Communication Abstractions for Distributed Systems

by

James Won-Ki Hong

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada 1991

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

This thesis presents techniques for reducing the complexity of communication in distributed systems. It presents a set of simple standards and tools that provide an efficient and consistent programming interface that can be used to implement a great variety of communication interactions both within and between various types of paradigms, abstractions, and entities. It also provides a framework based on these standards and tools, with which arbitrary communication systems can be constructed.

Various communication paradigms and issues are examined in order to identify and categorize the fundamental aspects of communication. These fundamental aspects are then redefined and organized into a set of communication abstractions which is simple and general, rigorous and flexible, low-level and extensible. The generic communication model based on this set provides a framework for arbitrary communication systems. Further, the model utilizes the efficiencies of a single memory domain while providing a universal interface between various types of paradigms, abstractions, and entities across a wide spectrum of environments.

The framework provided in the generic communication model is used to develop a specific model called the Buffer and Queue Model which provides a single efficient and consistent communications programming interface. The Buffer-Queue protocol extends this consistent programming interface transparently to a distributed environment. Finally, a few examples are presented using Buffers and Queues that illustrate how various complex communication facilities may easily be developed and how the use of a common fundamental base makes intermixing of locally defined user interfaces and conversion between higher-level protocols a relatively straightforward task.

Acknowledgements

I would like to thank many people who have given their time, support, advice and encouragement in helping me to complete this thesis. First and foremost among these people is my supervisor, Prof. Jay P. Black, who has put up with me and guided me from the very beginning. He has taught me how to research, how to present ideas clearly, and how to do academic writing in good English. Prof. David Taylor, who acted as my adviser for a year while Jay was on sabbatical, made sure that my research and thesis writing were on track and provided useful help on numerous occasions.

The other members of my thesis examination committee, Prof. R.D. Schlichting, Prof. P.A. Buhr, Prof. D.D. Cowan, and Prof. W.M. Loucks gave many useful comments and suggestions to improve my thesis. I owe a great debt to a former Shoshin research associate, Ross Wetmore, who has spent endless hours listening to many nonsensical ideas and suggesting how to turn them into useful ones.

I would like to thank my parents and grandparents who instilled in me the desire to strive for excellence in all aspects of my life. I would also like to thank my parents-in-law, who have sacrificed a lot during the last two years to make sure that I have enough time to work on my research.

Special thanks to my wife, In-Young, and my son, David. In-Young has inspired me from the beginning and kept convincing me that I could do it. She has provided me with constant encouragement, understanding and love. Although David, who was born at the early stage of my research, demanded a lot of hours and care, he has given me joy, inspiration and desire to work hard.

Finally, I am grateful to Natural Science and Engineering Research Council of Canada and Prof. Jay Black for financial support.

I dedicate this thesis to

In-Young

and

David

Contents

List of Figures

Chapter 1

Introduction

As computers have grown more powerful, their capabilities have become more complex and system software, defined as the standard package of system functions and programming tools demanded by the end user, has mushroomed in response. Such growth has been driven both by the increasing variety and sophistication of the hardware interface (either through directly connected hardware devices, or in the extended scope achieved through networking), and by the devolution of application-level software packages into standard interfaces for yet higher layers of application software [Sche86, Lint87, Flow90]. One consequence of this growth is that system software, like hardware, can no longer be treated as a single monolithic entity, but rather has become a complex network of interacting modules. Further, the diversity of the types of entities (*e.g.*, procedures, processes, devices) involved in such software has produced vastly different communication semantics and interfaces. In such a complex and diverse environment, there is a desperate need to identify and extract common concepts and build a set of simple, efficient and widely applicable communication abstractions which can be extended transparently across a distributed system. The analysis and design of such a set form the major contribution of this

thesis.

The modularization of such software was initially typified by the concept of layered structure as embodied in the ISO-OSI Reference Model [Zimm80] for network protocols. Each layer abstracted a particular subset of network operations and capabilities and possessed a loosely-defined interface by which it could be invoked by adjacent layers. As the move to hardware independence and open systems became stronger and went beyond network protocols, complete procedural definitions of operating system layers and standard interfaces have emerged, such as the System V Interface Definition (SVID) [ATT85] or similar POSIX standards [IEEE88].

More recently, a need for dynamic flexibility in the type and arrangement of layers has suggested the concepts of a much more self-contained and independent module which performs a well-defined set of functional tasks and which possesses a universal standard interface, and of a mechanism for assembling and disassembling many related modules into a complex hierarchy. AT&T STREAMS is an example of such a design [ATT87]. The growing move to the object-oriented design philosophy [Booc86, Cox86, Meye88] takes this idea one step further to that of self-contained code and data entities which embody a particular object or concept and provide a complete specification of the operations which can be performed by or on the entity. Furthermore, the object-oriented philosophy provides a framework for categorizing and developing various types or classes using the concepts of class hierarchy, inheritance and redefinition.

However, as system components become more modular and independent, such that even the hardware and software base of the system could be changed on a daily basis, the need for well-defined standards of communication between various entities and types of entities becomes an increasing problem. Moreover, as tasks involve greater numbers of independent modules and data flow among modules be-

comes a more significant component of such tasks, efficiency coupled with versatility becomes ever more important in such communication.

In order to satisfy a spectrum of both present and future needs, a set of communication concepts and standards is required which is simple and general, rigorous and flexible, low-level (for use within a single node) and extensible (for applicability to a distributed environment). This set of communication concepts and standards should be accompanied by a framework that can support existing communication paradigms and protocols and can be used to develop new paradigms or protocols.

This thesis develops such a set by reducing the communication problem to its critical components and building a framework upon which arbitrary communication systems can be dynamically organized and constructed. First, various communication paradigms and issues are examined as a means to identify and categorize some of the fundamental aspects of communication. An attempt is made to redefine and organize these fundamental aspects into three communication abstractions, namely *data*, *node* and *delivery/synchronization*. Using the object-oriented design approach, we organize these abstractions hierarchically and use them as the basis for a generalized communication paradigm. The resulting generic communication model consists of a set of communication concepts and standards which meets the requirements described above.

The Buffer and Queue Model is presented as a specific instantiation of the generic communication paradigm. It seeks to utilize the conceptual efficiencies of a single memory domain while providing a universal interface between various types of entities across a wide spectrum of environments. Finally, several examples are presented that illustrate how various complex communication facilities may easily be developed using these elementary tools and how the use of a common fundamental base makes intermixing of locally defined user interfaces or conversion

between higher-level protocols a relatively straightforward task.

1.1 Thesis Overview

The main theme of this thesis concerns the use of abstractions for reducing the complexity of communication in distributed systems. The major research goal is to develop a generic communication model that consists of a set of simple standards and tools, that provides a universal communication programming interface (for both the system and application levels), and that can be used to construct specific communication models or paradigms. Another major goal is to develop a specific communication model that satisfies the constraints of the generic model, and to demonstrate its simplicity and versatility.

Numerous researchers have introduced various high-level abstractions in attempts to provide a single consistent user interface that encompasses different underlying transport techniques. Message-passing (MP), streams and remote procedure call (RPC) are some examples of these attempts. However, these abstractions have been designed to support communication between the same or similar types of entities. At a low level, there are network communication protocols and hardware devices with well-defined interfaces. These different types of entities, communication abstractions and implementation mechanisms are characterized by vastly different communication semantics and interfaces. This makes communication between different types of entities, abstractions and mechanisms a very difficult (if not impossible) task. Currently, most of these interactions are implemented by communication programmers using their own *ad hoc* approaches yielding the diverse and unwieldy programming environment shown in Figure 1.1. In such an environment, there is a desperate need for simpler universal standards, and interfaces and tools,

which can be used by all types of partners for all types of communication.

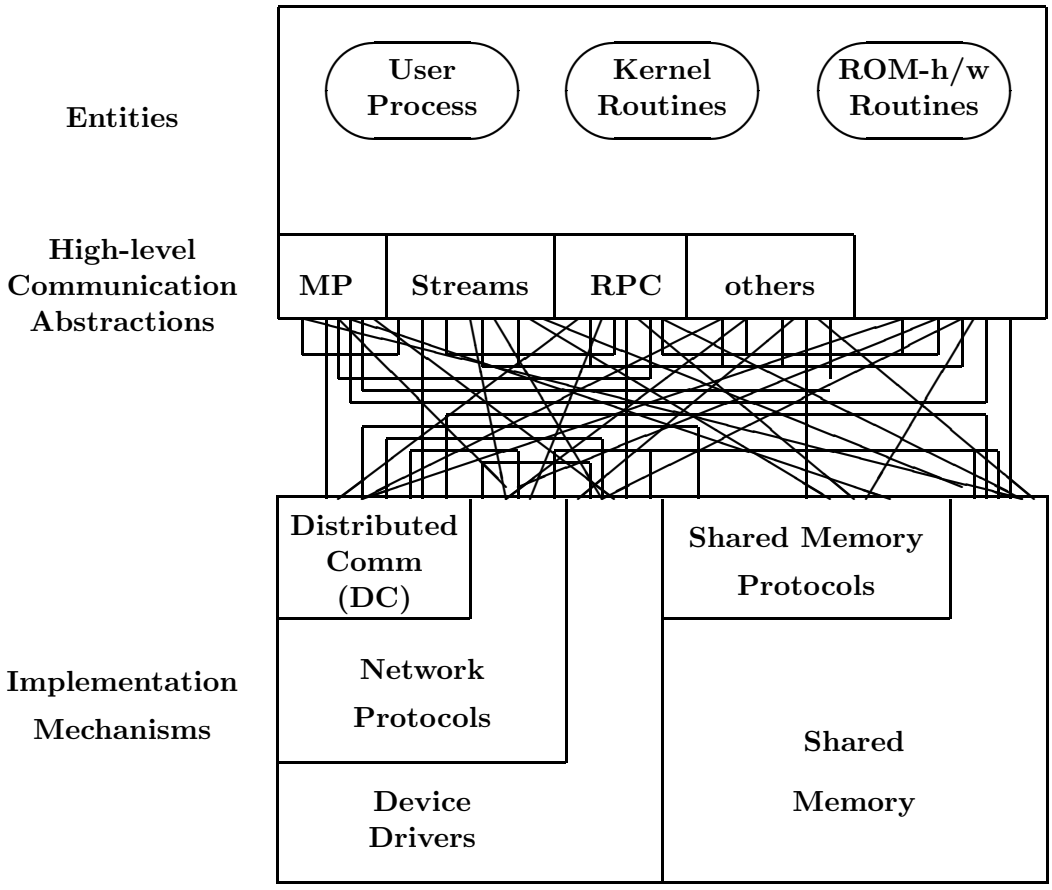

Figure 1.1: A Conventional Communications Programming Environment

In particular, there is a need for a universal interface lying just below the high-level abstractions (called the *internal communication interface*), which can utilize the conceptual framework and efficiencies of a local memory environment. The internal communication interface should be accompanied by a uniform data object structure that all types or levels of entities can deal with, thus yielding a truly universal or generic communication programming paradigm.

Figure 1.1 emphasizes the need for a better approach to communication in modern computer systems. The universal communication programming paradigm proposed in this thesis will provide simple answers to the following three related questions:

- How can a programmer write code to locate some service, and then communicate with the provider of that service?
- How can this code be made independent of the memory domain, execution control regime (procedures, threads, processes, device drivers *etc.*), and physical location of the service?
- Can this independence be achieved even when the communicating entities have different characteristics?

Chapter 2 surveys various existing communication paradigms and issues in distributed systems as a means to identify and categorize some of the fundamental aspects of communication. In particular, aspects of intra-task communication (communicating within the bounds of a protection domain), inter-task communication (both within and across protection boundaries), and network communication (general device and kernel-to-kernel communications) are examined for similarities and differences. Various types of entities involved in these communication paradigms have been examined, for example: conventional procedures, coroutines [Krak88], kernel routines and ROM hardware routines in intraprocess communication; threads (*i.e.*, light-weight processes) [Acce86], processes (*e.g.*, UNIX processes) and tasks in interprocess communication; and combinations of these in network communication.

In the first half of Chapter 3, the fundamental aspects identified in Chapter 2 are examined more fully in an attempt to elucidate the basic requirements for such

a set of simple, universal concepts and tools that can then be used in any type of communication among all types of partners. Based on these fundamental concepts and requirements, the object-oriented design framework is applied to develop a generic communication model in the second half of Chapter 3. This generic communication model can then be used to implement specific communication paradigms efficiently or to develop future communication paradigms. Furthermore, the generic communication model is designed to replace *ad hoc* interfaces used to implement most current communication systems (shown in Figure 1.1) with a single *internal communications interface*, which consists of a universal functional interface and a uniform data-object structure.

Chapter 4 develops a specific instantiation called the Buffer and Queue Model, which satisfies the constraints of the generic communication model developed in Chapter 3. The Buffer and Queue Model provides a simple, low-level but powerful and efficient communication paradigm, which utilizes the conceptual efficiencies of the shared-memory programmer's environment while providing a universal interface between various types of entities across a wide spectrum of environments. In order to extend the internal communications interface transparently to a distributed environment, the Buffer-Queue protocol (BQP) is developed to provide remote operations and data transfer.

Chapter 5 presents implementation examples of various communication paradigms using Buffers and Queues. In particular, examples of internal communication (including procedure call and coroutines), local interprocess communication, network communication, distributed communication and bulk data transfer are developed to illustrate the ease of construction. Further, examples implementing distributed and conventional semaphores, one-way communication (*e.g.* multicast) and distributed shared memory are also presented to show the versatility of the

methods.

Chapter 6 summarizes the thesis and makes comparisons of the work with related work. Also, several possible areas for future research that stem from this work are suggested. Finally, the major contributions of this thesis are discussed.

1.2 Related Work

Over the years, many researchers have worked on various aspects of communication, some in developing new communication paradigms [Gent81, Birr84, Ritc84, Li86, Ahuj88] or protocols [Abra73, Metc76, Post80, Post81, IEEE83, ISO84, Zwae85, Ross89, Sand90], others in attempts at improvements or performance optimizations [Cher86b, Vasu87, Cart89, OMal90]. Unfortunately, most have concentrated on a specific or limited set of protocols and area of application, resulting in a large number of highly specialized and mutually incompatible proposals.

Recently, there have been some attempts to provide a single consistent programming interface that bridges many underlying techniques, which would reduce the programmers' concern for dealing with the conceptual partitioning of interfaces. Berkeley sockets [Leff88], AT&T STREAMS [ATT87], TACT[Auer90], Choices Conduits [Zwei90] and the x-Kernel [Hutc88] are examples of such attempts. Unfortunately, most of these efforts fail to support new forms or as many forms of communication as one might wish. In my estimation, the main reason behind this failure is that most have *a priori* targeted a specific or limited set of areas and thus do not possess sufficiently general communication abstractions to start with. Below, each attempt is examined and its shortfalls are pointed out.

1.2.1 Berkeley Sockets

Sockets are a generalization of the UNIX file access system designed to incorporate network protocols [Koch89]. Sockets offer several types of data transfer service: *stream*, *datagram*, *raw* and *sequenced-packet*. A stream socket provides sequenced, reliable, bidirectional connection-based byte streams. A datagram socket supports connectionless, unreliable messages of a fixed maximum length. A raw socket provides access to internal network interfaces and is available only to the super-user. A sequenced-packet socket is similar to a datagram socket except that packets are guaranteed to be received in the order they are sent. They are also guaranteed to be error-free.

Sockets provide a common set of data transfer operations for these different types of service. A user specifies the type through a parameter field (or parameter fields) to an operation call. For convenience and efficiency, sockets allow transfer of data, which may be composed of memory fragments from non-contiguous memory locations. However, this simple *scatter-gather* list is not effective in supporting multiple protocol layers.

The Berkeley UNIX kernel implements communication using three conceptual layers: the socket layer, the protocol layer, and the device layer. Thus, there exist at least three interfaces: the user/socket interface, the socket/protocol interface, and the protocol/device interface. The socket layer provides the interface between user system calls and the lower layers, the protocol layer contains the protocol modules used for communication (*e.g.*, TCP, IP), and the device layer contains the device drivers that interact with the network devices. Legal combinations of protocols and drivers are specified at system configuration time.

Although sockets are used quite widely in a variety of applications involving

interprocess communication in Berkeley UNIX systems, they fall short of providing a single consistent programming interface for various communicating entities and paradigms. One of the major shortfalls of the socket abstraction is that it only supports one form of communication paradigm, namely the client-server model. Thus, it fails to support, for example, the shared-memory paradigm (*i.e.*, data passed by reference). Although it may be possible to simulate other communication paradigms such as shared memory, the socket abstraction does not provide the necessary fundamental tools for the programmer to work with. Another major shortfall of the socket abstraction is that it only supports one of three interfaces present in the Berkeley UNIX kernel, namely the user/socket interface [Pete90]. For example, sockets can be used to implement streams or sequenced packets in terms of datagrams, but the kernel implementation does not use the socket programming interface for Ethernet layer datagrams. Similarly, the socket abstraction cannot be used by all entities such as application-level protocols, network protocols, or device drivers.

1.2.2 AT&T STREAMS

AT&T STREAMS[1] [ATT87] is a general facility and a set of tools for the development of system communication services. It supports the implementation of services ranging from networking protocol suites to individualized device drivers. STREAMS defines standard interfaces for character input/output within the kernel, and between the kernel and the user. A Stream is an abstraction of a full-duplex data transfer path between a STREAMS driver in kernel space and a process in user space. In the kernel, a Stream is constructed by linking a Stream head, a driver and

[1] AT&T STREAMS is implemented in the System V UNIX system and thus the terms AT&T STREAMS and System V UNIX STREAMS are often used interchangeably.

zero or more kernel-resident modules between the Stream head and driver. These modules are used to perform intermediate processing of data as it passes between the Stream head and driver.

STREAMS modules are dynamically interconnected in a Stream by a user process. Each module consists of a pair of QUEUEs called message queues. One message queue is used to handle messages traveling downward (called the write queue) and the other is used to handle messages traveling upward (called the read queue). Messages are the means of communication within a Stream. All messages are composed of one or more message blocks. A message block is a linked triplet, defined recursively to consist of a message block, a data block and a variable length buffer block [ATT87]. The STREAMS mechanism also supports the ability to multiplex Streams in several configurations: many-to-one, one-to-many and many-to-many.

Because modules in STREAMS can be inserted to form a chain of modules at run-time, STREAMS is more dynamic and flexible than Berkeley sockets (recall that legal combinations of protocols and drivers are specified at system configuration time in Berkeley sockets). Although dynamic insertion of modules by the user may be authorized by a privileged entity such as the kernel, STREAMS provides a standard set of tools and a uniform interface, with which entities of various levels can work. STREAMS uses the concept of a queue as a repository into which messages may be stored and from which they may be retrieved. The message structure is flexible and versatile in that it is open-ended, and can support a tree structure of multiple message fragments. Thus, STREAMS is more generic than the Berkeley socket abstraction in that it can support all of application-level, network protocol, and device driver interfaces.

However, the design approach taken for STREAMS is a classical example of starting with abstractions for specific goals and having difficulties in enhancing or

extending the abstractions to support other forms of communication. STREAMS was originally designed to support character I/O within the kernel and later added the capabilities of messages and multiplexing for network protocol processing. A major shortfall of the STREAMS implementation is that its concepts do not extend well to remote nodes (*i.e.*, it does not provide a uniform interface for a distributed system). That is mainly due to lack of an identification scheme and a protocol which can be used to extend STREAMS concepts transparently to a distributed environment.

1.2.3 TACT

TACT (Transport Abstraction Conversion Toolkit) is a toolkit (software function library) for synthesizing conversions between different transport interfaces [Auer90]. TACT classifies transport interfaces according to their abstraction type or basic model of communication. Each actual transport interface is converted to or from a canonical form for its abstraction type, and TACT provides abstraction converters to convert between all of the canonical forms. A TACT conversion is accomplished by combining some number of elementary conversions which execute in series. In his work, Auerbach introduces the *transport abstraction*, which basically consists of the peer semantics between communicating endpoints and from which the details of the peer protocol, interface syntax and local semantics have been removed. He defines a set of *abstraction types* (P1, P2, *etc.* in Figure 1.2), which capture most of the crucial features of transport abstractions offered by all interesting transport interfaces (M1, M2, *etc.*) and actual transports (N1, N2, *etc.*). Each of the abstraction types offers a "universal interface" but only for that particular type. Software modules called *abstraction converters* are used to map every distinct abstraction type onto every other. TACT provides a set of *program interface packages*, which

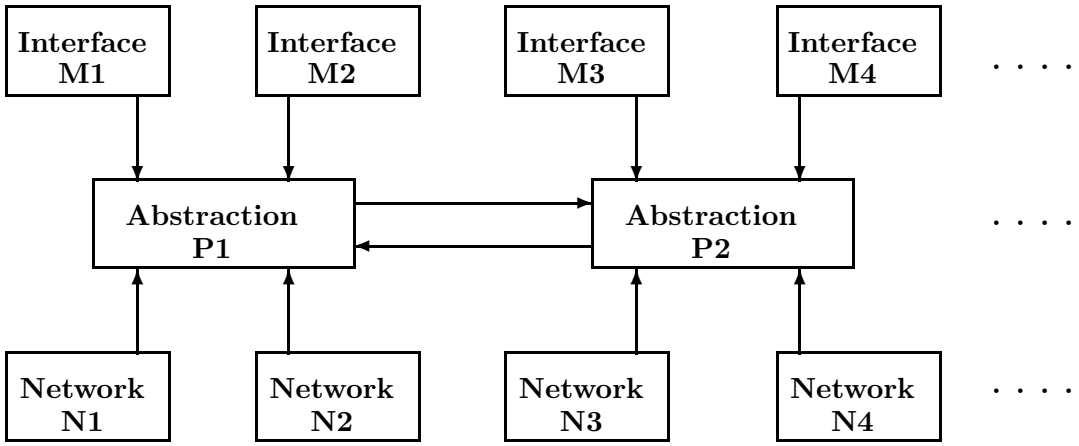

Figure 1.2: TACT Conversion via a Set of Transport Abstractions

maps each of the actual interfaces onto one suitably chosen universal interface from the set of such interfaces available. TACT also provides a set of *transport implementation packages*, which maps one suitably chosen universal interface from the set of such interfaces available onto one of the actual implemented transports such as TCP/IP, NETBIOS [IBM86], SNA [Cyps78] and OSI [ISO84].

TACT achieves its results by minimizing the number of "truly different" cases through the *transport abstraction* concept. TACT supports five abstraction types: 1) the *stream* and 2) *datagram* abstractions, which are like their counterpart Berkeley socket types [Leff88], 3) the *sequenced packet* abstraction, which is based on the OSI transport, and 4) the *byte conversation* and 5) the *record conversation*, which are based on APPC semantics [Bara85] but with the full duplex extension. The byte and record conversation abstractions support "short orderly exchanges", with the former being byte-oriented and the latter being record-oriented.

The goal of TACT is the same as one of the goals of this thesis, namely providing a universal interface between different transport interfaces. The approach used to

solve the problem is also very similar to ours. That is, both have reduced the complexity of the problem by using the *abstraction* concept. TACT, however, falls short of providing a "truly" universal interface since it provides a universal interface only at the transport layer.

1.2.4 Conduits

The Conduit is an abstraction for bidirectional communication that is being used to implement a family of network protocols as part of the Choices operating system [Camp87]. The Conduit abstraction is used as a communication channel through which messages are passed. There are different types of Conduits, each representing a network communication protocol. For example, TCPConduit and ISO-Class 4 Transport Conduit are two of the virtual-circuit Conduits, and IPConduit and EthernetConduit are two of the datagram Conduits. They have applied the object-oriented design approach to the Conduit abstraction, and these various Conduit classes are organized hierarchically. Conduits offer operations to connect to other Conduits to form a chain of Conduits, which represents a data path through the system, and to disconnect from each other when the path is no longer required. The setting up and tearing down of Conduit chains can be done either at system configuration time or at runtime. They also provide operations to insert messages from the top of a Conduit for sending messages downward (*i.e.*, user to device driver) and to insert message from the bottom of a Conduit for sending messages upward (*i.e.*, device driver to user).

