Seshan, F. Burghardt, and J. Rabaey. Software architecture of the InfoPad system. In *Proceedings of the Mobidata Workshop on Mobile and Wirelsss Information Systems*, Rutgers, New Jersey, November 1994.

[MBM95] S. Maffeis, W. Bischofberger, and K. Mätzel. A generic multicast transport service to support disconnected operation. In *Proceedings of the 2nd USENIX Mobile and Location-Independent Computing Symp.*, Ann Arbor Michigan (USA), April 1995.

[Mil83] D. L. Mills. DCN local-network protocols; RFC 891. *Internet Request for Comments*, pages 1–26, December 1983.

[MMO+94] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, T. A. Proebsting, and J. H. Hartman. Scout: A communications-oriented operating system. Technical Report 94-20, Dept. of Comp. Sci., Univ. of Arizona, June 1994.

[MMSA+95] L. Moser, P. Melliar-Smith, D. Agrawak, R. Budhia, C. Lingley-Papadopoulos, and T. Archambault. The Totem system. In *Proceedings of the 25th IEEE Symp. on Fault-Tolerant Computing*, pages 61–66, Pasadena, CA, June 1995.

[MP96] D. Mosberger and L. L. Peterson. Making paths explicit in the Scout operating system. Technical Report 96-05, Department of Computer Science, University of Arizona, Tucson, AZ, May 1996.

[MPS89] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing replicated objects using Psync. In *Proceedings of the Eighth Symp. on Reliable Distributed Systems*, pages 42–52, Seattle, Washington, October 1989.

[MPS92] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Vienna, 1992.

[MPS93a] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(3):87–103, December 1993.

[MPS93b] S. Mishra, L. L. Peterson, and R. D. Schlichting. Experience with modularity in Consul. *Software–Practice & Experience*, 23(10):1059–1075, October 1993.

[MPS93c] S. Mishra, L. L. Peterson, and R. D. Schlichting. Modularity in the design and implementation of Consul. In *Proceedings of the First IEEE Symp. on Autonomous Decentralized Systems*, pages 376–382, Kawasaki, Japan, March 1993.

[MS92] S. Mishra and R. D. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report 92-19, Dept of Computer Science, University of Arizona, Tucson, AZ, 1992.

[MS93] B. Mukherjee and K. Schwan. Experimentation with a reconfigurable micro-kernel. In *Microkernels and Other Kernel Archtitectures Symp. II*, pages 45–60. USENIX, September 1993.

[MSK+93] B. A. Mah, S. Seshan, K. Keeton, R. H. Katz, and D. Ferrari. Providing network video service to mobile clients. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*. IEEE, October 1993.

[NCN88] S. Navaratnam, S. Chanson, and G. Neufeld. Reliable group communication in distributed systems. In *Proceedings of the 8th IEEE Symp. on Distributed Computing Systems*, pages 439–446, San Jose, California, June 1988.

[Nel81] B.J. Nelson. *Remote Procedure Call.* PhD thesis, Dept of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.

[NK93] M. Nelson and Y. Khalidi. Generic support for caching and disconnected operation. In *4th Workshop on Workstation Operating Systems (WWOS-IV)*, pages 61–65, Napa, CA, 1993.

[NYKT94] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First Symp. on Operating Systems Design and Implementation*, November 1994.

[OCD+88] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23, February 1988.

[OOW91] M. Olsen, E. Oskiewicz, and J. Warne. A model for interface groups. In *Proceedings of the 10th IEEE Symp. on Reliable Distributed Systems*, pages 98–107, Pisa, Italy, September 1991.

[OP92] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Trans. Comput. Syst.*, 10(2):110–143, May 1992.

[PB94] C. Perkins and O. Bhagwat. A mobile networking system based on Internet Protocol. *IEEE Personal Communications*, 1994.

[PBS89] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst.*, 7(3):217–246, August 1989.

[Per96] C. Perkins. IP mobility support; rfc 2002. *Internet Request for Comments*, October 1996.

[PKV96] D. K. Pradhan, P. Krishna, and N. H. Vaidya. Recoverable distributed mobile environments: Design and tradeoff issues. In *Proceedings of the 26nd IEEE Symp. on Fault-Tolerant Computing*, June 1996.

[Pos80] J. Postel. User Datagram Protocol; RFC 768. *Internet Request for Comments*, pages 1–3, August 1980.