There is a separate abstraction of a message called a ConduitMessage. Various ConduitMessage classes, each representing a communication protocol packet, are also organized hierarchically. They offer operations for packetizing and depacketiz-

ing. These operations are used to add and remove control information (*i.e.*, protocol headers and trailers) conveniently. The internal structure of a ConduitMessage allows both a simple message (*i.e.*, a single block message) and a complex message (*i.e.*, a list of message fragments).

Conduits is basically an object-oriented version of AT&T STREAMS. They have employed class definitions, redefinitions and inheritance for simpler and more systematic development of various complex message and communication pipe structures required for user to kernel (protocol and device driver) communication. Their message structure is more flexible and allows more efficient insertion or removal of control information.

One of the major shortfalls of Conduits is that they are intended mainly for network communication (user-to-kernel, kernel-to-kernel). They do not provide a general basis for supporting other forms of communication such as user-to-user interprocess communication (both local and remote). Another but related problem is that their concepts do not extend to remote nodes and thus supporting transparent distributed communication is not presently obvious. Nor do they have a separate abstraction and hierarchy for either the delivery or synchronization. The delivery is provided by a single method, which is by procedure call (or nested procedure calls). They only support one form of user execution synchronization, namely synchronous, where the user is blocked until the invoked operation completes.

1.2.5 *x*-Kernel

The *x*-Kernel, being developed at the University of Arizona, is a configurable operating system kernel that supports multiple address spaces, light-weight processes, and an architecture for implementing and composing network protocols. The main

objectives of the *x*-Kernel are to facilitate the implementation of protocols, and to provide a framework for designing and evaluating new protocols [Hutc88].

The *x*-Kernel views a protocol as a specification of a communication medium through which a collection of participants exchange a set of messages. It provides three communication abstractions: *protocols*, *sessions* and *messages*. Protocol objects are used as an abstraction of network communication protocols (TCP, IP, *etc.*). The relationships to other protocols are defined and created at configuration time. Session objects are created dynamically as connections are made from one protocol object to another. Each session object contains the code to process the arriving messages (from either above or below) and data structures that represent the local state of some network connection. Messages are objects that move through the session and protocol objects. Messages arrive at the top and flow down, and at the bottom and flow up. The flow of messages through the *x*-Kernel (*i.e.*, between protocol and session objects) is implemented by procedure calls.

Protocol and session objects provide a uniform interface for each communication protocol. A series of protocol and session objects in the kernel provide a path for messages to travel up and down. This is very similar to a series of connected Conduits or a series of connected STREAMS modules. A protocol object supports operations to create session objects and to demultiplex messages received from its lower-level protocols. A session object supports operations for "pushing" messages from a higher-level to a lower-level protocol and for "popping" messages in the reverse direction. Both objects support control operations to access and manipulate the internal state of the protocol or session according to the specified operation code. Message objects support operations for manipulating the content of messages: operations for adding and deleting headers, fragmenting and reassembling messages, and saving and freeing messages.

I believe that the *x*-Kernel work is closest to what I hope to achieve in this research, which is a generic communication model that can be used to classify, develop and implement various forms of existing and future communication. They have abstracted fundamental concepts of communication and implemented a variety of network protocols using the "object-based" design approach.[2] Unfortunately, I do not believe their abstractions, namely the protocol, session and message, are a complete set or appropriate expressions of fundamentals to support all forms of communication in distributed systems. For instance, I believe that communication consists of three components (namely data, endpoints and delivery/synchronization) but they do not provide an abstraction for delivery or synchronization that is rich enough to support a variety of forms. Their session and protocol abstractions view communication as peer-to-peer transfer between endpoints with essentially identical characteristics. Furthermore, their abstractions do not extend easily to distributed communication (at least I have not seen their use in this paradigm), and thus their main area of support is local interprocess communication and network communication [Hutc88].

Research presented in this section gives examples of attempts to provide a single consistent programming interface that can be used in a variety of situations. In the next chapter, a survey of various existing communication paradigms is made as a means to identify and categorize some of the fundamental aspects of communication to provide a starting point for a more universal programming interface.

[2]The "object-oriented" design approach is differentiated from "object-based" design approach in that the former uses information hiding, data abstraction, class hierarchies and inheritance, whereas the latter uses information hiding and data abstraction only.

Chapter 2

Current Communication Paradigms

Communication is defined as the transfer of data between two or more entities. In this definition, three fundamental concepts have been used. The first is the concept of *data* or *message*, what is it and what forms might it take. The second is the concept of *nodes*, *entities* or *endpoints* of the communication. The third is the concept of *transfer* which involves the medium and mechanics of transporting the data while preserving its form and content. The first two, as will be shown, can generally be described by static structures, while the third represents the dynamic functionality.

In this chapter, several current communication paradigms are examined as a means to identify and categorize some of the fundamental aspects of communication. In particular, aspects of *intraprocess* communication (communicating within the bounds of a single protection domain[1], *interprocess* communication (both within

[1] *Protection domain* is defined as a set of accessible objects, with permitted access operations or rights for each.

and across protection domains), and *network* communication (general device and kernel-to-kernel communications) are examined for similarities and differences.

2.1 Intraprocess Communication

Intraprocess communication usually takes place within a single address space[2] and is typified by the traditional procedure call. Within a traditional sequential program, a single thread[3] of control serially accesses individual modules or routines. These modules communicate by passing data in registers or shared memory (which includes memory defined globally relative to the modules), or memory allocated dynamically on the stack. Data may be passed by value (*i.e.*, a copy of the actual data is transmitted), or by reference (*i.e.*, a pointer to the data is passed). In either case the form of the data is directly controlled by the user and is preserved during the process. Transmission is accomplished using an argument passing protocol, and transfer of control with implicit synchronization by the mechanics of the subroutine call. Upon return from the call, the operation is known to be complete, and any data structures may be accessed or modified by the caller without any fear that this will affect the communication process. Such a process is inherently serial or synchronous in nature. The thread of control in the calling routine is logically blocked until a return from the callee resumes execution in the caller (*i.e.*, synchronous user execution). Another inherent characteristic of procedure call is that it is a one-to-one operation between the caller and the callee.

In most systems, a variant of the procedure call is the most common form of communication between user and kernel. The transmission is accomplished with

[2]An *address space* defined to be a self-contained area of memory accessible to an executing program.

[3]A *thread* is defined to be a sequence of instruction executions.

an interrupt or its equivalent, rather than with a subroutine call, with possible changes to processor privilege level, stack environment or resource accessibility. While the protection domain may have changed, it is usually the case that the original user environment is a complete subset of the new environment, and thus nothing is significantly different from the traditional procedure call. However, an expanded protection domain may now overlap other previously inaccessible domains permitting data to be copied between previously isolated entities. The need to copy the data, that is to pass by value in such cases, is a property of a relay or pipeline between memory-independent entities. *Memory-independent entities* are defined as those that have separate protection domains. That is, an entity can not access the memory of the other without the help of a privileged entity such as a kernel.

A more complex example of the procedure call is found in the Clouds [Dasg87, Rama89] object invocation procedure. Clouds objects exist in individual address spaces in the global virtual address space that spans the entire network. When an object invocation is made, the kernel intervenes and remaps the segment(s) of the invoked object into the address space of the caller [Rama89]. In general, such an environment requires that data be passed by value, though *ad hoc* provisions can be made to map corresponding blocks of memory in both modules. In the case of large blocks of data, transmission by value can result in significant use of memory and CPU resources as it is relayed through a shared medium such as the stack. On the other hand, the memory-independent nature of the invocation makes it possible to complete the invocation on an arbitrary node within a distributed system. It is also possible, with the help of a name server and dynamic link table, to provide a completely dynamic runtime environment.

Communication in Remote Procedure Call (RPC) [Birr84] is similar to the Clouds example in that memory independence between caller and callee, trans-

mission of data by value, and the accompanying degree of node independence are distinguishing characteristics. RPC packages hide details of the underlying transport protocols and automatically provide packing and unpacking of data, locating remote procedures, bindings to remote procedures, and invocations of remote procedures. When an RPC[4] is made to a remote node, a new thread of control or agent is created which carries out the remote operations on behalf of the caller, and coordinates the communication with a local stub. Local and remote stubs connected through a low-level protocol is another example of a relay or pipeline, however one in which the ends of the pipeline are themselves independent entities communicating via a lower level protocol.

All of the above implicitly assume adherence of both caller and callee to a rigorous interface definition, both in the format of the argument list (data) including returned values, and in the mechanics of the transmission (calling protocol). The rigorous nature of this paradigm can be enforced by static type-checking at compile time, though this does not rule out all possibilities for runtime error. In addition, the nature of the interaction is one of client-server in which the server (callee) is always presumed ready to accept communication, and in which the client (caller) is always blocked for the duration of the communication process.

2.2 Interprocess Communication

Message-passing (MP) [Gent81, Andr83], *streams* [Leff88, Ritc84] and *shared memory* [Andr83, Cher86a, Bal88] are subcategories of *interprocess* communication (IPC), and as such are somewhat coarser-grained and more costly communication

[4]Although RPC is usually considered to be interprocess communication, its semantics are the same as procedure call, where a single, logical thread of execution is involved. Thus, RPC is categorized as part of intraprocess communication.

paradigms than the procedure call. In its simplest conceptual form, IPC consists of a pair of send-receive (or write-read) primitives executed by independent entities at each end of the communication pipeline.

Unlike procedure calls, where callees are always ready to accept communication, independent entities or processes[5] require privileged assistance (*e.g.*, from the kernel) in order to communicate (*i.e.*, in synchronizing the send and receive operations and to copy data), because in general there is no longer any guarantee that one party or the other will be ready to complete its side of the transaction, or in fact will ever do so. This characteristic of independent execution is referred to as *temporal independence* and techniques for overcoming it as *synchronization*. Thus, for independent entities or processes, there exists in the general case both memory independence and temporal independence. In addition, the dynamic nature of processes gives rise to a runtime identification or addressing problem in that a logical identification of the communication partner must be translated into a transient physical address at the time of the operation. This dynamic nature limits error detection in message-passing to runtime checks and can also introduce a significant efficiency bottleneck. Below, the characteristics of each IPC subcategory are examined.

2.2.1 Message-Passing

Several variants of message-passing exist in current systems [Andr83]. One of the primary differences among them is the degree of blocking of user execution (*i.e.*, when the communicating entities are unblocked to continue their executions). The extreme variants are *synchronous* and *asynchronous* message-passing. Two other

[5]A *process* is defined as a sequence of instruction executions (*i.e.*, a thread) plus an address space.

variants that will be discussed below are *highly synchronous* and *partially synchronous*, which are both types of synchronous message-passing.

In synchronous message-passing, either the sender, the receiver or both must be blocked prior to the start of actual data transfer. Only when data transfer is complete will the logical threads of control resume executing statements following the send and receive primitives, whereupon both sending and receiving parties can access any data structures associated with the communication without fear of interference. In general, some mechanism is required to signal the blocked entities when the active agent or relay has completed its task. This is typically achieved by the kernel marking each process as runnable and placing it in the ready queue. The blocking send and blocking receive primitives supported in the Shoshin distributed software testbed [Toku83] are an example of synchronous message-passing.

In asynchronous message-passing, the sender or receiver initiates communication processing, but the thread of control returns immediately (usually before completion of the operation). A separate synchronization step such as polling or blocking may be executed later to check for completion. In this case, the sending or receiving entities must ensure that the status of any data or control information is preserved between the time the operation is initiated and the time that data is actually transferred. Because a sender is allowed to continue execution before a receiver actually receives the message that has been sent, asynchronous message-passing can potentially allow a sender to get arbitrarily ahead of a receiver. Consequently, when a message is received, it contains information about the sender's state that is not necessarily current. Messages may also be dropped without notice if communication semantics permit (*e.g.*, datagram).

Because of the potential for inadvertent data corruption, most asynchronous message-passing implementations pass the data by value when the transfer is ini-

tiated. Asynchronous message-passing is less popular than synchronous message-passing in current systems because the buffer management for supporting asynchronous message-passing is simply too difficult. Asynchronous message-passing can be useful in applications where there is no need to wait for a result or acknowledgement (ACK). It can provide improved performance compared to synchronous message-passing by eliminating the wait for an unneeded result. It can also be useful where reliability is not important or where reliability is provided by the underlying communication system.

In highly synchronous message-passing, both the sender and the receiver must be blocked before the data transfer can actually take place as in synchronous message-passing. However, the sender is not unblocked when the sender's data is received by the receiver as in the case of synchronous message-passing. Instead, the sender is blocked until the user reply has arrived from the receiver. Since the reply carries user data, highly synchronous message-passing potentially guarantees user level reliability. The *send-receive-reply* message transaction supported in the V Kernel [Cher84] is an example of highly synchronous message-passing.

Partially synchronous message-passing[6] falls somewhere between synchronous message-passing and asynchronous message-passing. Others call this *unreliable blocking* message-passing [Tane85] because it does not guarantee the reliable message delivery to the receiver as synchronous message-passing does. Within the course of a message exchange between two entities, numerous independent events can easily be identified. Partially synchronous message-passing guarantees the delivery of data only to a particular event (or point), whereupon the blocked entity is unblocked. Thus, each point to which the data is delivered can be associated with

[6]This variant is called "partially synchronous message-passing" because the user only blocks partially compared to the degree of blocking that is generally accepted for synchronous message-passing.

a particular variant of partially synchronous message-passing, yielding numerous variants.

This variant of message-passing is more useful for applications where synchronization is not important or is achieved by some other explicit mechanism, but different degrees of reliability are useful. An electronic mail delivery application capable of sending back intermediate acknowledgements to the sender of how far a mail message has reached is a coarse-grained example of such an application. In a mail delivery system, the sender does not usually synchronize with the receiver. However, the sender may wish to discover how far the mail message has travelled, or may wish to have a guarantee of the delivery of his mail to at least a particular point. Partially synchronous message-passing can support this kind of application.

A finer-grained application of partially synchronous message-passing can be found in network communication. Among many different ways of identifying events, several protocol layers embedded in most communication systems involving network communication can be used to define event points. For example, the sender can be unblocked when it receives an ACK of the delivery of the message from the IPC layer in the receiver node. This ACK merely indicates that the remote entity in the IPC layer (*e.g.*, an IPC server) has received the message but does not say anything about whether it has been delivered to the intended receiver. Other possible variants include having the sender unblock when it receives an ACK from the remote transport layer, from the remote network layer, from the remote device layer, from the local device layer, from the local network layer, from the local transport layer and so on. Each merely signals the reliable delivery of a message to the corresponding layer but nothing more. As a particular example, datagram sockets in Berkeley UNIX unblock when the message has been copied to the protocol layer.

One of the requirements for the underlying communication system to support all these possibilities is to be able to send intermediate acknowledgements back to the sender. However, this capability is not provided in most communication systems because it is difficult or costly. Since all of the above possibilities of partially synchronous message-passing provide less than the guarantee of delivery to the final recipient, they also only provide loose synchronization. They all provide tighter synchronization than asynchronous message-passing but nevertheless not tight enough to use as a synchronization mechanism. Therefore, entities using partially synchronous message-passing would require an explicit user synchronization mechanism if end-to-end reliability is required (*e.g.*, constructing RPC above UDP [Post80]).

In these IPC examples, it can be seen that there is some ambiguity in the definitions of synchronous and asynchronous which must be resolved by consideration of the context or granularity of the environment. A user may view a send primitive as a synchronous or atomic operation, while to the system the same primitive is a collection of asynchronous steps. On closer examination, it is clear that there are two aspects to the problem, one dealing with the user execution thread and the other with control of the data object. The semantics associated with the blocking and unblocking of the data object[7] are often of greatest concern.

Blocking and unblocking of the data object in message-passing is related to the integrity of data. The user data is said to be *blocked* when it has been passed to the underlying communication system and any change to the data may disrupt the communication process. The user data is said to be *unblocked* when it has been released by the underlying communication system. The user data is released when

[7]Throughout this thesis, the terms *blocking* and *unblocking* of data can be interchangeably used with the terms *locking* and *unlocking* of data respectively.

the operation has been completed or a copy of the message is made to a safe place for subsequent transmission. Only when data is unblocked is it safe to modify it for reuse or delete it. Thus, if a copy of the user data is made, the user data may be unblocked immediately and any change to it will not disrupt the communication process. However, the convenience and simplicity resulting from copying of data comes at the expense of more CPU time and memory space for the communication system. This could potentially become a serious problem if the user data is large (on the order of megabytes).

The opportunity to corrupt user data during the communication process in synchronous and highly synchronous message-passing is much less than in asynchronous and partially synchronous message-passing. This is because the user's thread of execution is blocked until the data has been safely delivered to the receiver in synchronous message-passing, and until the user reply is received in highly synchronous message-passing: the processes can not possibly access the data until their threads of execution are unblocked. In both asynchronous and partially synchronous message-passing, however, the user's thread of execution is unblocked before the message is delivered safely to the end receiver. Thus, there is a possibility of corrupting the data before it is safely delivered to the receiver, unless a copy of the data is made by the underlying communication system. This is why most systems that support asynchronous or partially synchronous message-passing opt to make a copy of user data for subsequent transmission.

In addition to the one-to-one message-passing discussed above (*i.e.*, one sender and one receiver), message-passing includes the possibility of one-to-many communication (*i.e.*, one sender and many receivers), of which *broadcast* and *multicast* are two well-known examples [Cher85, Lian90, GM89]. At the hardware level, many networks support both a broadcast and a multicast facility. In broadcast, a message

is sent to all hosts connected to the network. In multicast, however, a message is sent to a certain subset of those hosts. At the interprocess communication level, the term multicast is more frequently used than broadcast since it is seldom necessary to send a message to *all* processes in the entire system. Multicast facilities usually provide operations to create, delete, join, and leave a group [Cher85].

Regardless of the level at which broadcast and multicast are used, they possess interesting properties that can be useful in certain applications. In general, neither broadcast nor multicast guarantee the delivery of data - no automatic retransmission is implied if messages are lost, and no acknowledgement of message delivery is expected from the receiver. (Of course, there exist broadcast and multicast communication proposals that guarantee the delivery of data [Powe83, Birm87b].) Since it takes approximately the same time to broadcast or multicast a message to a group of receivers as to send it to one specific receiver, it is faster and cheaper to multicast a message than to send many one-to-one messages [Bal89]. These characteristics are useful for applications such as a naming service. A kernel might broadcast a name look-up request to all hosts and it would be satisfied if only one of them replied. Another important characteristic of broadcast and multicast is that they may guarantee a certain ordering of messages that can not be easily obtained with one-to-one messages. This characteristic is useful in applications such as updating replicated data [Birm88].

Mailbox communication is an example of many-to-many message-passing (*i.e.*, multiple senders and multiple receivers) [Pete83]. In mailbox communication, the role of a relay is expanded to include memory resources, such that the sending party need never block or explicitly synchronize with the receiver. A message is copied from the sender to the mailbox (an intermediate message depository), and later from the mailbox to the receiver. Note however, that a receiver is implicitly

obliged to wait for a send event before it may successfully complete a transaction. Mailbox communication can be useful in providing system services such as a printer service, where there are multiple users and multiple printers. A mailbox, in this example, can be used by a printer server which receives files from users to be printed, receives printing job requests from printers and assigns files to available printers.

Distributed mailbox communication was demonstrated by the concepts of Tuple Space [Gele85, Mats88] and Distributed Rendezvous Store [Lau86]. Both concepts employ the mailbox-like message repository or repositories, each associated with a server entity. Message send and receive requests are sent to a server entity, which then performs a matching operation, and the intended message is transferred when the matching operation is successful. These send and receive operations are inherently asynchronous, though synchronous communication can be simulated. Since these concepts are generally implemented using server entities, an explicit synchronization mechanism is not necessary.

A variation of mailbox communication is called *port* communication [Balz71], in which there are multiple senders but a single receiver only. Thus, port communication is an example of many-to-one message-passing. The UNIX operating system uses the port concept widely in various system services such as remote login and mail [Koch89]. Each service is assigned to a fixed *well-known port*, and users send messages to this well-known port and a server receives messages from it.

In both mailbox and port communications, supporting long messages (*i.e.*, consisting of multiple message fragments) is inherently difficult. When multiple users send fragmented messages simultaneously, the messages may be interleaved and thus cause interference. Careful removal and reassembly of message fragments are required by the receiving entity.

2.2.2 Streams

A *stream* is another example of an interprocess communication abstraction. A stream provides sequenced, reliable, bidirectional connection-based data transfer. In current use, there are two major implementations of streams: one in Berkeley UNIX sockets (stream with lower case 's') [Leff88] and another in AT&T System V UNIX (Stream with upper case 'S') [Ritc84]. The Berkeley stream is mainly used for IPC, whereas the AT&T Streams are used for both IPC and network communication (*i.e.*, user to device or pseudo-device). A Stream developed by Ritchie contains rigorous definitions of components such as modules, queues and messages. AT&T Streams are implemented by passing messages between modules in a Stream, whereas Berkeley streams does not define the implementation and may run as pipes for local communication or on top of TCP [Post81] for remote communication.

Although their implementations and internal structures are different, they have several common features worth mentioning. They were both originally designed for character I/O and extended for use in IPC. They both provide bidirectional, reliable, sequenced, and unduplicated flow of data. In both streams, there are no packet or message boundaries, except those imposed by applications using their own delimiting conventions. They have the same user interface, the *write* operation for sending data and the *read* operation for receiving data. The write or read operation simply specifies as arguments an arbitrary amount of data to send/receive and the starting address of data to be sent/received. The sender and receiver can continue producing and consuming data as long as the stream is not full or empty, respectively; otherwise they block.

Message-passing, which requires message boundaries, can be easily implemented

on top of streams by inserting some protocol which has the effect of creating message boundaries [Auer90]. Conversely, streams can be implemented on top of message-passing by adding some logic to ignore message boundaries. However, it is not usually possible to allow mixed interfaces at the ends of communication. That is, current systems either do not allow or have difficulty supporting the streams interface at one end and the message-passing interface at the other. Such capability of supporting mixed user interfaces may be quite useful for applications running on heterogeneous systems, where one host may only support streams and the other only message-passing.

2.2.3 Shared Memory

In the previous two subsections, two categories of interprocess communication were discussed: message-passing and streams. In this section, interprocess communication based on shared memory is discussed. The basic idea behind shared memory IPC is that the independent communicating entities share a common area of memory for sending and receiving data. The shared memory may be physical memory to all communicating parties, one party's memory with access to it given to other parties, or a third party's memory (*i.e.*, a party that is not directly involved in communication) with access to it given to all communicating parties.[8] Transmission of data is achieved by writing data into shared memory by the sender and reading data from it by the receiver.

Unlike procedure calls, which are inherently a one-to-one operation, one-to-one, one-to-many, many-to-one, and many-to-many operations are all possible in the shared-memory communication paradigm, as in message-passing. For situations

[8]Secondary memory can be also considered as a third party's memory, which can be shared. However, *shared memory* is used to mean main memory.

other than one-to-one, the *multiple readers/writers problem* arises [Pete83]. In order to ensure that conflicts do not occur which may produce unintended results, an explicit synchronization mechanism is generally required in shared memory communication. Many synchronization techniques based on shared memory have been studied extensively. Some examples of these techniques are *busy-waiting* [Pete81], *semaphores* [Dijk65], *conditional critical regions* [Hoar72], *monitors* [Dijk68], and *path expressions* [Camp74]. All of these techniques (other than busy-waiting) need to be used with an explicit wakeup function, which would signal the blocked processes. An excellent survey of these techniques can be found in [Andr83].

When communicating entities reside in a common protection domain within a single host, data can be written into and read from shared memory directly (*i.e.*, without copying data). On the other hand, when communicating entities reside in separate protection domains within a single host, shared memory can be provided only with the help of a third party, such as a kernel. Shared memory IPC provided in System V UNIX is such an example. When communicating entities are located on different hosts, shared memory can be simulated by sending messages to the host where the actual shared memory resides. Duplicate copies may or may not be maintained on both machines. This form of shared memory is generally referred to as *distributed shared memory*.

The main advantage of the shared-memory communication paradigm over message-passing is its efficiency and simplicity. As mentioned earlier, it avoids the copying operation (which in general is more expensive in terms of memory space and time) as much as possible. Within a single protection domain, it also allows transfer of arbitrarily complex data structures by passing pointers to them. Across protection domains, however, the structural information of data must be transferred along with the content if the conceptual efficiencies of shared memory

are to be maintained. Note that there exist restrictions of which structures can be passed; these restrictions depend on specific details of the shared-memory semantics involved. For example, a pointer that points to a location outside the shared memory may cause a higher-level consistency problem.

2.3 Network Communication

Network communication encompasses such diverse aspects as process to device communication, kernel-to-kernel communication and such functions as would be part of the details of the lower layers of the ISO-OSI Reference Model [Zimm80]. As such it is often tailored to highly specialized environments and usually encompasses several modularized layers of functional processing. In addition to many of the problems of the preceding communication paradigms (such as memory and temporal independence), several new difficulties arise.

At the lowest level, there are Ethernet broadcast packets [Metc76], serial line character streams or direct memory access (DMA) ring buffers. At a slightly higher level, the physical form of device data is logically arranged as datagrams, sequenced packet streams, or logical disk blocks, and the node interconnection as a virtual circuit, or store-and-forward network. One side effect of this is that more and more intelligent devices are being developed with standardized high level interfaces to specialized internal versions of network protocol layers. By thus raising the hardware device interface to higher levels in the system software hierarchy, the system task becomes a more manageable one of dealing with a small number of standard logical communication interfaces [Auer90].

In contrast to previous communication paradigms, the data is not generally accessed or manipulated as a clean logical structure, but is often fragmented or

converted into diverse representations, which must be reconverted and reassembled by an inverse process to be usable by the receiving entity. Fragmentation of data is required by many device constraints, but it is often not precisely known what the optimal fragment size might be until the lowest protocol layers. Moreover, the size might change several times in the course of delivery through intervening nodes or modules. On the other hand, small fragments are generally buffered in the fragmenting peer module at the receiver side, even though delivery through the upward path and reassembly at the end-user interface might in fact result in less strain on system resources. In other cases such as asynchronous serial character devices, buffering or reassembly may be needed to reduce processing overhead through the layers to acceptable levels.

Reassembly often does not reproduce the precise original structure, but rather attempts only to preserve content and order. This is because the conventional network protocols only deliver the content of data, ignoring any structural information. If the precise original structure of data is required at the destination, structural information must be explicitly sent to the destination along with the content of data. Structure conversion protocols such as XDR [SUN87] and ISO ASN.1 [ISO87] are frequently used to achieve this goal. This process can be viewed as a form of data conversion. Another form of conversion involves converting the byte format of data, from ASCII to EBCDIC for example. As another example, there currently exist two types of byte ordering for sending data over the network: *big-endian* and *little-endian*. In *big-endian* (or network order), the most significant byte is transmitted first. In *little-endian*, the least significant byte is transmitted first. The conversion between these two byte orderings is required in a heterogeneous network environment, where computers from different manufacturers which support different byte ordering are involved in communication.

Every network communication protocol includes control information, which is encapsulated in a protocol *header* (and a *trailer* in some protocols). In general, the header is attached in front of the data and the trailer is attached behind the data. One of the responsibilities of a network protocol is to wrap control information around the data. This process of adding headers and trailers to data is generally referred to as *packetization*, and the reverse process (*i.e.*, extracting header, data and trailer from a sequence of data bytes) is referred to as *depacketization*. The performance of a network application depends heavily on the speed of protocol control information processing at each layer. Since there exists, in general, at least several protocol layers in most communication systems, slow protocol control information processing can add up quickly and thus affect the performance of the overall application. Hence, it is desirable to have a structure of data that is flexible enough to add and remove headers and trailers, and which also allows efficient protocol control information processing.

One significant constraint on the layered approach is the need to pass data from layer to layer through each intervening interface. As the number of layers increases, the network communication costs escalate. On the downward path from user to device interface, increasing privilege permits optimizations such as passing by reference, and in general full utilization of resources allocated to higher levels. This can, however, lead to a significant problem in freeing such resources, especially if the downward processing thread is suspended and further processing is carried out by subsequent kernel processes activated by interrupts. This is because when a user process is interrupted in the kernel space, the new interrupt-generated process does not have the same areas of accessibility as the original user process. Even worse is the upward path on the receiving end, where user resources may not yet be available to receive the data and thus the system must temporarily store the data;

system-wide resource allocations must be pre-configured to meet a wide range of anticipated needs. In any case, the final destination of such data can not be known without first traversing each layer in reverse. The crossing of protection domains further complicates the processing as it means that physical copies of data must be made at each such boundary with accompanying duplication of demands on system resources or reduction in the possibilities for optimization. These effects serve to make the upward path a significantly more complex operation, and have led to intense efforts to incorporate backwards flow control or throttling techniques in network protocols [Tane88, Zwae85, Cart89, OMal90].

Transmission is, at least at some point in the process, by value and not by reference, and thus resembles a relay or pipeline. Network communication channels are often used by IPC as the relays to transmit data between disjoint memory modules. Also, transmission invariably involves timing and synchronization aspects which are filled with asynchronous independent events that do not lend themselves well to normal serial processing. This can result in significant inefficiencies in overhead processing to maintain and switch contexts[9], or in resource scheduling to insure that proper management avoids bottlenecks and deadlocks.

In this chapter, various communication paradigms have been examined as a means to identify and categorize some of the fundamental aspects of all communication. In particular, aspects of intraprocess communication (communicating within the bounds of a single process), interprocess communication (both within and across protection boundaries), and network communication (general device and kernel-to-kernel communications) were examined for similarities and differences. The results of this examination have led to the more formal analysis of communication pre-

[9] *Context* is defined as the information accessible to the processor during an execution. The context includes a processor context (programmable and internal registers) and a memory context (code and data segments).

sented in the next chapter.

Chapter 3

Communication Issues and Abstractions

In this chapter, the fundamental aspects of communication discussed in Chapter 2 are examined more fully and redefined and organized into a set of communication abstractions that I believe is necessary to form the basis for a generalized communication paradigm.

3.1 Communication Issues

In Chapter 2, communication is defined as the transfer of data between two or more entities. Three components, namely data, entities and transfer (delivery and synchronization), are thus minimum requirements for any communication. In this section each component and issues involving it are examined fully as a means to develop the necessary requirements of communication abstractions.

3.1.1 Data

In this section, detailed discussions on a comprehensive set of issues related to data communication in distributed systems are presented. Data (or messages) may be passed from one entity to another either by reference or by value. Passing by reference, realized in a single memory domain, is more efficient[1] both in time and space than passing by value since it does not involve operations on individual bytes of data (*i.e.*, copying). Passing by reference generally requires only a few bytes of memory for passing a pointer to data of arbitrary size, whereas the amount of memory and time required increase with the size of data when it is passed by value. The memory resources required can become significant if communication involves data being passed by value from one layer to another as in network communication or if the size of data is large (*e.g.*, hundreds or thousands of kilobytes).

Passing by reference, however, has shortfalls as well. One of the main concerns in passing data by reference is the integrity of data during the transfer. The data must be guaranteed to be unmodified by the sender or some entity other than the receiver once the transfer is initiated and until the transfer is complete. On the other hand, when data is passed by value, a copy is made by the relay. Thus, the original data may be accessed or modified without any fear that this will affect the communication process.

The shared-memory communication paradigm discussed in Chapter 2 is based on passing of data by reference. Since communicating entities can both access the same memory using a pointer to it, passing the structural information as well as the content is not a problem. However, the same does not hold true for communication across protection domains, where copying of data is a necessity. When copying data,

[1]Efficiency can be subdivided into efficiency in time (*i.e.*, the amount of time required to perform an operation) and in space (*i.e.*, the amount of memory space required).

its structure must be understood by the copier, which often limits such structures to the simplest forms, such as a single contiguous sequence of bytes. More elaborate schemes (such as XDR [SUN87], ASN.1 [ISO87]) generally require an extra protocol layer to provide a structural description of the data to be transferred, which is necessary if a single memory communication paradigm is to be provided across the system.

As data is transferred, it may undergo a number of transformations. Most communication places constraints on the size of data that can be transmitted at one time. This may require the user data to be broken into a number of smaller data fragments on the sending side and then reassembled on the receiving side. Conventional network communication protocols require or make it advisable that data fragments be reassembled by the same (peer) layer on the receiving side as they were fragmented on the sending side. The rationale for this is the gain in modularity and independence of such protocols. This, however, may place more strain on system resources, particularly when large messages are being transferred. Reassembling fragments of a large message at the peer layer requires memory large enough to hold all the message fragments, and the user must provide at least the same amount of memory to receive it. This effectively requires at least twice as much memory as the original message. Such a strain on resources may be avoided if the capability to reassemble at an arbitrary layer in the communication system is provided and the memory provided by the user is available rather than requiring extra memory for an intermediate layer.

Furthermore, reassembly by conventional network communication protocols usually does not reproduce the original structure of data precisely, but rather preserves only the content and order. This is because the protocols do not, in general, deliver structural information about the data. Data format conversion (*e.g.*, ASCII to

EBCDIC and vice versa) is another form of data transformation, which is necessary in a heterogeneous computing environment (*i.e.*, when computers from different manufacturers are involved in communication).

There is often additional control data associated with simple data blocks. Although most modern computer systems supporting network communication do not follow the seven layers suggested by ISO-OSI Reference Model exactly, they do support a varying number of layers. Each layer is responsible for its own set of functions and is generally represented by a communication protocol. When the user wishes to send data to a remote entity over a network, the user data must traverse from layer to layer through each intervening interface. The control information (header and trailer) is added by each layer on the sending side, and removed on the receiving side. Flexibility and common standards should be provided in the structure of data so that the headers and trailers can be added and removed efficiently, while ensuring that data can be easily passed by reference or manipulated from one layer to another.

In order to distribute data objects transparently to nodes scattered about the distributed system, two requirements are essential. First, an identification scheme that can uniquely identify data objects across a system is required. Among many uses, it can be used to distinguish data objects arriving in a single node and to demultiplex message fragments out of interleaved messages from multiple senders. Second, a protocol is needed to transfer the structural information of data objects as well as control information for peer-to-peer interactions.

Finally, one of the most important system resources involved in communication is memory. In communication, memory is required for storing message components such as user data, control information, and descriptors that describe any of these message components. Management of memory for these components is referred to

as *buffer memory management*. Low-level aspects of memory management, such as dealing with how and when memory is allocated and deallocated, are complex issues which are best left to the user of the communication subsystem rather than being prespecified by the communication software. However, the communication subsystem must support memory management conventions that allow the efficiencies of shared memory to be exploited by avoiding copy operations wherever possible, and that minimize the need for expensive system resources for dynamic memory allocation and deallocation. This is most simply accomplished by mandating that the communication subsystem should return control of memory to the owner (*i.e.*, the entity who originally allocated the resource) when it is no longer needed. The principle involved is that the owner of the memory resource should be the only entity that can destroy it and that destruction should only take place some time after control is returned. This is a reasonable and necessary restriction needed to maintain order in resource management. It should be also noted that control and access are two different aspects. *Control* refers only to the determination of which entity is authorized to access the resource. *Access* refers to the right of reading and/or modifying the resource. Ensuring that only authorized access is possible is a function of protection mechanisms.

In this section, a set of fundamental issues related to data in communication in distributed systems has been discussed. Those issues included passing of data (either by reference or by value), fragmentation and reassembly, transmission of content versus structure, packetization/depacketization, resource management, ownership, distribution of data objects and control of buffer memory. If one wishes to support a wide spectrum of communication types among various types of entities, one needs a uniform data structure that is simple and general, low-level and extensible. It must be possible to manipulate this data structure efficiently both in

space and time when it is passed from one entity to another.

3.1.2 Communicating Entities/Endpoints

In this section, detailed discussions on a comprehensive set of issues related to the communicating entities or endpoints involved in communication in distributed systems are presented.

There exist various types of communicating entities in distributed systems: conventional procedures, coroutines [Krak88], kernel routines and ROM hardware routines, which communicate within a single protection domain; and threads (*i.e.*, light-weight processes) [Acce86], and processes (*e.g.*, UNIX processes), which communicate both within and across protection domains. Many researchers have introduced various high-level abstractions (*e.g.*, message-passing, streams, RPC) in attempts to provide a single consistent user interface that encompasses different underlying transport techniques. However, these abstractions have been designed to support communication between the same or similar types of entities. At a low level, there are network communication protocols and hardware devices with well-defined interfaces. As discussed in Chapter 2, these different types of entities, communication abstractions and implementation mechanisms are characterized by vastly different communication semantics and interfaces. This makes communication between different types of entities, abstractions and mechanisms a very difficult (if not impossible) task unless the external interface for all types is made uniform. Some possible interactions in this diverse programming environment are shown in Figure 1.1. Currently, most of these interactions are implemented by programmers using their own *ad hoc* approaches.

In this kind of diverse communications programming environment, there is a

need for simpler, universal concepts and tools (*i.e.*, a uniform data structure, functional interface and delivery/synchronization operations), which can be used by all types of partners for all types of communication. In particular, there is a need for a universal interface (which I call the *internal communications interface*), which can utilize the conceptual efficiencies of a shared memory paradigm. For example, various types of entities can use this universal interface to communicate with high-level communication abstractions, and in turn can use the same interface to communicate with low-level implementation mechanisms. Ideally, the universal interface would be used to communicate between entities using different local communication abstractions (*e.g.*, an entity with a message-passing interface attached to another with a streams interface). Further, the universal interface can be used between implementation mechanisms as well. For example, all layers in a network protocol stack could be linked in a uniform fashion, thus providing an easy direct interface to any sub-layer (*e.g.*, to TCP, IP) from any other. Distributed communication, which may involve multiple remote machines, can still enjoy some of the efficiencies of shared memory if this universal interface can be naturally extended to remote machines.

In actual implementations, however, the universal communications interface must be adapted internally to the characteristics of the owner entity. This can be achieved through local or type-specific conversions by the owner entity. Generic interface objects can be provided, which are then specialized by the owners to suit their own types. These interface objects should provide a storage area so that control or protocol-specific data structures can be added, and a capability for type-specific operations to be added or redefined.

Another important aspect related to communicating entities is *connectivity*, which is concerned with maintaining logical paths for data transfer. Connection-

oriented communication (*e.g.*, virtual circuit) explicitly constructs a logical pipeline between two communication endpoints before data transfer can take place. The construction of such pipelines should be flexible so that they can be set up either at system configuration time or at run time. Thus, operations that will connect entities together and disconnect them on demand must be provided. Although connection-less communication (*e.g.*, datagram) does not involve any explicit connections, minimum routing information must be maintained in each entity (both intermediate and end entities) for transfer of data from one point to another. I do not feel that providing data storage and operations to maintain such information is a fundamental responsibility of the communications interface, but rather of the individual protocol module or owner. However, since the connectivity issue is related to the delivery aspect of communication, it is not unreasonable to provide a capability to include routing information and operations in the interface object.

An important issue closely tied to connectivity of communicating entities is *naming*. When a sender wishes to send some message, the name or identification of the destination must be specified. In most systems, users of communication do not usually deal with physical addresses (*e.g.*, Ethernet or Internet addresses) but with logical names, which eventually get mapped or translated to physical addresses by the subsystem. Name servers are typically used to locate entities the names represent. For a universal interface that can be used at all levels of communication, an abstract naming scheme which ignores the details of any particular physical addressing schemes is desirable. I envision that an identifier represented by Abstract Syntax Notation [ISO87] or a 64-bit integer value would suffice. Since the issue of naming or name service is outside the scope of this research, the details are not discussed here, as the topic is discussed at length in [Reed78, Wats81, Cher86c, Lau87, Radi90].

In this section, a set of fundamental issues related to endpoints and interfaces for communication in distributed systems was presented. These issues include protocol issues (working and target environments), universal interfaces, local conversions, connectivity and naming.

3.1.3 Delivery and Synchronization

In this section, detailed discussions on a comprehensive set of issues related to delivery and synchronization are presented. Data delivery techniques vary greatly in different communication paradigms. In Chapter 2, a survey was done on various communication types and paradigms and their delivery techniques were discussed. Delivery of data is called by different names in different communication paradigms: send/receive, read/write, get/put, store/retrieve, push/pop, call/return, *etc.* Some examples of delivery techniques are transmission via stacks or globally referenced variables in procedure calls, writing into and reading from shared memory in shared-memory IPC, and copying or transmitting data from the source to destination in communication between disjoint protection domains. Generalized delivery primitives should provide the fundamental building blocks to encompass all of them.

Closely associated with the data delivery is a signalling mechanism. A *signal* is defined to be a way of notifying the receiver that control of data has been transferred from the sender to the receiver, and thus that the data is available for reading (receiving) and processing. This is necessary for all delivery techniques, since the receiver must have some indication that data has arrived and is available for access. Thus, the signal function is part of all delivery operations. As with the delivery techniques, there exist different signalling techniques for different communication paradigms. For example, the jump instruction is used both to transfer control and

implicitly signal the availability of data in procedure calls, while a wakeup signal may be used to unblock a process awaiting the arrival of data in message passing.

To handle various delivery and signalling techniques for various communication entities, a generic expression of the delivery technique and the signalling technique is desirable. Because both delivery and signalling techniques are tightly related to entity types, there should be associated means for the entities to define or specialize them to their own specific environments. Further, delivery means possible access by multiple entities. In order to guarantee the integrity of data, these operations may need to be executed under the control of a privileged entity such as a kernel (and thus would be stored in a kernel library).

User data may flow either unidirectionally as in the *send-receive* IPC paradigm or bidirectionally as in the *request-receive-reply* IPC paradigm. In these and other communication paradigms, some form of acknowledgement or status is expected on completion of the operation, even when user data or control information travels in only a single direction. I believe that returning the control of data objects to their owners at completion is a useful principle for local system resource management, regardless of whether the data flow operation was unidirectional[2] or bidirectional. Moreover, if the principle of always returning control of data objects to their owners when the operation is complete is enforced, the operation status or ACK can always be returned at little or no additional cost by piggybacking it on the returning data object.

Another important issue related to data delivery is efficiency. Although most modern computer systems are equipped with a large amount of memory (tens of megabytes), the *bounded buffer problem* [Pete83] still exists. That is, programmers

[2]For remote unidirectional communication (*i.e.*, involving transmitting packets over a network) the logical return of data objects from remote nodes should be suppressed as an optimization.

have to deal with a finite amount of memory and must take necessary measures to make programs as memory efficient as possible. Several examples of these measures in implementing data delivery are as follows. On the sending side, data may be buffered (*i.e.*, small fragments are combined to form a larger fragment) before being passed on to avoid wasting bandwidth by sending partially filled packets. Modern systems are also equipped with DMA devices [Hama84], and scatter-gather capabilities [Inte84], which can directly access user data at transmit or receive time and thus reduce the overhead of copying operations. On the receiving side, data may need to be reassembled before delivery to the final destination, in which case it might be desirable and less of a burden on system resources to reassemble data at an arbitrary layer (*i.e.*, not necessarily at the same layer as it was fragmented on the sending side).

Synchronization in communication can be broken down into two aspects: synchronization of user execution of sender and receiver (or user blocking semantics) and synchronization of data-object control (or blocking semantics of data objects). In synchronous communication, the user is blocked while waiting for the completion of the operation and return status. Here, the *signal* function discussed above can be used to unblock users. Assuming that the receiver is blocked waiting for data to arrive, the delivery operation will first transfer the control of data and then invoke the signal operation to signal the arrival of data to the receiver. When the status information is returned, the signal operation is also invoked, this time to signal the arrival of the status to the blocked sender. In asynchronous communication, the user does not block, but continues execution immediately after initiating either the send or receive operation. The only way the user can discover the status of the operation is either to poll for the completion or to explicitly block and wait for it. If the user later blocks waiting for the return status, the unblocking process

becomes the same as in the synchronous case. Although the signalling mechanism may not always be used in asynchronous communication as in synchronous communication, it should still be part of the generic delivery mechanism. That is, the signal function is receiver-defined but should be executed as a mandatory part of delivery by any sender. (Note that the sender of data is usually a receiver of a later acknowledgement. Thus, each entity involved in the communication may have a different implementation of the signal function.)

Synchronization of data-object control deals with the blocking semantics of data objects. During synchronous communication, the user does not have access to the data object until the user is unblocked and hence the integrity of the data object is guaranteed assuming the data object is not being shared by other processes. Thus, in synchronous communication the data object is unblocked at the same time as the user is unblocked. In asynchronous communication, however, the user is not blocked after initiating the data transfer, and thus can potentially access the data and modify it before the operation completes. In order to avoid the problem of determining when the data is unblocked, many systems supporting asynchronous communication make a copy of the user data when the user initiates the transfer, thus unblocking the data before the return from the initiation process, but this requires additional resources and time to copy. What would be more desirable is a rigorous principle for data-object control that would be adhered to by all participating entities for all types of communication. An example of such a principle is one that utilizes the conceptual efficiencies in passing a description of data objects rather than data objects themselves. Thus, one must obtain physical control of this descriptor to obtain control of the data object. If the descriptor is not possessed by an entity (even if the entity is the owner, or even if the entity has access to it), it does not have control of it. A principle like this can promote

efficient and reliable resource management.

In this section, fundamental issues related to delivery and synchronization were discussed. To handle various delivery and signalling techniques for various communication entities, a generic expression of delivery and signalling is desirable. This should be accompanied by a set of general principles discussed in this section to achieve efficiency and security.

3.2 General Organization of Communication Abstractions

In the previous section, fundamental aspects of communication as a means to identify the requirements of communication abstractions for distributed systems were discussed. Further, useful principles that can accompany these abstractions to achieve efficiency and security were discussed. In this section, I discuss hierarchical organizations of these communication abstractions, namely *data*, *node* and *delivery/synchronization*, that I believe should form the basis of a generalized communication paradigm.

In Chapter 2 and Section 3.1 (using Figure 1.1), the diverse communication programming environment and the problems software developers must cope with to support various types of communication between various types of entities were presented. The main problem, as stated earlier, is a lack of a universal interface and uniform data structure. I propose to replace the multiplicity of existing interfaces used to implement most current communication systems with *internal communication* (shown in Figure 3.1), which consists of a universal functional interface and uniform data structure. Internal communication is one of two major goals of the

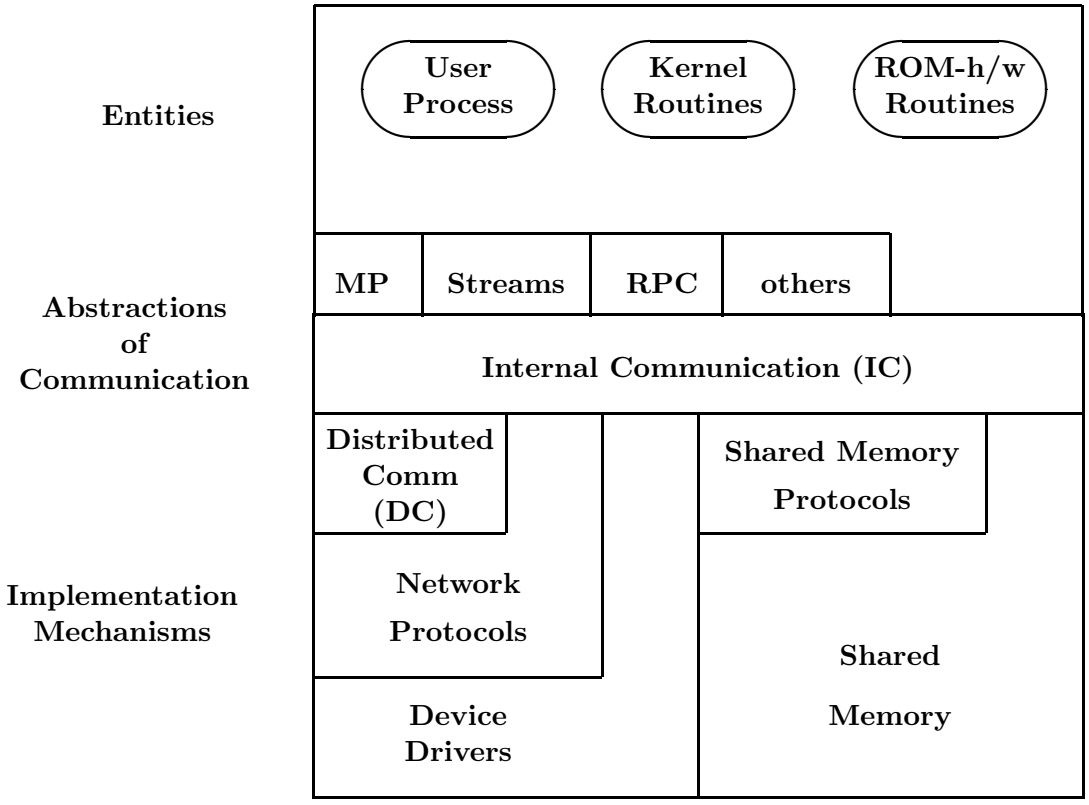

Figure 3.1: The Internal Communications Programming Environment

generic communication model. The other major goal is to provide a hierarchical framework upon which arbitrary communication paradigms can be developed.