[Pos81a] J. Postel. Internet Protocol; RFC 791. *Internet Request for Comments*, pages 1–45, September 1981.

[Pos81b] J. Postel. Transmission Control Protocol; RFC 793. *Internet Request for Comments*, pages 1–85, September 1981.

[PS88] F. Panzieri and S. K. Shrivastava. Rajdoot: A remote procedure call mechanism supporting orphan detection and killing. *IEEE Trans. on Software Engineering*, SE-14(1):30–37, January 1988.

[RBM96] R. van Renesse, K. Birman, and S Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, 39(4):76–83, Apr 1996.

[Rek93] Y. Rekhter. An architecture for transport layer transparent support for mobility. *Journal of Internetworking: Research and Experience*, 4, 1993.

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

[RS92] K. Ravindran and M. Sankhla. Multicast models and routing algorithms for high speed multi-service networks. In *Proceedings of the 12th IEEE Symp. on Distributed Computing Systems*, page 194, Yokohama, Japan, June 9-12 1992.

[SB90] M. Schroeder and M. Burrows. Performance of Firefly RPC. *ACM Trans. Comput. Syst.*, 6(1):1–17, February 1990.

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

[Sch90] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SKM+93] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experience with disconnected operation in a mobile environment. In *Proceedings of the 1st USENIX Mobile and Location-Independent Computing Symp.*, pages 11–28, Cambridge, MA, August 2-3 1993. USENIX.

[Spa91] A. Spauster. Ordered and reliable multicast communication. Technical Report CS-TR-312-91, Princeton UNIV, DEPT of CS, 1991. Thesis (Ph.D.).

[SS90] M. Satyanarayanan and E. H. Siegel. Parallel communication in a large distributed environment. *IEEE Trans. on Computers*, March 1990.

[SS94] D. Schmidt and T. Suda. The service configurator framework: An extensible architecture for dynamically configuring concurrent, multi-service network deamons. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, pages 190–201, Pittsburgh, PE, 1994.

[STW93] B. Schilit, M. Theimer, and B. Welch. Customizing mobile applications. In *Proceedings of the 1st USENIX Mobile and Location-Independent Computing Symp.*, pages 129–138, August 1993.

[TYT91] F. Teraoka, Y. Yokote, and M. Tokoro. A network archtitecture providing host migration transparency. In *Proceedings of AXM SIGCOMM 91*, pages 209–220, 1991.

[VKP93] N. H. Vaidya, P. Krishna, and D. K. Pradhan. Recovery in distributed mobile environments. In *Proceedings of IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 83–88. IEEE, October 1993.

[VM90] P. Verissimo and J. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symp. on Reliable Distributed Systems*, pages 54–63, Huntsville, AL, October 1990.

[VRB89] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *Proceedings of SIGCOMM'89*, pages 83–93, Austin, TX, September 1989.

[vRBF+95] R. van Renesse, K. Birman, R. Friedman, M. Hayden, and D. Karr. A framework for protocol composition in Horus. In *Proceedings of the ACM Symp. on Principles of Distributed Computing*, pages 89–102, Vancouver, Canada, August 1995.

[vRBG+95] R. van Renesse, K. Birman, B. Galde, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report 95-1500, Cornell University, Dept. of Computer Science, March 1995.

[vRHB94] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and performance of Horus: A lightweight group communications system. Technical Report 94-1442, Cornell University, Dept. of Computer Science, August 1994.

[Wal80] D. W. Wall. *Mechanisms for Broadcast and Selective Broadcast.* PhD thesis, Department of Computer Science, Stanford University, Palo Alto, CA, 1980.

[WPD88] D. Waitzman, C. Partridge, and S. Deering. Distance vector multicast routing protocol; RFC 1075. *Internet Request for Comments*, November 1988.

[WYOT93] H. Wada, T. Yozawa, T. Ohnishi, and Y. Tanaka. Mobile computing environment based on packet forwarding. In *Proceedings of USENIX Winter '93 Conference*, pages 503–517, January 1993.

[WZZ93] X. Wang, H. Zhao, and J. Zhu. GRPC: A communication cooperation mechanism in distributed systems. *ACM Op. Syst. Review*, 27(3):75–86, July 1993.

[YJT88] K. Yap, P. Jalote, and S. Tripathi. Fault tolerant remote procedure call. In *Proceedings of the 8th IEEE Symp. on Distributed Computing Systems*, pages 48–54, June 1988.