3.2.1 Data Abstraction

The fundamental logical abstraction of *data* is that of a memory object consisting of an ordered sequence of bytes.

The simplest form of such an object is a single contiguous block of memory, which might be described by its starting address and size. Users require such information to perform operations such as locating the memory, writing data into it, and reading data from it. This set of information is referred to as a *descriptor*. Further, the term *buffer* is used frequently in operating systems and communications to describe an area of memory that could contain transient messages. Thus, the simplest *data* or *buffer object* consists of a data object descriptor, an actual block of memory for data, and a user interface that provides operations to create/destroy the data object, to read data from and write data into it, and to obtain data object descriptor information. This simple unstructured data object (called a *simple data object*) is shown graphically in Figure 3.2.

The user may send and receive a single block of data at once or fragments of data piecemeal, or he may wish to to send a structured object consisting of several fragments. Thus, the next level in the hierarchy is a data object that contains a sequence of blocks of memory, which are non-contiguous, in general. The capability to accommodate a sequence of memory blocks supports data fragmentation and reassembly as well as scatter-gather techniques. Two reasonable implementations of a sequence of memory blocks are linked-list and array. Hereafter, the use of a linked-list is assumed as the implementation of a sequence for simplicity. For

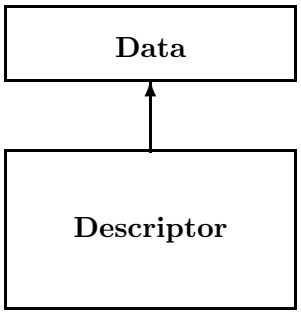

Figure 3.2: A Simple Data Object

example, the user may wish to send data in small fragments at a time, and these fragments may be buffered before they are actually transmitted to the receiver. If the fragments are pieces from a contiguous area of memory, they could be merged to form a single large block of memory, otherwise the capability to form a list of non-contiguous areas of memory is essential. As well, a large message may need to be fragmented into a number of smaller fragments before being sent, due to physical limitations of the transmission media . On the receiving end, the received data may also need to be fragmented and passed to the receiver in small fragments at a time, possibly due to the receiver's limitation on request size. Alternatively, small data fragments may need to be reassembled into the original data form before being delivered to the receiver.

In order to describe and manipulate a list of memory blocks, a linkage capability must be added to the simple data object described above. In addition, status fields to record the details of fragmentation (*e.g.*, sequence number, flag indicating whether the fragment is the last block in the list) must also be added. This will allow a list of simple data object descriptors, each describing a block of memory, inside a single data object (which I call a *segmented data object*). Standard operations

added with this type of data object include those for reading and writing data, for fragmenting a block of memory into two smaller blocks, for reassembling two blocks into a single large block, and for obtaining information describing the data. The code for the read and write operations must be changed from that for the simple data object described earlier to cope with the list of memory blocks.

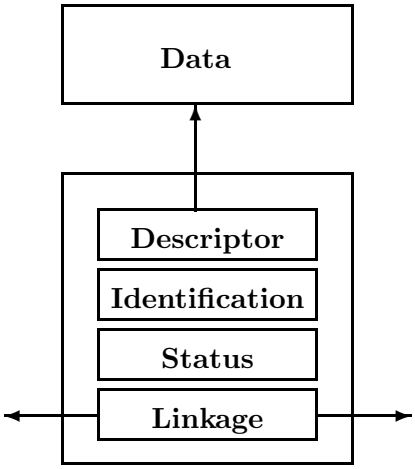

Figure 3.3: A Segmented Data Object

Identification fields might include data object type, identifier and ownership. There will be various types of data objects for various types of communication, thus the type of data object needs to be specified. Identification is useful and convenient for system management of memory blocks as well as for references to them by users. (Identification could be part of a simple data object; it is not fundamental to a segmented data object.) Since there may be hundreds or thousands of data objects in a system, a unique identifier for each object in the system is desirable. Further, since each object may consist of multiple blocks of memory, a scheme that can uniquely identify each memory block within an object is also required. I propose an augmented identification scheme, which uses a unique object identification scheme

for identifying data objects, and a unique sequence field for identifying each memory block within an object. Finally, a specification of the owner of each memory block is also desirable so that they can be returned and appropriately disposed of or reused at the completion of an operation.

Thus, the segmented data object consists of one or more simple data object descriptors, identification, status and linkage fields, and requires operations for manipulating a list of data object descriptors and the memory blocks they describe as mentioned earlier. This data object is graphically shown in Figure 3.3.

A data structure such as the segmented data object described above has been demonstrated to be inefficient for protocol processing [Hutc88, Zwei90]. The main reason is that it is not suitable for handling nested protocol layers and their embedded control information. In most types of communication, control information is transmitted along with data. This control information is often mixed with protocol information and contained in header and trailer fields of a protocol packet. Thus, the next level in the data object hierarchy is one that can handle control information efficiently in addition to the identification, status and linkage fields as described above. Functional capability is required for handling packetization and depacketization, which are an essential part of protocol processing, and so this descriptor includes control fields to describe a protocol header, and a trailer (if one exists) in addition to the user data.

Most modern communication systems are designed with multiple protocol layers. As discussed earlier, one of the major concerns of communication is how one can pass data efficiently from one protocol layer to another. Copying of data at each layer interface can be very expensive in terms of both time and space, and is compounded when multiple protocol layers are involved. A desirable solution is to pass data by reference (*i.e.*, without copying) as much as possible. Thus, the inter-

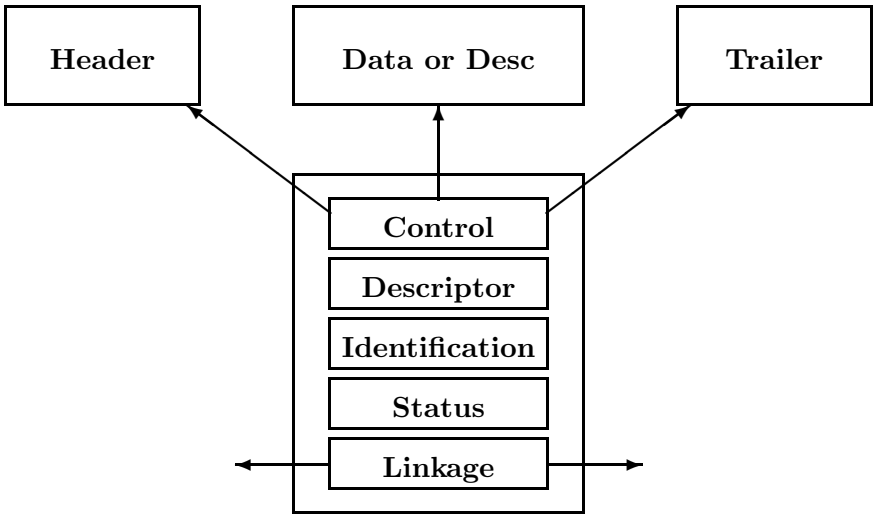

Figure 3.4: A Recursive Data Object

nal structure of data should be standardized in such a way that it can be augmented gracefully and efficiently, and the resulting data structure would correspond to a stack of protocol layers. This can be achieved by allowing recursion, where the user data fields in the data object descriptor point to either a simple user data object or to an arbitrary data object descriptor. The resulting form is a tree structure corresponding to a stack of nested protocol layers, with each layer possibly including header and trailer descriptors. Such a structure is open-ended (*i.e.*, extensible) and can easily accommodate an arbitrary number of protocol layers. In principle, each protocol layer could potentially make use of the recursive and segmented features.

A recursive form of segmented data object, shown in Figure 3.4, is well-suited for efficient packetization and depacketization. For example, when a recursive data object is passed from a higher layer, each network communication protocol module would create a data object descriptor (if not already created) for the layer, fill in the appropriate information, link to it the data object passed from above, and

pass the resulting data object to the layer below until information is eventually transmitted by a network hardware device. In this way, the data from one layer can be passed by reference between layers in protocol stack processing, avoiding expensive copying operations. Furthermore, the recursive data object structure and functional interface can be used as a "standard" structure for all levels of communication. It provides all the facilities needed to implement a communication protocol. Since all levels use this common structure, a uniform and simple software interface can be made for each layer. Thus, a layer module should not have to worry about the details of any of its higher or lower layer interfaces. The data is available to any intermediate layer module. This is possible through recursive data object pointers that can be followed by any module to access the data.

Since the recursive data object possesses all of the features of the simple and segmented data objects described above, it should inherit all the operations from them (modified accordingly for the new structure) and add some new operations. The operations that need to be redefined are read and write operations, which must traverse the nested structure to find the appropriate locations. In order to add and remove headers and trailers efficiently, new operations to push a block or to pop a block from a data object descriptor need to be added. These operations will allow non-contiguous blocks of memory to be dynamically and efficiently inserted and removed. Thus, this recursive data object may contain nested data object descriptors that can describe multiple levels of control and protocol, and operations to traverse this structure and manipulate parts of it.

The recursive data object described above possesses a data structure and a comprehensive set of operations that are common to various communication types. One can easily develop a particular protocol object by adding the type-specific data structures and operations to the recursive data object and/or redefining existing

operations. In particular, network protocol-specific data and operations need to be added to the recursive data object to create layered network protocol-specific data objects (TCP data object, IP data object, *etc.*).

The hierarchy of data objects presented in this section can be used to implement various data forms used in various communication paradigms.

3.2.2 Node Abstraction

A *node* is an abstraction of a communicating entity or endpoint. More specifically, a node is an abstraction of a communication interface object to various types of communicating entities.

The simplest form of node object is one that simply provides a capability to store and retrieve data objects. It basically provides storage for a list of data objects, where the sender may place data objects being transmitted and where the receiver may retrieve data objects. The static structure that "describes" such a capability is called a *node object descriptor*.

Figure 3.5: A Simple Node Object

Abstract data types (ADTs) such as *queues* or *stacks* [Stan80] are reasonable data structures that can support the functional requirements described above. They offer basic operations such as *enqueue* and *dequeue* in queues and *push* and *pop* in

stacks for storing and retrieving data respectively. The most common conventions for storing and retrieving elements are first-in/first-out (FIFO) for queues and last-in/first-out (LIFO) for stacks. Queues are more widely used ADTs in operating systems since the FIFO discipline can be used to maintain the proper sequence of pieces of data being delivered [ATT87]. Besides these generally accepted access disciplines, however, node objects must also provide other access modes for operations such as extracting fragments of a single message from a node object holding interleaved fragments of a number of messages, or for handling priority data. For implementation of these modes and efficient traversal and accounting of data objects in a node object, an internal pointer and a counter respectively are examples of basic descriptor information. Thus, the simplest node object contains a simple node object descriptor and operations to create/delete node objects, to store/retrieve data objects. The *simple node object* is shown in Figure 3.5.

The next level in the hierarchy of node objects is one that contains necessary information for communication and system management in addition to the descriptor of the simple node object. Specifically, identification and other status information are needed. Identification fields might include its type, node object identifier, and ownership. There will be various types of node objects for various types of communicating entities, thus the type of node object needs to be specified. Each node object should be associated with a unique identifier within a system for management purposes, and which can also be used as an address to which data can be sent and from which data can be received. The owner field specifies who created the node object (the semantics of owner will be discussed in detail in the next section).

This node object (a *system node object* as shown in Figure 3.6) should include status, identification and a descriptor, and operations to set and obtain this extra

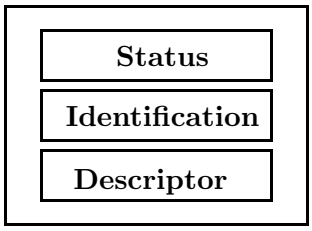

Figure 3.6: A System Node Object

information in addition to those inherited from the simple node object.

The next in the hierarchy of node objects is one that can easily manage a protocol layer. Network communication usually involves a message being passed through nested layers of communication protocols. At each layer, protocol-specific processing is performed such as adding header/trailer information and updating state information. Since the header/trailer information and the operations for manipulating it are stored within a data object, it is the data object that is modified as it travels down or up a protocol stack. However, the state information to manage each protocol layer cannot be stored in the transient data object. In keeping with the object-oriented design philosophy, the state information should be kept not with the protocol-specific code but with a code-independent data structure. There is no better place to put the state information than in the node objects. Thus, this node object (called a *network node object*) can potentially include a protocol specific data structure and operations on it. The network node object is shown graphically in Figure 3.7.

In a layered communication system, the stack of communication protocols represents a pipeline or path data must travel through. In connection-oriented protocols (such as TCP), explicit connections must be made, whereas in connection-less protocols (such as UDP) no explicit connections are required before data transfer can

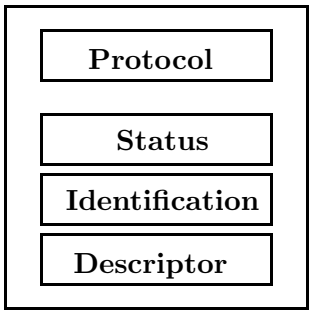

Figure 3.7: A Network Node Object

take place.

Two variants of interface are envisaged to support creation and destruction of logical pipelines. One is to provide a procedure call interface which provides explicit connection and disconnection operations. These allow users to connect protocol software modules and initiate a pipeline so that messages can flow or to disconnect them when they are no longer needed. Invoking any of the connection/disconnection operations will cause the owner of the node object to perform appropriate actions. For example, when responding to a connection request operation from the node object in the layer above, the owner of a node object would check whether the connection is valid and then complete the necessary steps. Note here that the exact semantics of these operations depend on the type of target entity. For example, when separate processes are involved, it will be the owner (or target) process that will actually respond to an invocation (or a message) from the process in the layer above. On the other hand, when a single thread of execution is involved, the thread will be merely executing the code contained in the target node object in the caller's context (*e.g.*, nested procedure calls).

The other type of interface to support logical pipelines is one that does not

provide these explicit connection/disconnection operations but deals only with "self-contained objects" which contain addressing and routing information. In this case, the code of the protocol module must be capable of deciphering addressing and routing information from received data objects, perhaps encoded in a header.

The preferred choice of interface is one that can handle both types described above. That is, the generic interface should be able to support a procedure call interface for setting up pipelines explicitly as well as the self-contained objects interface. The generic interface provides flexibility so that the implementor can simply add the necessary data structures and operations to the generic interface to tailor it. This design gives a large degree of freedom to the implementor in the choice of logical pipeline strategy.

In summary, the network node object described above possesses a static data structure and a comprehensive set of operations that may be required by various communication types. One can easily develop a particular node type by adding the type-specific data structures and operations to the network node object and/or redefining existing operations. In particular, network protocol-specific data and operations need to be added to the network node object to create network protocol-specific node objects (*e.g.*, IPC node object, UDP node object, *etc.*).

The hierarchy of node objects presented in this section can be used to implement various communicating entities or endpoints involved in various communication types.

3.2.3 Delivery and Synchronization Abstraction

Delivery and *synchronization* is an abstraction concerned with the dynamic functionality of transporting the data while preserving its form and content.

In the generic communication paradigm, the *store* and *retrieve* operations of node objects are the generalized delivery primitives. These operations, however, must be atomic, that is, the execution of either operation must be completed without any interference. How these operations may be implemented atomically depends on the types of entities involved in the data transfer. For example, since there is only a single thread of execution in the procedure call paradigm, providing atomicity is trivial. However, in the paradigms involving multiple threads of execution, implementation may require assistance of the system hardware instructions (at a low level) or of semaphores or monitors (at a higher level).

As mentioned earlier, there exist various types of communicating entities. Each may require a different mechanism to signal the delivery of data and/or the initiation of processing for that data. I propose a generalized mechanism for notifying all types of communicating entities of the availability of data and/or resuming execution. This signal function should be a part of the node object since the signalling mechanism depends on the type of the receiver. Thus, defining or specializing the signal function is the responsibility of whoever defines the communicating entity types.

Since the signal function is an essential part of the data send operation, we propose that it be a part of the store operation. Thus, the store operation consists of two major functions: one to transfer control of the data object and another to signal the receiver. The code of the signal function is defined by the owner of the node object. Thus, the sending entity does not know (and need not know) the details of the signal function. The sender's thread of execution merely invokes and executes the signal function as part of the delivery operation. For example, the sender's thread of execution is continued in the called routine in case of procedure calls and, in the case of IPC, may wakeup a blocked receiver or cause an interrupt,

which would reschedule the receiving entity to continue its communication process.

Three general principles have been chosen to be applied to the generic communication paradigm: 1) any entity can store data objects to a node but only the owner of the node can retrieve them, 2) the data object must always be returned to its "home" node at the completion of an operation, and 3) the return of data objects can be made directly or recursively. As mentioned above, sending a data object is achieved by storing a data object onto the node object that is associated with the destination entity, and signalling the receiver. Receiving a message is achieved by retrieving a data object from a node object. A data object may be added to any node object in the system. However, retrieve operations are restricted to the owner of a node object. This is a reasonable restriction which serves to maintain order in resource management. If more flexible store and retrieve semantics are required, this capability can be built on top of the current semantics. For example, multiple readers can be handled by interposing a server process which has specific code to deal with resource management, synchronization, demultiplexing of long messages and other interference aspects.

The home node, defined as the node whose owner created a particular data object, is used extensively for returning acknowledgements and synchronization as well as for resource management in the generic communication model. It is associated with each communicating entity mainly for the purpose of recovering a data object, which the entity has transferred earlier to another entity. It is where the owner of a data object resides. For example, a user process sends a data object containing a message to another process. The process either immediately (in case of blocking IPC) or arbitrarily later in time (in case of nonblocking IPC) blocks on retrieving this data object from its node object. As mentioned in Section 3.2.1, each data object contains the information about where it should be returned when

the operation is complete.[3] Thus, the return of the data object to its home node signals the completion of the requested operation. Furthermore, the return of the data object also signals its availability. The user process is unblocked and the data object is retrieved from the node object. The user process discovers the status of the operation (*i.e.*, whether the operation was successful or not) from the status contained in the returned data object.

Note that in the generic communication model, the return of data objects at completion of operations within a local host is required for all communications. In remote communication, however, ACK or status is designed to be returned to peers in synchronous or two-way communication but not in unidirectional or one-way communication. For one-way remote communication, the data objects should be returned their owners as soon as the packet transmission is made to the destination host over the network. Since an ACK or status must be returned from a remote entity (*e.g.*, a remote communication server) in remote synchronous communication, data objects are blocked until an ACK or status is returned from the remote host. This requires that the request type be specified by the sending communication server and recognized and acted upon accordingly (*i.e.*, no return of data objects for one-way remote communication). Thus, the home node concept can be used for good local resource management with very little or no overhead. The use of home nodes, I believe, is unique to the generic communication model. A more detailed, specific example of one-way communication will be presented in Section 5.8.

Many data objects in the system may reference a particular node object as their home node. Thus, one must be careful when destroying a node object since destroying one could leave dangling data objects which have this node as their

[3]The home node information is inserted into each data object at its creation time and is supplied by the creator.

home node. This has implications in that the resources that have been allocated previously by the owner of the node object can not be returned to the owner and thus can not be freed properly.

The generic communication paradigm provides a simple but versatile mechanism for returning data objects. In nested local procedure calls or RPCs, the return status travels along with the control in the reverse direction of the calls when it returns. However, returning the status of the data delivery involving multiple layers, where an independent thread of control is involved in each layer, is not as simple or clean. In many cases, it would be more efficient and convenient if the return status could be returned directly to non-adjacent layers. In conventional communication systems, this capability is generally very difficult to achieve or not possible at all. However, the home node concept coupled with the capability to access nested data object descriptors inside a standard recursive data object from any level provide an elegant solution. Any callee can either return the object descriptors in the reverse direction of the delivery path (*i.e.*, popping only the descriptor it had created and passing the rest to its caller) or return them all directly to the appropriate home node. Also, should some intermediate entity fail in the middle of a communication process, error status and descriptors can still be recovered and returned to appropriate places since each descriptor always contains its own home node information.

Thus, the delivery and synchronization abstraction of the generic communication model consists of generalized delivery operations, a generalized signal mechanism and a mechanism to return data objects. The return mechanism can provide an acknowledgement scheme, synchronization of user execution, synchronization of data control, and efficient resource management at little or no additional cost. However, in order to support these operations and mechanisms in a distributed

environment, an explicit protocol such as the Buffer-Queue protocol introduced in Chapter 4 is required.

In this chapter, a detailed discussion was provided of important issues related to the fundamental concepts of communication, namely data, entities and transfer. Based on this discussion, necessary requirements for communication abstractions were derived as well as some basic design constraints for a simple, efficient and general communication model that will provide a single consistent programming environment. Some of these constraints are 1) a standard data-object structure is required, which all levels and types of communication can use consistently, 2) a universal interface is required, which can be used among various types of entities across a wide spectrum of environments, 3) for efficiency reasons, the model should utilize the conceptual efficiencies of shared memory by passing a descriptor (or control) of data objects rather than data objects themselves, 4) any entity can store data objects to a communications interface object but only the owner of the interface object can retrieve data, and 5) a data object must always be returned to its owner at the completion of an operation. Providing this consistent programming interface across a distributed environment has never been explicitly included in previous work.

The major contribution of this chapter is to develop a general communication model using the object-oriented design methodology [Booc86, Meye88] based on a set of simple, efficient and versatile communication abstractions. This model provides a single efficient and consistent programming interface that allows different types of entities to communicate easily across a wide spectrum of environments, including distributed environments in particular. Further, this model can be used to develop both existing and new communication paradigms. In the next chapter, a specific implementation of the generic communication model called the Buffer and

Queue Model is presented.

Chapter 4

The Buffer and Queue Communication Model

In this chapter, the Buffer and Queue Model is developed, which satisfies the constraints of the generic communication model developed in Chapter 3. The Buffer and Queue Model is a simple, low-level but powerful and efficient communication model, which utilizes the conceptual efficiencies of a single memory domain while providing a universal interface among various types of entities across a wide spectrum of environments. In particular, the model includes the use of the "Buffer-Queue" protocol, which allows the paradigm to be applied to communication among entities scattered throughout a distributed system.

4.1 The Buffer Abstraction

In this section, the details of the Buffer abstraction are presented and how it handles various communication problems related to data is shown. The development of

an efficient, versatile, uniform data object structure that all levels and types of communication can use is also presented.

4.1.1 A Simple Buffer

A Buffer is an abstraction of a memory object consisting of an ordered sequence of bytes. The simplest form of Buffer is one that consists of a single contiguous block. A Buffer object consists of two parts: data and operations. The data portion of a Buffer object is referred to as the Buffer descriptor or Bufd.[1] A simple Bufd basically consists of four elements: the starting address of a block of memory used by the Buffer, the size of the Buffer, and the starting offset and the size of valid data. Having these latter two variables to specify the valid data as opposed to having just a single counter is useful when data fragments may be written and read concurrently, as in the *bounded-buffer problem* [Pete83]. The class definition for the simple Buffer is given in Figure 4.1. The class definitions throughout this thesis will be described in C++ [Stro86]. However, it should be emphasized that they could have been described in any object-oriented programming language that supports inheritance of data and operations and redefinition of operations.

Next, the details of the operations defined in the Simple_Buffer class are discussed briefly. The Buffer creation operation (*constructor* in C++) called *Simple_Buffer()* is the initialization routine invoked when a Simple Buffer object instance is created. It takes as an argument the address and the size of the data memory the user has created, and initializes the Bufd accordingly. The starting *address* field is initialized to the data buffer address. The *size* field is set to the

[1]Throughout the thesis, when the term Buffer is used it will mean the object in the object-oriented sense, while Bufd will mean only the area of memory occupied by the descriptor corresponding to a particular instance.

```
class Simple_Buffer {
   ADDR       address;        // start of buffer
   int        size;           // buffer size
   OFFSET     offset;         // offset of start of valid data
   int        count;          // number of valid data bytes
                              // (offset + count <= size)
 public:
   Simple_Buffer();           // Buffer constructor
   ~Simple_Buffer();          // Buffer destructor
   // write data into and read data from buffer
   virtual int    write_data( ADDR data, int len );
   virtual int    read_data( ADDR data, int len );
   // move the pointer to specified location in buffer
   virtual int    seek_data ( int type, OFFSET off );
   // get info related to valid data
   virtual OFFSET get_offset();
   virtual int    get_count();
};
```

Figure 4.1: The Simple Buffer Class Definition

size of data buffer. The *count* and *offset* fields are initialized to zero under the assumption that there is no valid data at creation time. The Buffer destruction operation (*destructor* in C++) called *~Simple_Buffer()* is responsible for disposing of the Buffer object.

Writing user data into a buffer is handled by the *write_data()* operation. It takes as its arguments the starting address of user data and its length. The user data will be written into the buffer from the location that the end address (*i.e.*, *offset* + *count*) of valid data indicates. The size of valid data will be increased by *len*. Reading user data from buffer is handled by the *read_data()* operation. It takes as its arguments the address where the data read will be stored and the size of data to be read from the starting address of valid data. The starting address of valid data is updated by advancing the offset by the size of data read. In both *write_data()* and *read_data()* operations, when an error condition occurs (for example, the size of data to be written or read is greater than the actual size of buffer), an appropriate error code will be returned to the user.

The valid data offset can be set explicitly by using the *seek_data()* operation. It takes as its arguments the type of seek and an offset value. I envision the use of three types of seek operation. The first option is ABSOLUTE, which sets the starting offset of valid data to be the specified offset. The second option is RELATIVE, which sets the starting offset of valid data to be the current offset of valid data plus the specified offset. The last option is APPEND, which sets the starting offset of valid data as the number of specified offset bytes from the end of valid data. In all cases the valid data counter will be adjusted based on the new value of offset. The starting offset of valid data can be obtained by the *get_offset()* operation. The current size of valid data can be obtained by invoking *get_count()*.

4.1.2 A Segmented Buffer

When a Buffer is involved in communication, it abstracts data or a message being transferred from one communicating entity to another. Communication based on Buffers and Queues involves transferring Buffers among Queues in the system. This requires extra information be maintained in Bufds in addition to data, namely identification and status information. The capability of handling a sequence of Bufds is also added, which is essential for message fragmentation and reassembly. The class definition for the Segmented Buffer object is given in Figure 4.2.

A pair of doubly-linked pointers, *q*, is used for implementing a sequence of Bufds. They are also used for linking Buffer objects to some Queue. Identification fields include *type*, *bid*, *sequence*, *owner* and *returnQ*. Since various types of Buffers are envisaged for various types of communication, the type needs to be specified. Note that depending on the programming environment, it might be possible to eliminate the type field. Recall that a data object identifier is needed for distribution. *Bid* uniquely identifies the Buffer object across the system under consideration. I propose a two-level identification scheme, hostID-localID pair, for Buffer identification. A 32-bit hostID can be used to identify the host and another 32-bit localID for local identification within a host. I believe that the hostID-localID pair naming scheme is sufficient for Buffers and Queues. Higher-level naming schemes may be implemented if desired, but naming is not discussed in detail since it is outside the scope of this work.

The *sequence* field is used when there are multiple Bufds in a Buffer object. Here, the ways of providing unique sequence numbers are briefly discussed. One way to provide unique sequence numbers is to use offsets in an ordered sequence of bytes. There are two options to this method. One is to use the sequence number

as the offset from the beginning of data. The other is to use the offset from the end of data (which is equal to the number of bytes remaining). The latter method is appropriate for handling messages since they are always fixed by some finite size. However, it is generally difficult to discover what the total data size is in a streams type of communication. For the streams type, the former method is more suitable. Another possible method is to use an extension identification scheme adapted by ASN.1 [ISO87]. In the presence of multiple Bufds, each Bufd will have the same Buffer identifier with a unique sequence number. The *owner* field specifies who created the Bufd (the semantics of owner are discussed below). The owner identification also uses the two-level (64-bit PID) identification scheme used for the Buffer identification. *ReturnQ*, which corresponds to home node mentioned in Chapter 3, specifies the QID to which the Buffer is supposed to return when the requested operation is complete (the details of the Queue abstraction are left to the next section).

Status fields include *currentQ*, *return_status*, *ref_count* and *more_blocks*. *CurrentQ* indicates the present location of the Buffer. This information is changed as a Buffer moves from one Queue to another. *Return_status* contains the status of the most recent operation on the Buffer. For example, when a Buffer is enqueued onto a Queue and dequeued by the receiving entity, the status of the enqueue operation is set in the *return_status* field, and thus the sender can discover the status of the enqueue operation by dequeueing the returned Buffer and examining the return status in it. Keeping the status with the data rather than with the code complies with the philosophy of the object-oriented design approach. This will support any kind of invocation strategy the implementor wishes to use. *Ref_count* is used to keep track of the number of current references made to a particular object instance. The *more_blocks* flag is used when a user data block in a Buffer is fragmented into mul-

tiple fragments. This flag is set in each Bufd of the fragmented blocks except the last, and the receiver is signalled that the current fragment is a part of a message and more is to come. This flag is used in streams communication as well as other types of communication which involve fragmentation and reassembly.

Static information such as *owner*, *type*, *bid* and *returnQ* will be set as part of the Segmented Buffer object initialization process in the *Segmented_Buffer()* constructor operation. On the other hand, dynamic information such as *q*, *currentQ*, *return_status*, *ref_count* and *more_blocks* will be set and changed as the Buffer object is passed around the system. Here, the details of the constructor and destructor operations defined in the Segmented_Buffer class are briefly discussed. The constructor operation called *Segmented_Buffer()* takes as arguments the address and the size of the data memory the user has created, and initializes the Bufd accordingly. It also takes the home node (returnQ) as a parameter to the Buffer creation operation (the use of returnQ is given in more detail in the next section). It then invokes a kernel routine, which actually creates a copy of the Bufd with the user supplied information in the kernel protected area. This means all Bufds are "shadowed" by the kernel and any updates to shadow copy Bufds require privileged assistance. This would prevent unauthorized accesses to the Buffers. The destruction operation called *~Segmented_Buffer()* is responsible for disposing of the Buffer object, performing such tasks as freeing memory for Bufd information of the object. This is also handled by the kernel which can check the kernel copy of Bufd and make appropriate clean-ups. (This description assumes users are not trusted by the kernel; clearly the assumption can be relaxed in many situations, and different subclasses of Buffer can be created for trusted and untrusted users.)

The five basic operations defined in the Simple_Buffer class are automatically inherited since the Segmented_Buffer class is defined as a subclass of Simple_Buffer.

```
class Segmented_Buffer : Simple_Buffer {
   // linkage
   QDHDR      q;                // doubly-linked Queue pointers
   // identification information
   BTYPE      type;             // Buffer type
   BID        bid;              // Buffer identifier
   int        sequence;         // sequence field for memory block
   PID        owner;            // owner of Bufd
   QID        returnQ;          // return queue
   // status information
   QID        currentQ;         // current queue
   int        ref_count;        // Buffer reference count
   int        return_status;    // Buffer operation status
   FLAG       more_blocks;      // more horizontal Bufds if set
   // data
   Simple_Buffer* databuf;      // data block
 public:
   Segmented_Buffer( BufQ returnQ );   // constructor
   ~Segmented_Buffer();                // destructor
   virtual int write_data( ADDR data, int len );
   virtual int read_data( ADDR data, int len );
   virtual int seek_data( int type, OFFSET off );
   virtual OFFSET get_offset();
   virtual int    get_count();
   // operations for fragmentation and reassembly
   virtual int   fragment_data( OFFSET off );
   virtual int   reassemble_data( Seg_Buf b1, Seg_Buf b2 );
   // operations for setting and getting Bufd fields also go here
   virtual BTYPE get_type();          // get Buffer type
   virtual BID   get_bid();           // get Buffer identifier
   virtual int   get_sequence();      // get sequence field
   virtual PID   get_owner();         // get owner of Bufd
   virtual QID   get_returnQ();       // get return queue
   virtual QID   get_currentQ();      // get current queue
   virtual int   get_ref_count();     // get Buffer reference count
   virtual int   get_return_status(); // get Buffer operation status
   virtual FLAG  is_last_block();     // check if last block
};
```

Figure 4.2: The Segmented Buffer Class Definition

However, they need to be redefined to handle a list of memory blocks. Moreover, the operations to fragment a large data object into two smaller pieces (*fragment_data()*) and to reassemble them (*reassemble_data()*) are required. Given two data fragments, the *reassemble_data()* operation merges the two and creates a single Buffer if they are contiguous. Otherwise, the second Bufd is simply linked to the first Bufd forming a chain. Since numerous additional fields were added to the Bufd, the user interface should also include operations such as *get_returnQ()*, *get_sequence()*, *etc.*

Buffers may be created dynamically, and the creator becomes the owner of the Buffer by virtue of owning the memory blocks it describes. In the Buffer and Queue Model, the owner of the Buffer is the only entity that can destroy the Buffer and recover the memory blocks (except when the owner crashes and the system cleans up on its behalf). This is a reasonable and necessary restriction needed to maintain order in resource management. It is not desirable to allow an arbitrary entity to delete Buffers without the owner's knowledge. If more flexible owner semantics than this are required, one can employ an explicit protection mechanism for Buffer objects. Such a protection mechanism may be used to allow privileged entities other than the owner to delete Buffers. However, this is outside the scope of the current work.

4.1.3 A Recursive Buffer

A simple communication requires at least two communicating entities and data to be transferred. But data used by more than one entity is often padded with control and protocol information. Thus, communication involves transferring not only user data but also control information. Communication protocol headers and trailers are designed to transfer such information. The structure provided by Segmented

Buffer objects is not capable of handling such control information (or cannot handle it efficiently). Data structures and operations are needed, which can handle more complex forms of data than the one described in the Segmented Buffer.

```
class Recursive_Buffer : Segmented_Buffer {
   // a subclass of Segmented_Buffer
   FLAG       linearized;       // send Bufd if this flag set
   FLAG       which_blks;       // indicates which memory blks exist
   // Bufd structure information
   Simple_Buffer* header;       // protocol header
   union struct {
       Recursive_Buffer*  bufdptr; // recursive Bufd pointer
       Simple_Buffer*  databuf; // data buffer
   } nextptr;
   Simple_Buffer* trailer;      // trailer
 public:
   Recursive_Buffer( BufQ returnQ );  // constructor
   ~Recursive_Buffer();               // destructor
   virtual int read_data( ADDR data, int len, int type, int level);
   virtual int write_data( ADDR data, int len, int type, int level);
   FLAG which_blks_exist();     // which memory blks exist?
   FLAG check_linearized();     // check if Bufd linearized
   int  push_blk (addr, len, type) // insert control block
   int  pop_blk (addr, len, type)  // remove control block
};
```

Figure 4.3: The Recursive Buffer Class Definition

A more desirable structure is one that can handle a hierarchy of Bufds, each Bufd capable of containing multiple blocks of memory for header, trailer and data (or another Bufd). Such a structure could handle arbitrary layers of protocol headers and trailers. The Recursive Buffer class definition is given in Figure 4.3. It contains several pointers and flags. The first points to a protocol header field, the second to either data or another Bufd, and the third to a protocol trailer field. The flags,

which_blks, is used to keep track of which memory blocks exist in a Bufd. It also specifies whether this Bufd contains a data block or a pointer to another Bufd; these alternatives are mutually exclusive. The flags can be retrieved by the user invoking the *which_blks_exist()* operation. The *linearized* flag is used to support the Buffer-Queue protocol (the detailed use of the flag and the protocol is presented later in this chapter). To support efficient manipulation of control structures, the operations *push_blk()* and *pop_blk()* are added. The *push_blk()* operation is used during the packetization process to insert header or trailer control blocks by linking the appropriate Bufd pointers to either block. The *pop_blk()* operation, on the other hand, is used as a depacketization process to separate a header or trailer from user data.

It is also useful to be able to read/write data from/to user data or control blocks. This requires that the *read_data()* and *write_data()* operations be modified to traverse through recursive Bufd pointers and access individual blocks. In addition to the usual parameters to these operations, parameters such as the type of memory block and the traversal level will also be necessary (*e.g.*, 0 may mean the current Bufd, 1 may mean the next Bufd, and so on).

As mentioned in Chapter 3, efficient packetization and depacketization can lead to increased performance and saving of resources. The recursive Bufd structure is well-suited to efficient packetization and depacketization. The data can be passed by reference between layers in protocol stack processing, avoiding expensive copying operations. As the recursive Buffer is passed down through a protocol stack, the Bufd of each layer is linked to it, and as it is passed up, the Bufd of the corresponding layer is popped and the rest is passed on. Furthermore, the recursive Bufd structure and functional interface can be used as a "standard" structure for all levels of communication. Since all levels use this common structure, a uniform and simple

software interface can be made for each layer. By appropriate traversal of pointers, data can be available to any intermediate layer module.

A recursive Buffer usually contains multiple Buffers, each of which may have different owners. This is quite common in implementing a stack of protocols (*e.g.*, in network communication implementation). For example, when the IPC layer is implemented by an IPC server, the IPC process will create a Bufd, allocate a header and trailer, and link them to the Bufd. Appropriate information including a new Buffer identifier is filled in the newly created Bufd, and the Bufd that was passed from the above layer is linked to the IPC Bufd. Direct access by unprivileged entities to the header, trailer and user data memory can be prevented since they would have their own protections. Besides, they must access through the Buffer interface, which would also require its own access privilege. Each owner can receive an individual ACK through the return of local Bufds. Even if some intermediate entity fails in the middle of a communication process, error status and Bufds can still be recovered and returned to all appropriate places since each Bufd always contains information regarding the place of return (returnQ). A further discussion of these ACKs will be given in Section 4.3.

As an example, Figure 4.4 shows the resulting Buffer structure at the Ethernet device driver layer, just before being transmitted for an X Window [Sche86] client-server interaction. The content of the Buffer is transmitted using an algorithm which traverses each Bufd and transmits all the headers first, then user data followed by trailers. If the network controller supports the *gather* capability, then copy operations can be avoided completely since the controller can transmit multiple memory fragments directly, including those from the user process. A more detailed discussion of the construction and transmission of such recursive Buffer structures will be given in Chapter 5.

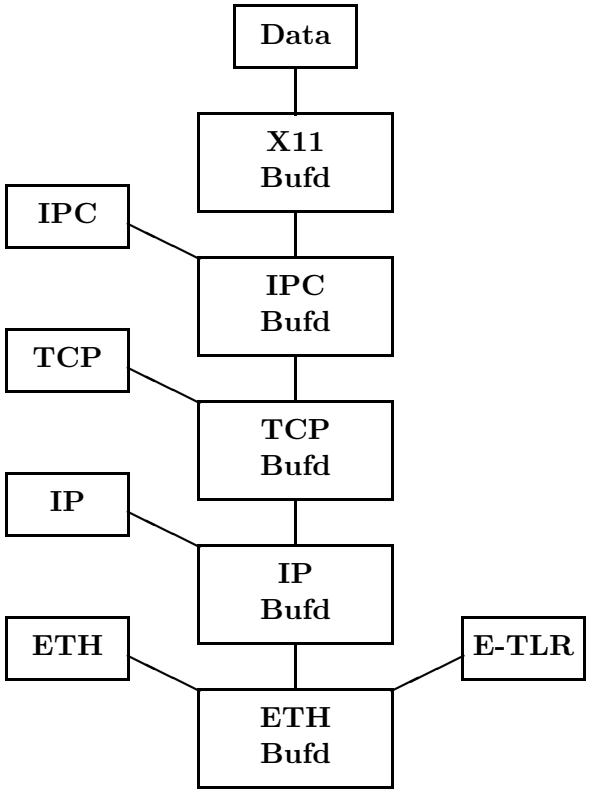

Figure 4.4: An Example of the Recursive Buffer Structure

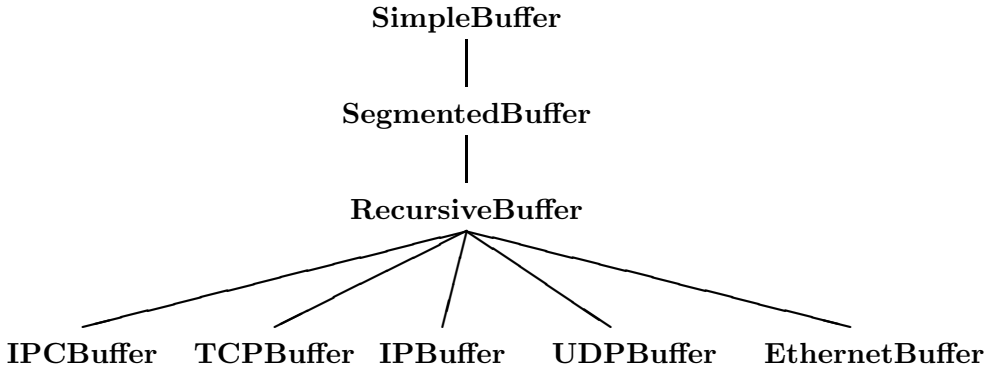

Figure 4.5: The Buffer Hierarchy

4.1.4 The Buffer Hierarchy

The framework that is used to develop hierarchical Buffer structures above can be used to modify existing or develop new Buffer structures as needed. For example, one can create a network Buffer by simply inheriting the features of the recursive Buffer and adding the protocol-specific data and operations. One can create TCP Buffers by adding TCP-specific control data and functional operations to the recursive Buffer. Similarly, one can create IPC Buffers (such as message-passing Buffers, stream Buffers), as well as IP Buffers, UDP Buffers and so on. These different network protocol Buffers all have the generic recursive structure but differ in the protocol-specific data and operations they handle. This Buffer class hierarchy is shown in Figure 4.5.

In this section, the details of the Buffer abstraction and the solutions to various communication problems related to data were presented. Also, the development of an efficient, versatile recursive Buffer structure that all levels of communication can use was presented.

4.2 The Queue Abstraction

In this section, the details of the Queue abstraction and the solutions to various problems related to communication entities or endpoints and their communication interfaces are presented. Also, the development of a universal interface that all communicating entities can use is presented.

4.2.1 A Simple Queue

The conventional concept of the queue abstract data type is used. The simplest queue consists of a pair of pointers, head and tail, and the operations *enqueue* (to add elements) and *dequeue* (to remove elements). The most common convention for adding and removing elements is that they are added to the tail of a queue and are deleted from the head. Furthermore, the first element added is the first element that can be removed (*i.e.*, FIFO). In most cases, such a convention must be used to maintain proper sequence of pieces of data being delivered. Although FIFO is the accepted access discipline in queues, other access modes can be useful in such operations as extracting fragments of a single message from a Queue holding interleaved fragments of a number of messages, as well as handling priority data.

The class definition for the Simple Queue is given in Figure 4.6. For convenience and utility purposes, a counter, *q_cnt*, is maintained for keeping track of the number of data elements in the Queue. This counter is incremented as an element is added and decremented as it is removed. Instead of counting every time a user wants to know the number of data elements in the Queue, this counter can be accessed using the operation *get_q_cnt()*. For efficient access and retrieval, a position pointer, *q_pos*, is also maintained within a Queue. This pointer is set or moved using the

```
class Simple_Queue {            // the simplest Queue
   QDHDR    head;               // doubly-linked queue pointers
   int      q_pos;              // current position in queue
   int      q_cnt;              // number of elements in the queue
 public:
   Simple_Queue();              // constructor
   ~Simple_Queue();             // destructor
   virtual int enqueue();       // add data elements
   virtual int dequeue();       // remove data elements
   virtual int seek_pos();      // move queue position pointer
   virtual int get_q_pos();     // get current position in queue
   virtual int get_q_cnt();     // get no. of elements in the queue
};
```

Figure 4.6: The Simple Queue Class Definition

seek_pos() operation. The pointer can be reset to the head of a Queue, or moved forward or backward by the specified number of data elements. The position pointer can be obtained using the *get_q_pos()* operation. The dequeue operation dequeues the data item the position pointer currently points to. These operations are the fundamental Queue operations that are required (or will be inherited) by subclass Queues.

4.2.2 A Buffer Queue

Queues can store various types of data elements. One can think of queues of memory segments, queues of processes and so on. Since the Buffer and Queue paradigm transfers Buffers among Queues in the system, Queues are needed that can handle various types of Buffers. Such Queues can be obtained by specializing the Simple Queue developed above. The resulting Queue is called a Buffer Queue (or BufQ), and its class definition is given in Figure 4.7.

```
class Buffer_Queue : Simple_Queue {  // Qs for handling Buffers only
   // BufQ identification
   int   type;                // Queue type
   QID   qid;                 // global Queue identifier
   PID   owner;               // owner process of this Queue
   // BufQ status
   int   flags;               // flags
   int   status;              // status
   int   ref_count;           // reference counter
   virtual int signal();      // signal the receiver
 public:
   // operations
   Buffer_Queue();            // constructor
   ~Buffer_Queue();           // destructor
   virtual int enqueue();     // enqueue Buffer
   virtual int dequeue();     // dequeue Buffer
   // operations for getting BufQ fields go here
   virtual int get_type();       // get Queue type
   virtual QID get_qid();        // get qid
   virtual PID get_owner();      // get owner
   virtual int get_flags();      // get flags
   virtual int get_status();     // get status
   virtual int get_ref_count();  // get reference counter
};
```

Figure 4.7: The Buffer Queue Class Definition

In addition to data fields described in Simple Queues, extra information is needed in BufQs for system management and communication, namely identification and status information. Identification fields include *type*, *qid* and *owner*. I envisage that there will be various types of Queues for various types of communicating entities, thus the type of BufQ needs to be specified. The *type* field may also be used to indicate the type of delivery method (*e.g.*, via procedure call, wake-up signal). Note that the type field is used only for information purposes and is not necessary

for Queue operations. As with the Buffer types, the *type* field would not necessarily be required if one uses an object-oriented programming language. The *qid* field uniquely identifies the BufQ across the system under consideration. I propose to use the same two-level identification scheme that was used for the Buffer identification: a 32-bit hostID and a 32-bit localID pair. The *owner* field specifies who created the BufQ.

In the Buffer and Queue communication paradigm, transferring a Buffer is achieved by enqueueing a Buffer onto the BufQ that is associated with the destination entity. Receiving a message is achieved by dequeueing a Buffer from a BufQ. A Buffer may be enqueued on any BufQ in the system. However, dequeue operations are restricted to the owner of a BufQ. This is a reasonable restriction which serves to maintain order in Buffer management. As mentioned in Chapter 3, if more flexible enqueue and dequeue semantics are required, one can build that capability on top of the current semantics.

Status fields include *flags* and *ref_count*. The *flags* can be used for setting various bit options (such as whether the owner is blocked on a dequeue operation of a Buffer object from a Queue). The *ref_count* field indicates the count of Bufds in the system using this particular BufQ as a returnQ. A BufQ can only be properly destroyed if this value is zero. Otherwise, Buffers, which contain memory previously allocated by the owner of the BufQ, can not be returned to the owner and thus can not be freed properly.

Users can obtain the identification and status fields by invoking appropriate 'get' operations that are defined in the Buffer Queue. For example, *ref_count* can be obtained by invoking the *get_ref_count()* operation and so on. The *signal()* operation is used to signal the receiver of the arrival of Buffers, and is a part of the *enqueue* operation (the detailed definition and use of the *signal* function will be

given in the next section).

Thus, the Buffer Queue includes pointers for a doubly-linked queue, status and identification information, and operations for transfer of Buffers as well as to set and obtain BufQ descriptor information.

4.2.3 A Return Queue

Here, the return Queue (or returnQ) concept is introduced, which I believe is unique to the Buffer and Queue model. The returnQ is used extensively for acknowledgements and synchronization (described later) in the Buffer and Queue communication paradigm. The returnQ is a BufQ with no added features. It is associated with each communicating entity mainly for the purpose of retrieving Buffers which the entity has transferred earlier to another entity. Since all Buffers involved in communication must contain the returnQ information as the place of return when the intended operations are complete, the creator of a Buffer supplies a returnQ as a parameter to the Buffer creation operation.

As mentioned in the last section, the Buffer creation operation is a kernel operation in the Buffer and Queue Model (assuming untrusted users). The kernel creates a Bufd with the information supplied by a user. The kernel maintains a descriptor for each Bufd created in the kernel. This is required for garbage collection of Buffer resources left by their owners who may have accidently died without proper clean-ups. A similar situation can occur if a Queue is destroyed and there exist Buffers in the system that have used the Queue as their returnQ. For these situations, the kernel can traverse a list of kernel Bufds and do the proper storage reclamation.

Here, an IPC example is used to demonstrate the use of returnQs, and as a concrete example of the home node concept discussed in Section 3.2.3. A user

process sends a Buffer containing a message to another process. The process either immediately (in the case of blocking IPC) or arbitrarily later in time (in the case of nonblocking IPC) blocks on dequeueing this Buffer from the returnQ. As mentioned earlier, each Buffer object contains the information about where to return the Buffer when the operation is complete. Thus, the return of a Buffer to the returnQ signals the completion of the send operation. The user process is unblocked and the Buffer is retrieved from the returnQ. The user process discovers the status of the operation (*i.e.*, whether the operation was successful or not) from the status contained in the returned Buffer. Furthermore, the return of the Buffer also signals its availability. This encourages efficient resource management.

4.2.4 Network Queues

Network communication usually involves a message being passed through multiple layers of communication protocols. At each layer, protocol-specific processing is performed such as adding header or trailer information and updating state information. Since the header and trailer information and the operations for manipulating them are stored within a Buffer, it is the Buffer that is modified as it travels downward or upward in a protocol stack. However, as discussed in Section 3.2.2, the state information to manage each protocol layer should not be stored in the transient Buffer. In keeping with the object-oriented design philosophy, the state information should be kept not with the protocol-specific code but in a code-independent data structure. There is no better place to put the state information than in the Queues. Thus, a Network Queue is a subclass of Buffer Queue and can potentially include a protocol-specific data structure.

As discussed in the generic communication model in Chapter 3, the network

node objects can have two types of interface for supporting logical pipelines, 'procedural' and 'self-contained objects'. The Network Queue is a *template* class that can support both types of interface. The choice is left to the implementor to define the type of interface he wishes in the protocol-specific object that will be inserted in place of *protocol_descriptor* as shown in Figure 4.8.

```
class Network_Queue : Buffer_Queue { // a subclass of Buffer_Queue
   Protocol protocol_descriptor;     // protocol specific stuff
   virtual int signal();             // signal the receiver
 public:
   Network_Queue();                  // constructor
   ~Network_Queue();                 // destructor
   virtual int enqueue();            // enqueue network Buffers
   virtual int dequeue();            // dequeue network Buffers
};
```

Figure 4.8: The Network Queue Class Definition

4.2.5 The Queue Hierarchy

Thus, a Network Queue is a specialized Buffer Queue that can contain the protocol-specific data, and operations to manipulate them. The framework that is used to develop hierarchical Queue structures above can be used to modify existing or develop new Queue structures as needed. For example, Queues for various network communication protocols (*e.g.*, TCP Queues, IP Queues) can be created by simply inheriting the features of the generic Network Queue and adding the protocol-specific data structures and operations. The Queue class hierarchy containing various Queue classes is shown in Figure 4.9.

In this section, the details of the Queue abstraction and the solutions to various

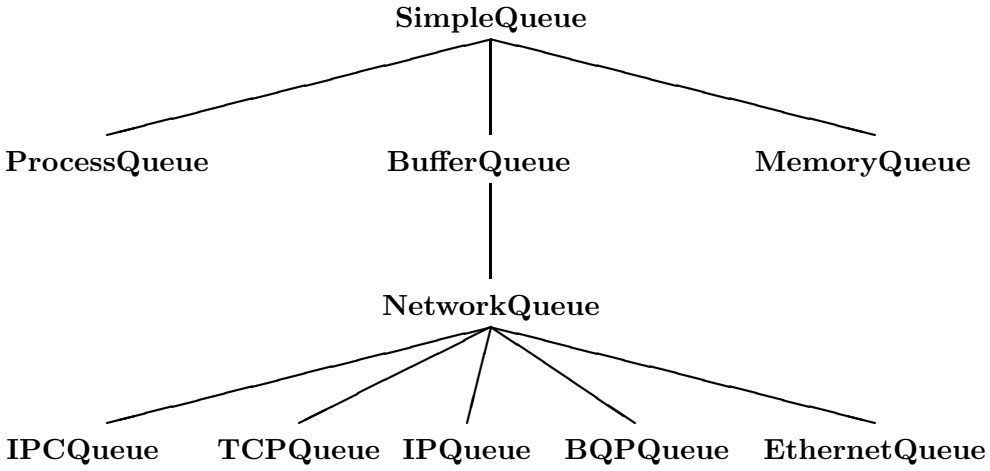

Figure 4.9: The Queue Hierarchy

problems related to communication entities or endpoints and their communications interfaces have been presented. The universal interface (*i.e.*, a Buffer Queue) that has been developed in this section can be used by all types of communicating entities in a wide spectrum of environments.

4.3 The Delivery and Synchronization Abstraction

In this section, the third abstraction in the Buffer and Queue Model is presented, namely the delivery and synchronization abstraction, as well as some solutions to communication problems related to the dynamic functionality of transporting the data.

4.3.1 Delivery

The *enqueue* and *dequeue* operations of Queues are used for delivery of Buffers in the Buffer and Queue paradigm. The enqueue operation is used to transfer the control of the Buffer to the receiver. The dequeue operation is used to accept the transfer of the control from the sender. Unfortunately, the enqueue operation itself does not suffice to deliver the data since, short of continuous polling, the receiver has no way of knowing when the data is available. Therefore, a *signal* function must necessarily be incorporated into the enqueue operation to notify the receiver of the Buffer transfer. Because different entities may wish to be signalled in different ways, the *signal* function is defined as part of the Queue definition by its owner, and its implementation details will usually differ from one type of communication paradigm (or entity) to another. For example, the *signal* function might simply be a call instruction in the procedure call paradigm, which is used both to transfer control and to signal the availability of data, or a wakeup call in the message-passing paradigm to signal a blocked process to resume execution and process new data. The sender (or the entity performing the enqueue operation) should not need to know the details of the signal function defined for any receiver's Queue.

The *signal* function is initially defined in the Buffer Queue class definition in Figure 4.7, and is subsequently redefined in the Network Queue class definition in Figure 4.8. For the moment, it is defined as a private operation, which means it can only be invoked by member functions (*i.e.*, can not be invoked directly by users). The reason for defining the *signal* operation as a private operation is that initially I envisage only invoking it as part of the *enqueue* operation. Thus, the first part of the *enqueue* operation stores the Buffer in the BufQ and then invokes the signal operation, which notifies the receiver. It may be necessary to separate the signal function from the enqueue operation in special cases such as sending

multi-segmented messages and signalling only once at the end. However, this is left for future work.

The Buffer and Queue Model also provides a simple but versatile mechanism for returning data delivery status. The Buffer returnQs coupled with the capability to access nested higher layer Bufds inside a standard Buffer object from any level provide an elegant solution. Any callee can either return the Bufds in the reverse direction of the delivery path (*i.e.*, removing the Bufd it had created and passing the rest to its caller) or return them all directly to appropriate returnQs. Also, even should some intermediate entity fail in the middle of a communication process, error status and Bufds can still be recovered and returned to appropriate places since each Bufd always contains its own returnQ information. Finally, returnQs are also useful for resource management and synchronization as will be discussed below.

4.3.2 Synchronization

In Chapter 3, synchronization in distributed systems was subdivided into *synchronization of user execution* and *synchronization of data-object control*. Recall that synchronization of user execution is concerned with supporting various execution blocking semantics and synchronization of data-object control is concerned with control of data objects (Buffers in the Buffer and Queue communication paradigm). Below, the ways of handling both types of synchronization in the Buffer and Queue Model are discussed.

By convention, a Buffer is blocked (*i.e.*, the user does not have access to it)[2] when it is enqueued on some BufQ other than the sender's or it is in the control of

[2]The user may actually have access to it but the underlying assumption is that it is not safe to modify the contents since the message has not yet been sent or else is needed for some other reason.

some other entity. A Buffer is unblocked when it is dequeued. The return of the Buffer indicates that whichever entity had control of it relinquished it and is done with it. For instance, a message contained in the Buffer may have been copied or sent so that the original Buffer is no longer needed.

In asynchronous or non-blocking IPC, the user initiates a send or receive operation and then resumes its execution. When it wants to discover the status of the operation, it simply checks or blocks on the dequeue operation of the returned Buffer from its returnQ. Recall that the Bufds contain return status and are returned to the appropriate returnQs when the operations are complete. From the sender's point of view, the Buffer is said to be blocked until it is returned to its returnQ. The Buffer will be returned by the recipient as soon as it becomes free (*i.e.*, data is copied or sent).

In synchronous or blocking IPC, a Buffer will not be returned to its returnQ until the message has actually been delivered to the end destination. In the meantime, the initiating user would block attempting to dequeue from the returnQ immediately after initiating the operation.

There are other user blocking semantics: highly synchronous and partially synchronous IPC operations. These can be supported by building on the basic synchronization provided by the Buffers and Queues. Highly synchronous IPC operations can be built on top of synchronous IPC operations. Since the user who initiated a highly synchronous send operation is blocked until the user reply is received, having a sent Bufd returned to its returnQ is not sufficient: two steps are required. The first step is to wait for the return of the original Bufd on its returnQ, which acknowledges the safe delivery of the message to the end destination. The second step is to block on dequeueing a Bufd containing the reply message on its message queue. The first step must succeed before going on to the second step because there

is no point in waiting for the reply message if the request message has not been delivered successfully to the destination.

To reduce communication cost, many message-passing IPC implementations have tried to reduce the number of packet exchanges. Thus, one way to optimize the request-receive-reply interaction has been to suppress the explicit ACK for the request message but include it implicitly in the reply message [Cher84]. In the Buffer and Queue communication paradigm, one can easily achieve this optimization with the help of the Buffer-Queue Protocol (which is presented in the next section), by including two Bufds (ACK and reply) in a single packet. However, the sender must still perform two separate dequeue operations, first to receive the ACK and then to receive the reply message.

Partially synchronous IPC operations can also be built on top of the basic synchronization provided by the Buffers and Queues. Partially synchronous operations require the ability to return an "intermediate" acknowledgement to the sender entity. This, in turn, requires the ability to extract the returnQ information from Bufds. The returnQ information can be easily extracted from any layer in the local node since all layers deal with the standard Buffer structure and know how to access any part of its components. Any layer module can access the returnQ information of its higher layers from the Bufd passed by its higher layer. Thus, it is possible to return intermediate acknowledgements to any higher layer entities. In the simplest case, the Bufd can be returned when it reaches a predetermined point such as the transport layer, network layer or device layer, which acknowledges that the message has reached that layer.

However, the basic Bufd return mechanism allows the Bufd to be returned only once, implying at most a single intermediate acknowledgement. It may be useful to receive more than one intermediate acknowledgement in some applications such

as communication system debugging or mail delivery. For example, an intermediate acknowledgement can return when a message reaches the transport layer and another when it reaches the network layer and so on. However, this would require extra work to be done outside the normal Buffers and Queues communication. The user can be provided with an optional flag indicating the desire to receive multiple acknowledgements. Intermediate modules can then return Bufd duplicates to the sender as the real Bufd passes through, decrementing a counter. When the counter reaches zero, the real Bufd is returned.

The ability to receive local intermediate acknowledgements can be easily extended to remote intermediate acknowledgements as well. That is, intermediate acknowledgements can be received indicating that the message has reached the remote device layer, network layer, transport layer and so on. Such a capability is independent of the number or type of intervening nodes.

The principle of returning the Buffer to its sender when the operation involving the Buffer is complete also provides the basis for Buffer resource management. When the Buffer is no longer needed, the entity that used or held it enqueues the Buffer on its returnQ, where the owner has the responsibility to recover and dispose of or reuse it. Although this mechanism is intended initially to assist Buffer resource management, it can be used as a vehicle for several other useful mechanisms needed in communication. For instance, it can be used to transport acknowledgements back to the requesting entities upon completion or failure of an operation. Directly related to acknowledgements is the synchronization of user execution. Return of acknowledgements can be used to unblock any blocked user entities. Another use is in synchronization of data control. The return of a Buffer signals its availability, as discussed above.

In this section, the delivery and synchronization abstraction of the Buffer and

Queue Model was presented. The semantics of enqueue, dequeue and signal operations were presented, which provide the dynamic functionality of transporting data and synchronizing user execution and control of Buffers. In a distributed environment, the Buffer-Queue protocol discussed in the next section is responsible for transmitting Buffers and synchronizing between sites. In addition, it must also guarantee the return of Buffers to their owners, even in the presence of failures.

4.4 Buffer Queue Protocol

The Buffer and Queue Model presented thus far views a system as a single node, where efficiencies of shared memory can be enjoyed. That is, from the user's point of view, communication takes place within a single global domain. In a distributed Buffer and Queue paradigm, Buffers are transferred among Queues that are scattered throughout a distributed system. In reality, however, there are a number of separate nodes and protection domains in such a system and some tools to bridge this gap are necessary if a truly transparent distributed system is to be provided.

Local communication which passes Buffers by reference can easily deliver arbitrarily complex local Buffers to any Queue within a single protection domain without losing any structural information. Thus, the structural information of Buffers as well as other control information to make the distributed Buffer and Queue operations transparent should be somehow delivered along with the content. This contrasts with conventional network communication which is designed to deliver only the content but not the structure of data. The Buffer-Queue protocol (BQP) is intended to do just this.

The BQP, thus, is concerned with two aspects of distributed Buffer and Queue communication: remote Buffer and Queue operations and the description and en-

coding of data. The BQP peer modules handle and synchronize remote operations and ensure that copies of updated fields are passed back and forth as required. Note that for the current version of the Buffer and Queue Model, only remote enqueue and return operations are allowed. This restriction is a side effect of one of the general principles, which allows only the owner of the Queue to dequeue Buffers from its Queue. However, if Buffers are allowed to be dequeued by arbitrary entities then remote dequeue operations would also be needed. This would also require implementing higher level access control mechanisms, and modifying the current principles drastically.

The data structure encoding (called *linearization*) as opposed to the control aspects of BQP could easily be replaced by external data representation standards such as XDR [SUN87] or ISO ASN.1 [ISO87] at the possible expense of some efficiency. Besides linearizing the Buffer on the sending side and delinearizing on the receiving side, BQP provides remote routing through Queue identifiers, and if not handled by lower-level protocols it can also provide reliability (*e.g.*, error recovery, sequencing), fragmentation/reassembly and multiplexing.

4.4.1 Transmission of Structural Information

In a layered communication system, an implementation of BQP would result in a separate, thin layer in a stack of protocol layers. One possible placement of the BQP layer in a typical system is shown in Figure 4.10. However, it should be emphasized that the BQP layer can be placed at an arbitrary level as will be demonstrated in implementation examples in Chapter 5.

In the Buffer and Queue Model, the structural information of a Buffer is contained in a Bufd or nested Bufds of the Buffer. Thus, one essential aspect of the

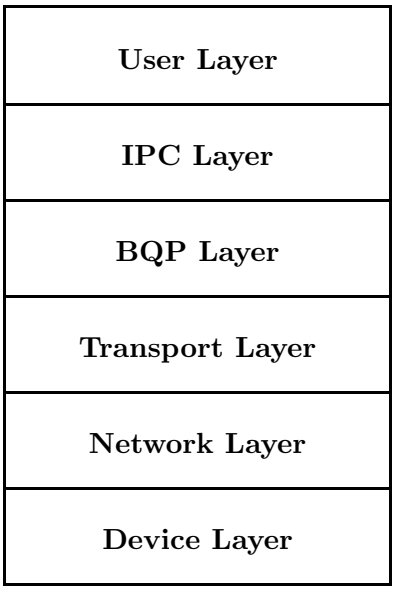

Figure 4.10: The BQP Layer in a Layered Communication Model

BQP is that it transfers structural information contained in Bufds above the BQP layer to the remote node. The BQP layer on the receiving side uses this information to reconstruct the original Buffer, so that a message with its internal structure intact can be delivered within the remote node as if done locally. A Bufd also contains the Buffer and Queue operational information such as the Buffer return address (returnQ) and status of Bufds as the requested operation completes. This information must also be transferred along with the structural information. These two types of information (structural and B-Q operational) are referred to as *essential* Bufd information that needs to be transferred in order to support the shared-memory communication paradigm across the system.

Before discussing how to deliver the essential Bufd information to remote nodes, what to transfer must be decided since the Bufds contain more than just the struc-

tural and operational information. They also contain information that is useless in remote nodes (*e.g.*, local virtual memory addresses), which need not be transmitted. There are roughly two ways to transfer the essential Bufd information. One is to extract and transmit only the essential information from each Bufd. The other is to transmit the entire Bufd. Both have advantages and disadvantages. Although the second approach uses some additional bandwidth by transferring additional data, I believe the second approach provides a better choice for its simplicity and uniformity. The main disadvantages of the first approach are the complexity involved and its susceptibility to changes in Bufd fields in the future. These problems of course vanish if one has already paid the price of using XDR or ASN.1 encoding. The second approach puts less burden on the receiver as well, as the receiver need not re-allocate any extra memory for Bufds but may simply strip them from the inbound packet and overwrite fields to make them into appropriate Bufds. This assumes that no Bufd straddles a packet boundary which is a reasonable assumption since I believe in most cases the headers and Bufds will fit in a single network packet. If they do not fit in a single packet, then the receiver must ensure that the Bufd fragments are reassembled as a contiguous memory allocation. Should Bufd fields change in the future, one can simply change the Bufd header and recompile across the system without any changes in the code since all Bufd structures are in effect treated as simple Buffers during communication.

There is still another major problem that must be overcome, which is to transfer the higher-layer Buffer structure (*i.e.*, those Bufds above the BQP layer) to remote nodes. In normal network communication, the transmission mechanism such as an Ethernet device driver sends only the headers, data and trailers but not the structural relationships among them specified by the Bufds themselves. I have thought of three possible ways to accomplish this task. One is to modify the

Bufd structure above the BQP layer in such a way that the higher-layer Bufds, headers, and trailers all appear as data blocks. In this scenario, the Bufds would be transmitted as the contents of a Buffer. The other two methods do not modify the Bufd structure but instead modify the read operation's traversal method slightly in the device layer to include these higher-layer Bufds, headers and trailers. Below all three schemes are described followed by my preferred choice.

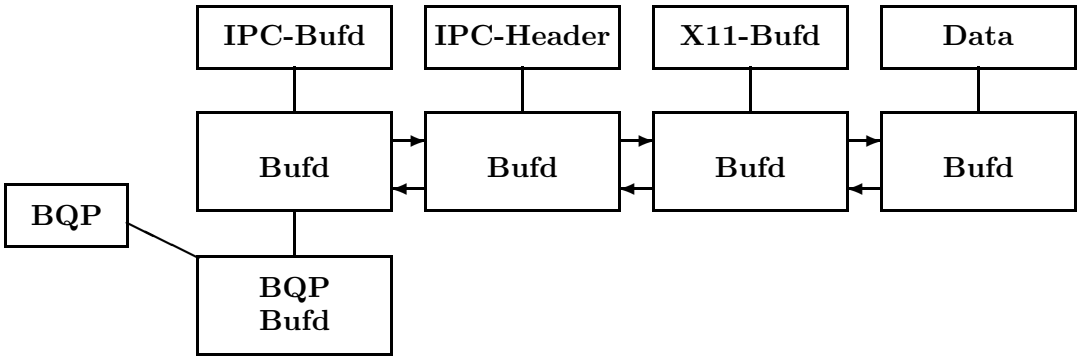

Figure 4.11: Physically Linearized Higher Layer Bufds

The term *linearization* is used to describe the process of preparing the Bufds as well as the header, trailer and user data for the purpose of transmission. In the first scheme, linearization requires physical structural changes to the Bufd above the BQP layer. In order for all memory blocks (*i.e.*, Bufds, headers, trailers and data) above the BQP layer to be transmitted as data, the Bufd structure is flattened with each memory block described by its own additional Bufd (except the one for the user data memory block, which already has one). The resulting Bufd structure above the BQP layer consists of a list of Bufd-header-trailer triples for each layer followed by the user data. For example, the linearized Bufd structure for Figure 4.4 above the transport layer would look like Figure 4.11 before it is passed to lower

layers (assuming that the BQP layer is located between the IPC and transport layers).

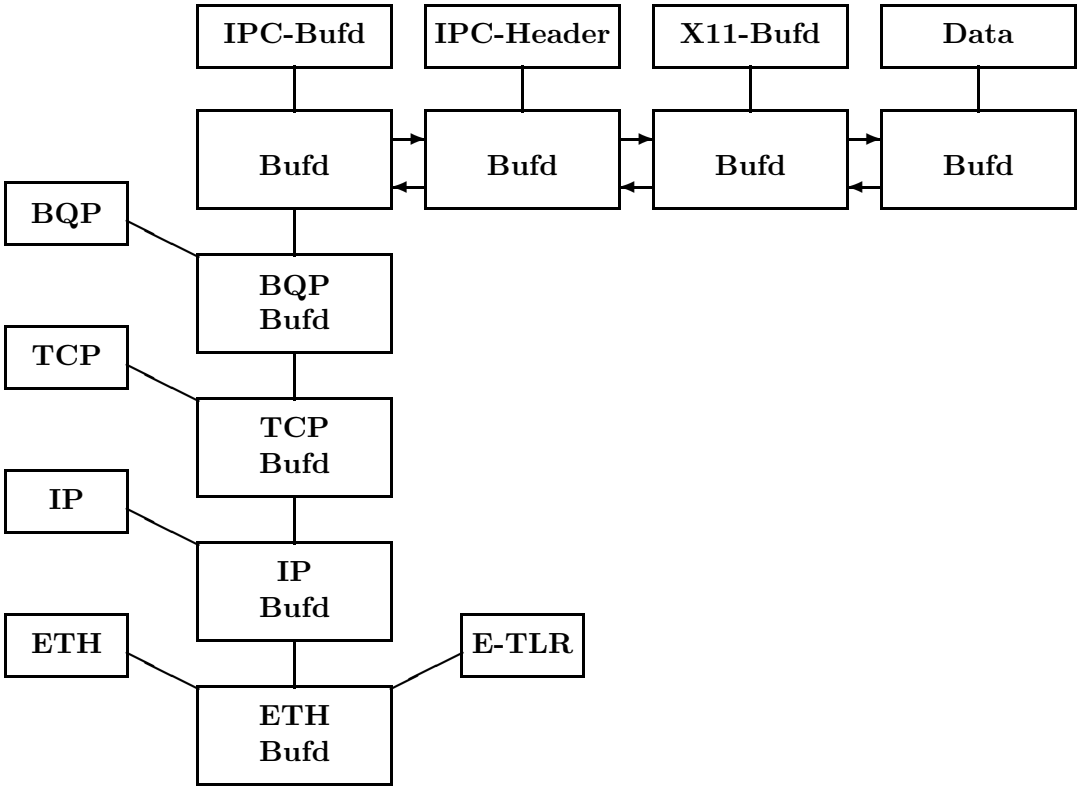

Figure 4.12: An Ethernet Buffer with Physically Linearized Bufds Above the BQP Layer

The BQP-Bufd points to the BQP header and a list of Bufds (or segmented Bufd) that contain the Bufds, headers, and trailers of the higher-layer protocols as well as the user data. The actual data seen by the transport layer is the BQP header followed by IPC-Bufd, IPC-Header, X11-Bufd and user data. The BQP header, therefore, must contain enough information to be able to reconstruct the above structure on reception. However, what is missing in Figure 4.4 is the BQP

layer. When the BQP layer is inserted, the entire Buffer structure would look like Figure 4.12. When the original traversal scheme described in Section 4.1.3 is employed on this physically linearized Bufd structure, the actual transmitted Ethernet frame would be as shown in Figure 4.13.

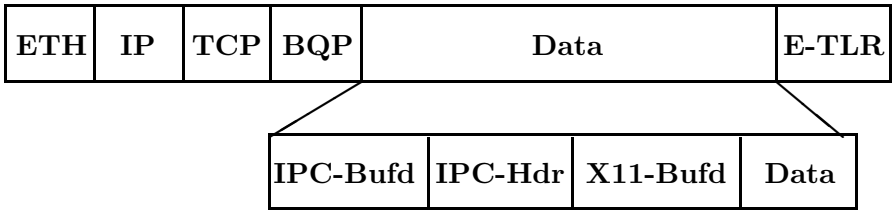

Figure 4.13: Actual Transmitted Data in an Ethernet Frame

The second and third linearization schemes involve no physical modification of the Bufd structure, but rather change the way a device layer traverses the Bufd structure for transmitting various memory blocks. That is, the Bufd structure is not modified as done in Figure 4.12 in the first scheme but rather the read operation is modified. In this scheme, the original Bufd structure is maintained with the BQP layer inserted as shown in Figure 4.14, with each higher-layer Bufd marked as "send this Bufd as a part of Buffer content." A flag is simply used in the Bufd (*i.e.*, *linearized* flag bit in the Buffer definition) to mark it as such. This flag can be set when higher-layer Bufds are created (when the layers already know that the Bufds they create will be transmitted as part of a linearized Bufd structure), or the BQP layer can simply traverse the Bufd structure and set the flag in each Bufd as a part of the linearization process. In either case, the flag in each of the higher-layer Bufds will be set so that the Bufds themselves are transmitted as data.

The main difference between the second and third schemes is in the way they traverse the Buffer structure for transmitting the content and Bufds. In the original

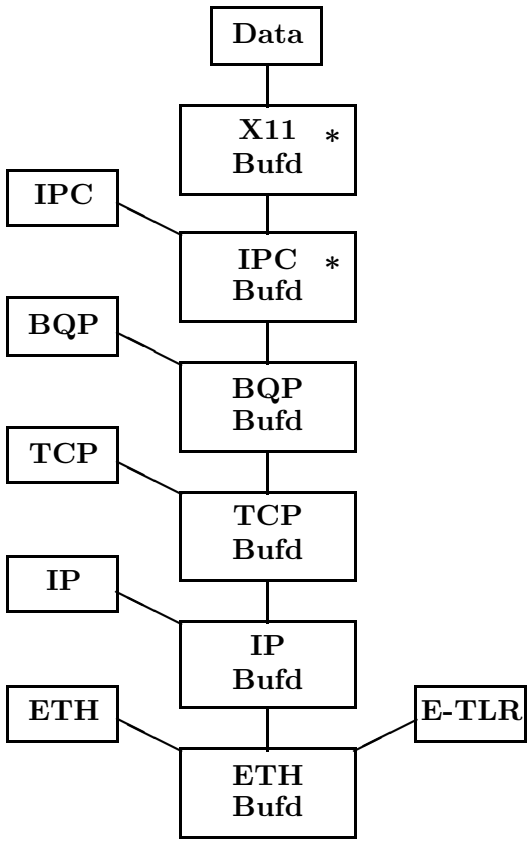

Figure 4.14: The Linearized Buffer Structure for the Second and Third Linearization Schemes – asterisks in Bufds indicate *linearized* flag set

traversal scheme, the Bufd structure is simply traversed from bottom to top, taking all headers, the user data and then all the trailers. Here, the device driver simply checks the flag in each Bufd as it traverses from bottom to top and takes headers only if the flag is not set, then whatever is in the data field, and then trailers. If the flag is set, it takes the Bufd itself, then its header and then the trailer, and goes on to the data. So, from Figure 4.14 the Ethernet device driver would check Ethernet, IP, TCP and BQP Bufds and find the flags not set so it would first take their headers: Eth-header, IP-header, TCP-header, and BQP-header. When it checks IPC-Bufd, it will find the flag set, so it would take IPC-Bufd, IPC-header (and IPC-trailer if it existed), and go on to the next Bufd, which is an X11-Bufd. It also finds the flag in the X11-Bufd set, so it takes X11-Bufd. The next layer only includes the data memory block so it simply takes it. This process would then check for BQP, TCP, IP and Ethernet trailers in that order, and succeeding with Ethernet. Overall, the resulting packet is equivalent to the one created by the first scheme, and is shown in Figure 4.13.

The third scheme also checks the *linearized* flag in each Bufd as in the second scheme. However, the difference is that the third scheme transmits all the linearized Bufds first, then headers, trailers and data, as opposed to the second's Bufd-header-trailer-data. So the third scheme transmits all the headers of non-linearized layers, then all the Bufds of linearized layers, headers, trailers and data of linearized layers and finally the trailers of non-linearized layers.

The second and third schemes are preferred over the first one for efficiency in memory space and time. The first scheme requires more memory space for extra Bufds, as well as extra complexity for rearranging the Bufd structure. The second and third schemes require only a minor modification in the traversal algorithm, which involves checking an extra field in each Bufd. However, the idea of transmit-

ting all the Bufds and headers first as in the third scheme should benefit in long message reassembly operations since the receiver would then receive all the headers first and extract whatever information necessary for reconstructing the original Buffer structure before data arrives in communications involving long messages.

Because BQP transmits Bufd fields directly, it must be concerned with low-level details such as byte ordering, if it is to be used in a heterogeneous environment. There are two approaches that can be taken to support such heterogeneity. One is to convert the BQP data to "network order" at the source and convert back at the destination if necessary. The other is to set a flag in the BQP header to denote the type of byte ordering the source data is in and convert at the destination BQP layer only if it uses the different byte ordering. The latter approach is better since it will perform a *lazy* conversion (*i.e.*, convert only when it is necessary to do so).

4.4.2 The BQP Header

What has been shown so far is how higher-layer Bufds can be delivered to achieve transparency. However, the information needed in the BQP header to support remote Buffer and Queue operations transparently has not been discussed. A Buffer object potentially consists of a number of memory blocks: Bufds, headers, trailers and data. One or more of these memory blocks may exist in a layer. For example, the user layer may only have data and a Bufd for it. The IPC layer may have a Bufd, IPC-header and a user Bufd and user data. Another possibility is a list of Bufd, header, Bufd, data1, Bufd, data2, ... Bufd and dataN. There are of course other possibilities. In order to be able to reconstruct the original Buffer structure, one needs to know exactly what memory blocks exist at each layer and their individual lengths as well as the total number of Bufds. The first two pieces of information are

already available in each Bufd, thus transmitting Bufds would be sufficient. The total number of Bufds, and the total message length need to be specified in the BQP header. Above the BQP layer, the order of memory blocks used in packaging will be Bufd, header, trailer followed by data if the second traversal algorithm is used, or all Bufds, headers, trailers and data if the third traversal algorithm is used. The BQP header, shown in Figure 4.15 will also need to specify how many layers it has linearized; this information is especially necessary when more than one Bufd exists in a layer (*e.g.*, multiple fragmented data memory blocks and their Bufds in user layer).

Source QID	**Destination QID**
Type	**Flags**
More	**Layer Count**
Sequence	**Checksum**
Total Message Length	**Number of Bufds**
argument memory blocks go here . . . . . **. . . . .**	

Figure 4.15: BQP Header Format

What other pieces of information are required in the BQP header? Since communication based on Buffers and Queues involves transferring Buffers among various

Queues in the system, Queue identifiers (QIDs) are used as addresses of endpoints. Thus, *Source QID* indicates the source address and *Destination QID* indicates the destination address. The *Type* field indicates the type of remote Buffer and Queue operation. The *Flags* field is used for setting various flags. The *More* flag is used when a BQP message is fragmented into a number of small messages. It indicates that this message fragment is not the last. The *Sequence* field is used for numbering fragmented messages. It is required for checking that fragmented messages are received correctly as well as for retransmission in case of errors. For checking the correctness of transmission, a *Checksum* field is also required. Two popular approaches of calculating checksum are 1) to do the checksum of the BQP header only and 2) to do the checksum of the entire BQP packet. Both approaches are allowed. The choice is made by the implementor by setting a flag bit in the *Flags* field. The choice usually depends on the type of underlying service. For example, if the underlying protocol provides a reliable service, then it is probably not necessary to do the checksum of the entire BQP packet. On the other hand, if the user of BQP requires a reliable service which is not provided by the underlying protocol, then the choice is more likely to do the checksum of the entire BQP packet. *Layer Count* indicates the number of layers linearized above the BQP layer. *Total Message Length* specifies the total message size (*i.e.*, header plus data) and *Number of Bufds* specifies the number of linearized Bufds above the BQP layer.

The BQP peer-to-peer communication is basically a transaction-oriented, RPC-like interaction. A BQP entity sends a BQP message to another BQP server requesting an action (*e.g.*, enqueue a Buffer, or abort) and waits for a reply. In the most common case of the enqueue operation, the reply occurs when the Buffer in question is returned, that is, when an entity at the remote node performs what it considers a remote enqueue operation on the returnQ of a remote Buffer.

4.4.3 The BQP State Machine

The BQP protocol requires end-to-end sequenced reliability. Internal mechanism is provided to do this in order to provide flexibility in the placement of the BQP in a protocol stack. If the reliability aspect had not been included in the BQP design, it would always have to be implemented on top of a reliable protocol in order to perform any remote operation. If the BQP layer is placed on top of a reliable protocol then the implementor has the option to ignore this aspect. The BQP is designed to be a state-driven protocol which contains explicit definition of state and transition actions. The BQP is designed based on the reliable message exchange protocol that has been implemented and tested on Shoshin [Toku83]. However, the protocol itself is independent of Buffer and Queue functions. The BQP yields a well-defined interface for sequenced reliable remote operations. The protocol can support multiple transactions by using a separate state machine for each transaction. If entities involved in a transaction goes out of synchronization, the protocol can easily detect and abort the transaction. Figure 4.16 illustrates the possible states, events and actions for various events The diagram actually does not show all possible actions but only the important ones. A full list of events and actions is given in Appendix C. Ellipses in the diagram represent the current state, numerators represent events, and denominators represent actions.

Initially, the entities (*i.e.*, the users of the protocol) are in the null or idle state (IDLE). When the receiving module receives a sender data packet (S_SDD), it creates a state record for a new transaction (R_INIT). A state record is basically a BQP header with the available information filled in and modified as the transaction progresses. If the request was contained within a single packet, then the receiver goes into the waiting-reply-from-user state (R_REP) immediately. If the request packet consists of multiple packets (*i.e.*, more packets to follow), then the receiver

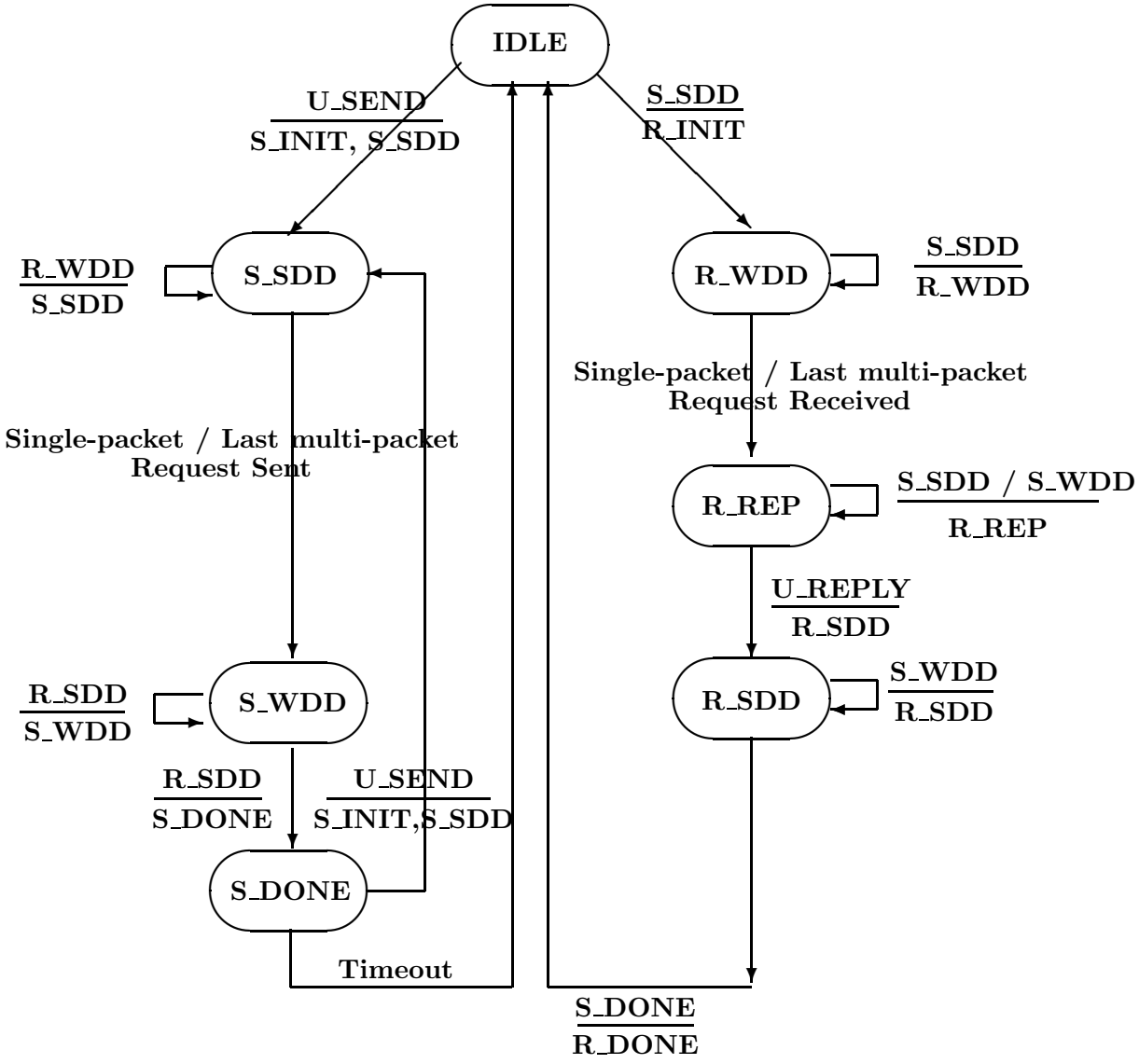

Figure 4.16: The BQP State Transition Diagram

goes into the waiting-data-from-sender state (R_WDD) to receive the remaining packets. When the last data packet is received, the receiver goes into the R_REP state and waits for the user reply, that is, the return of the Buffer. When the receiver module receives a user reply (U_REPLY), it sends the reply to the sender and goes into the receiver-sending-data (R_SDD) state. If the reply contains more than one packet, it keeps sending and remains in the R_SDD state. The receiver then waits for an acknowledgement from the sender indicating it has received the reply correctly. If the acknowledgement is not received within a timeout period, then the receiver retransmits the last packet sent. This will handle any lost packets. When the receiver receives the final acknowledgement packet (S_DONE), it cleans up the state record for the transaction and goes immediately back to the idle state.

When the sender entity is in the null state and receives a send request from the user (U_SEND), it first creates a state record for a new transaction. It then sends a request packet (S_SDD) to the receiver. If the request packet is the first of many, then it goes into the sender-sending-multiple-packet-request (S_SDD) and remains in that state while transmitting those data packets. When the sender finishes sending the last data packet or if the initial packet was the only packet, it goes into the S_WDD state waiting for the reply. In general, there is always a short timeout mechanism to resend unacknowledged packets. The appropriate ACK is always to send the last packet plus the current state. If the receiver module receives a duplicate packet while waiting for the reply from its user, it will simply reply with a R_REP packet indicating so. When the sender receives the reply, it acknowledges with the S_DONE packet and goes into the S_DONE state. In the case of a multi-packet reply, it acknowledges incoming packets and remains in the S_WDD state until it has received all reply data packets.

Timeout mechanisms are used to retransmit packets and to check whether re-

mote nodes are alive. A short timeout mechanism is used by a BQP server entity when it initiates a packet (or multi-packet) transmission and is waiting for an ack (or acks) from the remote peer. For example, when the local server entity transmits a multi-packet request in the S_SDD state, it expects acks to arrive from the remote peer after a short period of time (*e.g.*, after several packet transmission time). On the other hand, when it goes into the S_WDD state (waiting for a user reply which is initiated from the remote node), it uses a long timeout mechanism. In that state, if the reply is not received and a timeout expires, it sends a "ping" message to the remote node to check whether it is still alive. Thus, if timeouts expire but the remote node does not respond or if the transaction goes out of synchronization, the transaction will abort. When the long timeout expires in the S_DONE state, however, the sender assumes the receiver received the final acknowledgement correctly, cleans up the state record for the transaction, and returns to the idle state. While in the S_DONE state, if the sender receives a send request (U_SEND) from the user, it creates a new state record for a new transaction, sends the request packet and goes to the S_SDD state just as if it were idle.

There also exist some internal actions for the BQP entities. The R_DONE action is initiated when the receiver entity receives the S_DONE acknowledgement packet from the sender entity after sending the user reply. The R_ACK action is used to acknowledge multiple data packets from the sender entity. The R_INIT and S_INIT actions are used to create state records for the receiver and sender entities respectively. P_IGN is used to drop or ignore useless packets. P_CLR is used to abort and clear transaction when communicating entities lose synchronization, and resets the state machine. When an unexpected packet arrives, it usually generates an error (P_ERR). In this case, an appropriate message is generated and the transaction is aborted. A complete data table is given in Appendix C.

In this chapter, the Buffer and Queue Communication Model is presented, which satisfies the constraints of the generic communication model of Chapter 3. The Buffer and Queue Model is a simple, low-level but powerful and efficient communication model for distributed systems. The reliable Buffer-Queue Protocol has been developed to support transparent distributed Buffer and Queue operations. Furthermore, the BQP can be used to transfer long messages efficiently across multiple nodes, to reduce the number of packet exchanges in transaction-oriented remote interactions, to support efficient bulk data transfer, and to support distributed shared memory as will be shown in the next chapter.

Note that the current version of the BQP does not support "forwarding" of Buffers. That is, when a Buffer is sent to a remote entity, which then in turn sends it to another remote entity and so on, the Buffer must be returned through the exact path it took in the reverse direction. Returning the Buffer directly to its original entity from the last entity to which the Buffer was forwarded requires some extra work among the parties involved. One simple solution is to notify the original sender of the fact that the intermediate entity is forwarding the Buffer to a third entity and to close the transaction with the original sender. The original sender would now expect the transaction to continue between the entity returning the Buffer and itself. There are other simple solutions as well. Supporting such capability may be useful or allow more efficient communication in some applications, and I envision such extension as a trivial task. However, the extension is left for the future work.

In summary, I claim that the Buffer and Queue Model is able to support a wide range of communication requirements between diverse types of entities both within a single node and across a distributed system. The claim is substantiated by the examples presented in the next chapter.

Chapter 5

Examples Using Buffers and Queues

In Chapter 4, the Buffer and Queue Model was presented. In this chapter, a number of examples of how Buffers and Queues can be used in different types of communication paradigms and communication related problems is presented. First, several examples of simple internal communication are shown, all within a single protection domain. Internal communication is elaborated and used to present local interprocess communication using Buffers and Queues. An example of network communication involving multiple communication protocol layers is presented. Combining the above, a simple example of distributed communication which involves transferring short messages is presented. Then an example of distributed communication which involves transferring long messages is presented. The last two examples demonstrate how the BQP layer is used to provide location transparency. Optimistic blast protocols for high throughput bulk data transfer proposed by Carter and Zwaenepoel [Cart89] and O'Malley *et al* [OMal90] are examined, and a sim-

pler solution based on Buffers and Queues is presented. Examples of implementing distributed and conventional semaphores [Dijk65], one-way communication (*e.g.*, broadcast, multicast), and distributed shared memory [Li86] are also presented.

5.1 Internal Communication

As stated earlier, *internal communication* refers to communication between entities within a single protection domain, where shared memory is provided. It is a communication environment where messages are passed by reference and copying of messages is avoided as much as possible. The internal communication example, shown in Figure 5.1, involves two entities, A and B. A receiveQ associated with the entity B is a BufQ onto which incoming Buffers are enqueued and where they remain until dequeued by the owner. A returnQ associated with the entity A is also a BufQ, to which Buffers are returned after completing their journey to one or more communication endpoints.

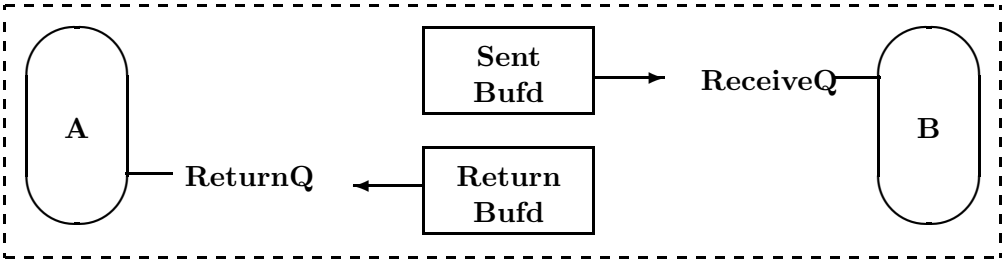

Figure 5.1: An Example of Internal Communication

Entity A transfers a Buffer to entity B by enqueueing it onto B's receiveQ, and invoking its *signal* function as a side effect. Entity B retrieves the Buffer by dequeueing it from its receiveQ. The status of the Buffer transfer is recorded in the

return_status field of the Bufd, and it is enqueued to the source's (A's) returnQ by B. Entity A discovers the status of the data transfer by dequeueing the returned Buffer from its returnQ and examining the *return_status* flag.

Note that so far, the direction of data flow has not been mentioned, but only that of Buffer flow. The direction of data flow and the direction of Buffer flow are orthogonal in the Buffer and Queue Model. That is, entity A can send data to entity B by transferring a full Buffer to B's receiveQ, and entity A can request data from entity B by transferring an empty Buffer to B's receiveQ. In the former case, the empty Buffer along with the status will be returned to A's returnQ. In the latter case, the filled Buffer along with the status will be returned. It is also possible to generate bidirectional data flow by having the returning Buffer filled with B's user data. When a Buffer is dequeued from a returnQ, it is assumed to be safe to reuse it or destroy it.

The simple internal communication example described above can be directly applied to communication between threads (in the Mach [Acce86] sense) within a single protection domain. In this case, the signal function unblocks a blocked process. In the case of coroutines, signal is simply the coroutine *resume* operation. If the two entities are required to communicate by procedure call, the enqueue and signal functions become asymmetric and more complex.

Consider a procedure A (the calling procedure) invoking a procedure B. Assume that there are some parameters transmitted by A to B and results returned from B to A. Further, assume that the process of saving and restoring the context of the calling procedure will be handled by the normal procedure call mechanism, so that the description can be focused on the communication aspect of procedure call. The calling procedure "prepares" the input parameters by filling in appropriate information in the Buffer to be sent to the called procedure. Those parameters

passed by value are copied into the Buffer, and those passed by reference will have their addresses copied into it. The prepared Buffer is enqueued to the receiveQ of the called procedure. The signal function is then invoked as part of the enqueue operation to switch the thread of execution from the caller's context to the callee's context. The signal function invokes the dequeue operation of the callee's receiveQ to access the input parameters (*i.e.*, put them on callee's stack), before finally calling procedure B. After the execution of B, the result is appropriately stored in the Buffer again and returned to the called procedure by enqueueing it to the caller's returnQ. As in the calling sequence, the signal function would be executed and the result is put on the caller's stack. At this point, the normal procedure return mechanism causes execution to return to the next statement in A.

Note that in the Buffer and Queue paradigm, entity A does not need to "know" that B is accessed by a procedure call, and B does not need to "know" that it is being called through the Buffer and Queue interface. Furthermore, the Buffer and Queue interface which encloses B does not need to know details of the implementation of A's returnQ.

This example of procedure call communication can be extended trivially to remote procedure call by performing the enqueue operations on remote Queues. Any failure of the remote system would be detected by the local BQP server, who would return the Buffer with an appropriate error status, which would be received at the time of the caller's dequeue operation.

These simple uses of Buffers and Queues for internal communication are used as a building block in the subsequent examples.

5.2 Local Interprocess Communication

Next, an example of interprocess communication between two local processes or entities that do not share an address space is presented. Since the communicating entities do not share a common area of memory, data transfer involves a copy operation from the sender's address space to the receiver's. The copy operation is usually carried out by a privileged third party such as a kernel, which also acts as an agent to synchronize the transfer, and which has access to both address spaces.

A process, A, has some data to transfer to another process, B. Process A obtains an IPC Buffer by either creating a new one or reusing an existing one. If a new IPC Buffer is required, A would make a kernel request. The kernel would then create an IPC Buffer and return a user copy to A. Since the sender has the data and knows to whom it wishes to send, it inserts the pertinent control information in the appropriate fields of the IPC header and links the data memory block to the IPC Buffer. Process A then passes the IPC Buffer to the IPC layer entity or IPC server (*e.g.*, the *PostMaster* in Shoshin [Shew90]) by enqueuing the IPC Buffer to the IPC server's BufQ as shown in Figure 5.2. Process B, which wishes to receive some data from process A also "prepares" an IPC Buffer with an empty data block and passes it to the server. The IPC server is equipped with two BufQs: *sendQ* and *receiveQ*. A sendQ is a BufQ, where the Buffers that contain data to be sent are enqueued, and a receiveQ is also a BufQ, where the Buffers that contain empty data blocks to be filled with incoming data are enqueued.

The IPC server now has the sender's Buffer on its sendQ and the receiver's Buffer in its receiveQ. The IPC server matches the sender and receiver by examining the source and destination fields of the IPC headers (any specific matching algorithm is not discussed here since it is a function of the IPC protocol and outside the scope

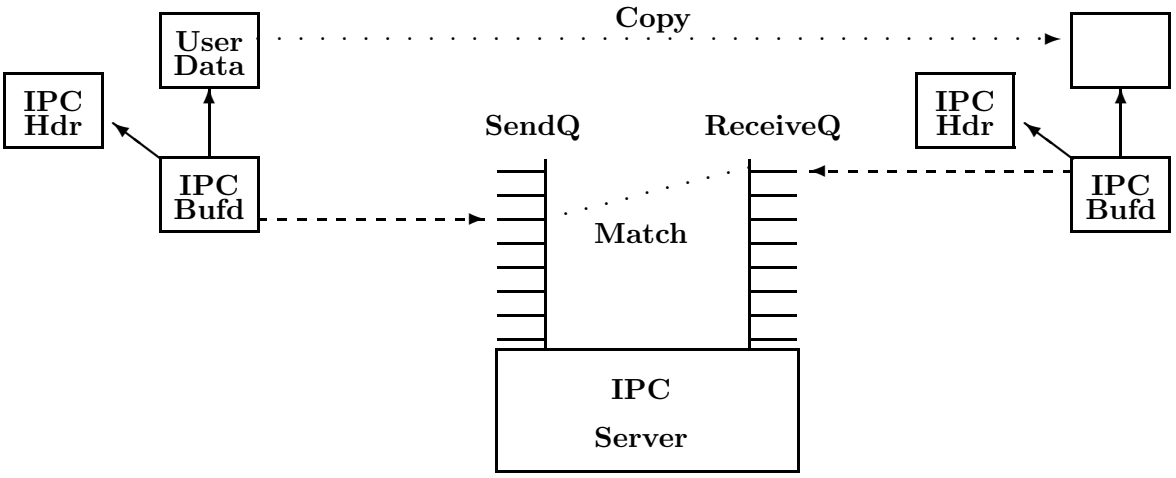

Figure 5.2: An Example of Local Interprocess Communication

of this work) and copies the data from process A's Buffer to process B's Buffer. At this time, the IPC server records the status of the data transfer by setting the *return_status* field in each Buffer. It also records the number of bytes that have been actually transferred to the receiver's Buffer. The IPC server then returns A's Buffer to A's returnQ and B's Buffer to B's returnQ. Process A obtains the status of the data transfer by dequeueing the Buffer from its returnQ and examining the *return_status* field. Process B, on the other hand, can retrieve the received data by dequeueing the returned Buffer from its returnQ.

Note that the user memory provided by the sender or receiver does not necessarily have to be a single contiguous block. It can be in other forms such as a scatter-gather list of memory fragments. It merely needs to be an acceptable Buffer object. The Bufd (or Bufds) in a Buffer contain all the necessary information pertinent to the composition of memory blocks. Thus, a novel feature of the Buffer and Queue Model is that the Buffer structure of the sender and that of the receiver

do not need to be the same. This feature was used to show the equivalence of message-passing and streams through the intermediate use of Buffers and Queues. Basically, the sender has a message-passing user interface (*i.e.*, a MP Buffer) and the receiver has a streams user interface (*i.e.*, a streams Buffer). A sender's message is received by the receiver in small chunks as desired, and the end of message signals the end of stream. Thus, user interface and basic communication are orthogonal components in the Buffer and Queue Model.

Note that in streams IPC, several Buffers may actually be involved at one end to fill or empty a single one at the other. A stream is closed by setting the *more* flag to false in the last Bufd, which forces both sides to terminate in an appropriate manner. Also, privileged processes, which have access to shared Buffer memory, need not copy data into their own memory areas. They can simply arrange to have Buffers forwarded directly to their own BufQs, effectively turning the IPC server into a simple router. They would have the option of returning the Buffers themselves, or of returning them through the IPC server. Obviously, many systems may choose to encapsulate these Buffer and Queue implementations of IPC in library or kernel routines for the convenience of user processes.

5.3 Network Communication

Next, an X11 [Sche88] client-server communication example is used to demonstrate how Buffers and Queues can be used for efficient conventional network protocol processing. An X11 client, which wishes to send a request to a server located on another node on the network, creates an X11 Buffer and fills it with the appropriate information such as the request op-code and its arguments. It then invokes an appropriate X Toolkit [Swic88] or X library routine. Recall that in the hierarchy

of X11 communication protocols, there is a thin layer of IPC. X11 uses sockets in Berkeley UNIX and STREAMS in System V UNIX as its underlying IPC mechanism. However, they are similar in that both layers are responsible for connection and addressing. In this example, the use of a stack of TCP, IP and Ethernet protocols under the IPC layer is also assumed.

Presumably, the invoked X library routine would request that the kernel create an IPC Buffer. The kernel then would create an IPC Buffer, and return a user copy to the X library routine. The X library routine "prepares" it as process A did in the local IPC example above and passes the IPC Buffer to the IPC server by enqueueing it on the server's sendQ. The server notices that the Buffer is to be transferred over the network, so it passes the Buffer to its lower layer, the transport layer, again by an enqueue operation. Since TCP is the transport layer protocol in this example, TCP prepares a TCP Buffer, links the IPC Buffer to it and passes the result to the network layer. The network layer, IP, prepares an IP Buffer and passes it to the device layer. The device layer, Ethernet, then prepares an Ethernet Buffer and notifies the Ethernet controller to transmit the data over the Ethernet network. At this time, the Ethernet Buffer's internal structure looks like Figure 4.4, reproduced in Figure 5.3.

The content of the Buffer is transmitted using an algorithm which traverses each header and transmits all the headers first, then user data followed by trailers. If the network controller supports the *gather* capability, then copy operations can be completely avoided since the controller can directly transmit multiple memory fragments including those from the user process without any copy operation. The actual transmitted packet is shown in Figure 5.4. In this way, the Buffers are passed by reference from the user layer to the device layer, avoiding copy operations as much as possible. When the Ethernet controller transmits the packet to the remote

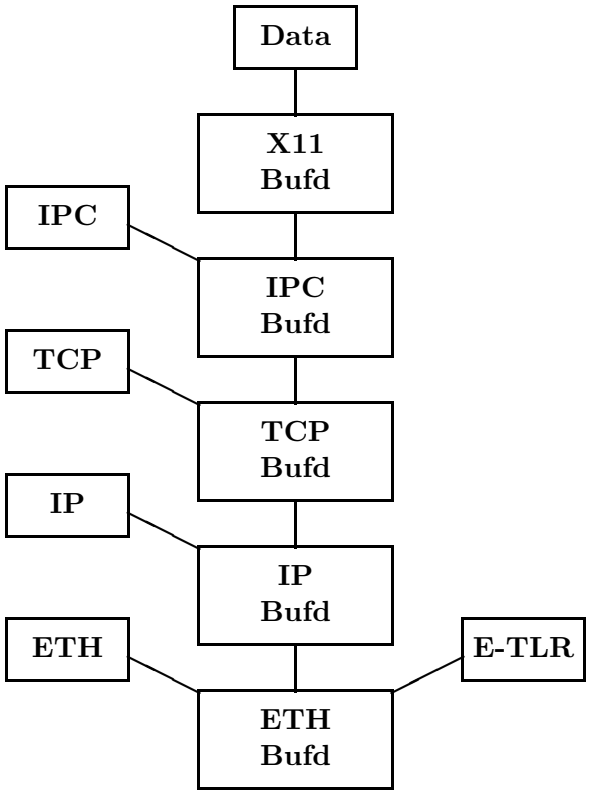

Figure 5.3: The Internal Structure of an Ethernet Buffer

node successfully, the Bufds in the Ethernet Buffer can be returned recursively to the previous returnQs, or directly to individual returnQs by the Ethernet or any intermediate layer as appropriately flagged. As mentioned earlier, this is possible due to the capability to access nested higher layer Bufds inside a Buffer at any level.

This example demonstrates not only how one can implement conventional network communication efficiently, but also how different types of entities can communicate using a uniform data structure as well as a uniform communications interface. The entities involved in this example range from user processes, through

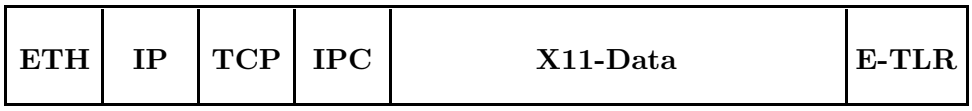

Figure 5.4: The Transmitted Ethernet Packet

kernel routines or processes, to devices.

5.4 Distributed Communication

Next, an example of distributed communication, which involves transferring Buffer objects to remote BufQs in a distributed system, is presented. As demonstrated earlier, the complexity of the Buffers' internal structures can vary greatly. How some of these simple and complex Buffers are transferred among Queues within a single memory domain has been demonstrated in the last three examples. In the network communication example presented above, only the content of a Buffer is transmitted to a remote node. To achieve transparent internal communication throughout the system, any Buffer object should be transferrable to anywhere in the distributed system while keeping its original structural form intact. The BQP layer also relays other control functions (*e.g.*, abort, enquiry and recovery) and information (*e.g.*, returnQ, return_status, *etc.*) essential for remote Buffer and Queue operations. Here, we demonstrate how the BQP supports location transparency in distributed communication. The transfer of short messages is demonstrated in this example and the transfer of long messages in the next example.

In the X11 communication example used earlier, a reasonable place to insert the BQP layer is between the IPC and transport layers. Since the transport layer (*e.g.*, TCP) provides reliable end-to-end data delivery, the BQP layer can ignore

this aspect and concentrate on transferring the content as well as the structural information of Buffers.

As in the network communication example, the X client prepares an X11 Buffer and invokes an X library routine, which in turn prepares an IPC Buffer and enqueues it to the IPC server. Since the Buffer is destined to a remote X server, the IPC server passes the Buffer to the BQP layer by enqueueing it to the BQP server. The BQP server fills in the source and destination QIDs in the BQP header. If higher-level naming was used by the user, a name server can be called to resolve the destination address. The BQP server sets an appropriate packet type and flags in the BQP header as well. Since the BQP layer sits on top of a reliable stream, the *sequence*, *more* flag and *chksum* fields (of the BQP header shown in Figure 4.15) are redundant unless BQP is multiplexing several concurrent operations and thus fragmenting and sending Buffers piecemeal itself. Since there are two layers, IPC and X11, above the BQP layer, it sets 2 in the *layer_count* field. The *total_message_length* field would contain the length of all the headers and data above the BQP layer. The *number_of_bufds* field would contain the value 2 since there are two Bufds above the BQP layer: the IPC and X11 Bufds. The BQP server then traverses the Buffer structure pointed by its next Bufd field and sets the *linearized* flag field in the IPC and X11 Bufds. As explained earlier, the Bufds with this flag set are transmitted to the remote node as part of data to be used for reconstructing the original Buffer structure.

The BQP server then passes its Buffer to the transport layer. The lower layers (*i.e.* transport, network and device layers) prepare their Buffers, perform protocol processing and pass them to their lower layers as described in the network communication example. The resulting Buffer structure before transmission is shown in Figure 5.5. The content of the Buffer is transmitted using an algorithm which is a

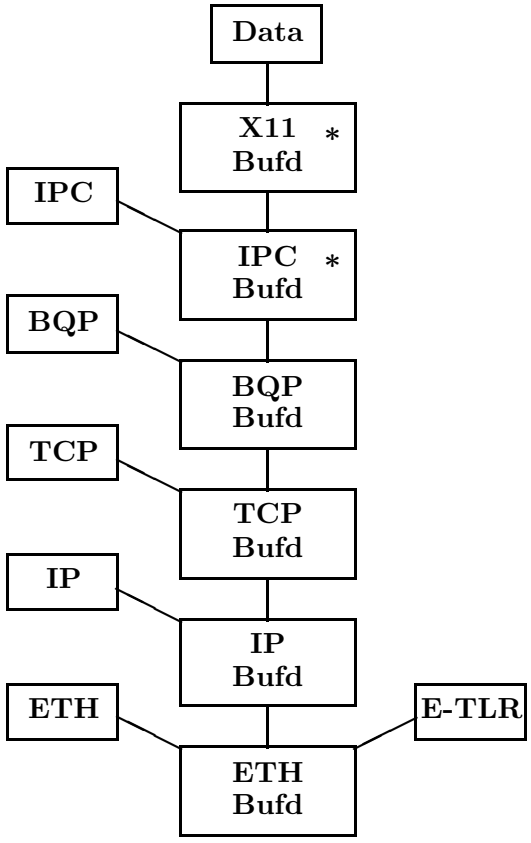

Figure 5.5: The Internal Structure of an Ethernet Buffer with the Flagged Bufds

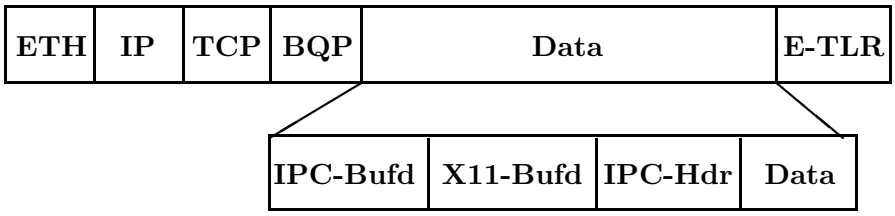

Figure 5.6: The Transmitted Ethernet Packet Containing Flagged Bufds

.

variation of the one used for the conventional network communication packet transmission (the third scheme described in Section 4.4.1 is used here). As each Bufd is visited, the algorithm checks to see if the *linearized* flag is set. If it is not set, the Bufd is not transmitted but only the header. If it is set, the Bufd is transmitted first and the algorithm continues upward until no further Bufd is found.[1] Then all the headers, trailers of flagged Bufds and user data are transmitted, followed by the trailers of unflagged Bufds. The corresponding Ethernet packet transmitted is shown in Figure 5.6.

When the packet arrives in the remote node, it is processed in a conventional manner (*i.e.*, headers and trailers are stripped from the received data and passed to higher layers) until data reaches the BQP layer. The BQP server also strips its header from the data. It then reconstructs the IPC Buffer from the information in this header. It strips the IPC Bufd and X11 Bufd from the data and fills in the appropriate local information such as local virtual addresses of the X11 Bufd and IPC Bufd. Thus, the resulting IPC Buffer contains the IPC header block, the X11 Bufd, and the user data block. It also sets a local returnQ QID and the appropriate inverse QID mapping so the local return Bufds will be caught and redirected to the remote node. It then passes the IPC Buffer to the IPC server by enqueueing it to the server's sendQ. The IPC server performs a matching operation. If a matching Buffer (*i.e.*, the intended receiver's Buffer) is found in the server's receiveQ then the data transfer takes place, otherwise it remains in the server's sendQ until a matching Buffer arrives. Upon completion, the Buffer will be returned to the BQP layer, which, acting as the agent for the original sender, will ensure that status and controls are returned to the originating node and the corresponding actions and

[1] Unless there was an error in preparations of Bufds above the first Bufd that had the *linearized* flag set, the flag should be set in all Bufds. Thus, these flagged Bufds get transmitted as part of data.

clean-ups carried out.

5.5 Distributed Communication with Long Messages

The last example demonstrated a distributed communication involving short messages. This example involves long messages (*e.g.*, 100 kilobytes), which requires message fragmentation and reassembly. Assume that the IPC layer does not fragment the message but passes the entire message to its lower layer, the BQP layer. Since the BQP layer has been placed on top of a reliable stream transport protocol, TCP, the BQP layer also does not fragment the message but simply transmits the entire message to the transport layer. TCP and/or IP will fragment the message into a number of smaller message fragments so that each can fit in an Ethernet packet.

The remote BQP layer will receive a small message fragment at a time from its lower layer, because of the TCP stream semantics. The first BQP packet will contain the BQP header, all the higher layer headers and Bufds, the structural information for reconstructing the Buffer, and perhaps even the first few bytes of user data. (If all the Bufds did not arrive in the initial packet, the remote BQP server simply waits for the subsequent packet(s) until all Bufds arrive.) At this time, the Buffer can be reconstructed by stripping the Bufds and headers from the arrived packet, filling in the appropriate local information in various fields of Bufds and linking them appropriately to form a template of a segmented Buffer according to the passed structural information. In conventional protocols, all the message fragments would have to be reassembled before the message could be passed

to the higher layer. However, much greater flexibility can be achieved using the BQP protocol. The BQP allows one to pass each subsequent message fragment immediately as another segment, wait for multiple message fragments until some desired amount of data is accumulated (*i.e.*, partial reassembly) or wait until the entire message has arrived. The reassembly does not need to be done in the same layer as fragmentation is performed at the source. In fact, the BQP allows the reassembly to take place at an arbitrarily higher layer (*e.g.*, at the user layer) as will be demonstrated below.

The IPC layer also has an option of sending each message fragment to the user or waiting for the entire message. An example of message reassembly at the IPC layer is shown in Figure 5.7. Since the IPC layer knows that all Buffers in a chain are part of a segmented Buffer, it can return the Bufds and headers from each template immediately to the BQP layer reassembling the data into a single Buffer or a more compact segmented Buffer as an efficient memory management measure. The IPC server can keep the full structure of the first Buffer and keep only the X11 Bufd and data memory blocks for the subsequent Buffers. Thus, the data fragments can be linked via queue pointers in the X11 Bufds.

The capability of deliverying each fragment to an arbitrarily higher layer without reassemblying all the fragments at the same layer as the source is one of the novel aspects of the BQP. This capability is essential for supporting a stream type of communication, where arbitrarily-sized fragments of data are written or read. This capability is also useful for efficient resource management. Since each fragment can be delivered and processed all the way up to the final destination Buffer (*i.e.*, the user Buffer), tying up large amounts of valuable resource for a long period of time can be avoided. Using the conventional approach, the same amount of memory would be needed as the original unfragmented message memory tied up in the IPC

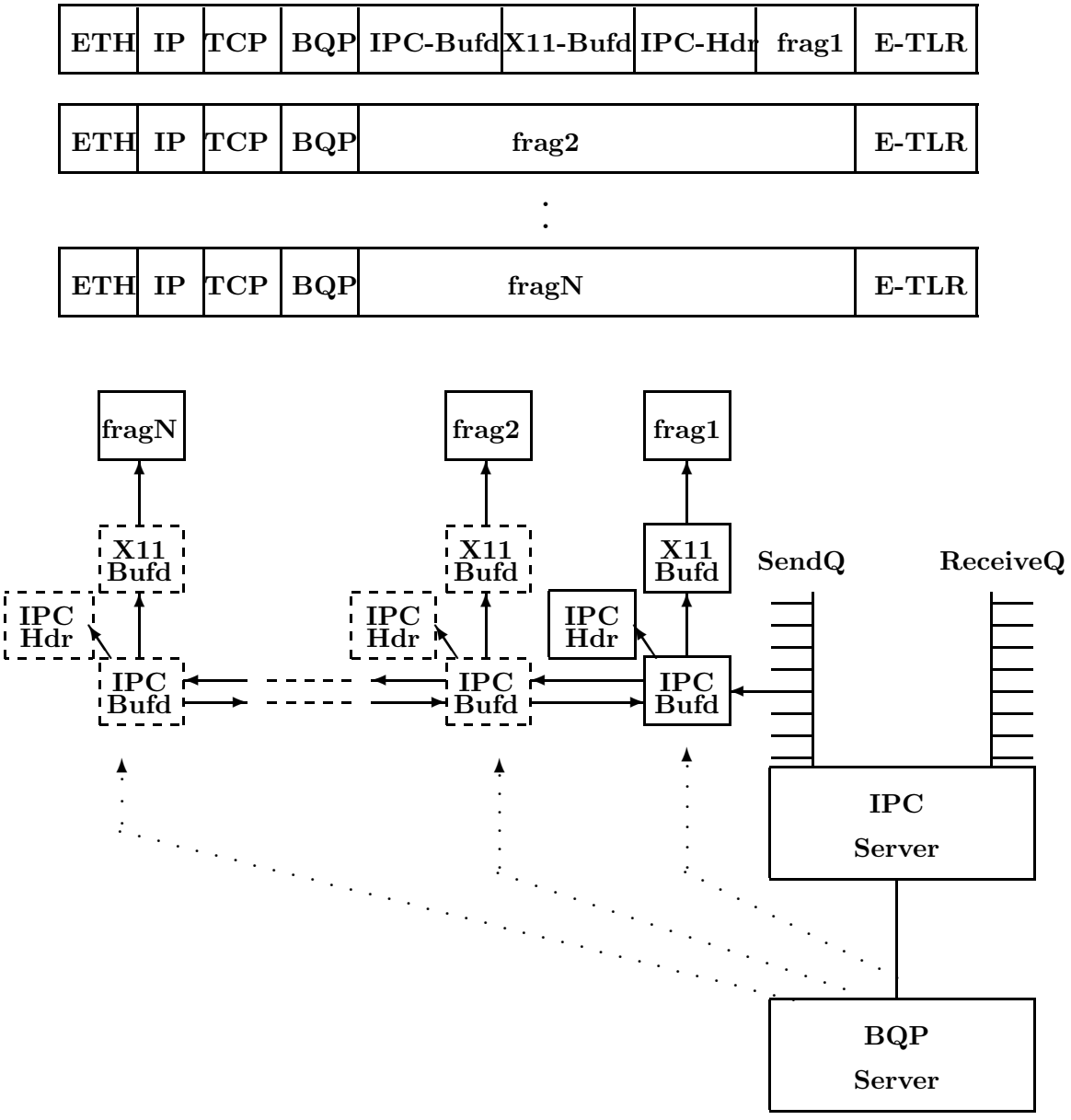

Figure 5.7: Reassembly of a Long Message in Distributed Communication

layer until all the fragments are reassembled. Here, the IPC layer could queue message fragments until either a receive operation has been issued at application level or a buffer has been made available by some other means, then copy them as they arrive. Furthermore, since the copy operation is avoided in passing the Buffer from one layer to another, the extra memory which would be needed otherwise is not necessary.

Here, multiplexing in the BQP layer is described briefly, which can be useful and more efficient if multiple users share a single transport channel at the BQP layer. Kernel-to-kernel communication [Vasu87] is a good example of using multiplexing. Rather than dedicating the entire bandwidth to one user and denying service to others for potentially a long period of time, assigning a portion of the bandwidth to each user may be more efficient. This can be achieved by fragmenting large messages in the source BQP layer, multiplexing multiple user data and demultiplexing at the destination BQP layer. Here, the *more* flag, *sequence* and *chksum* fields of the BQP header need to be used. The *more* flag will be set as each fragment is transferred except the last fragment. The *sequence* and *chksum* fields are used for error recovery purposes.

5.6 Optimistic Blast Bulk Data Transfer

Recently, there have been studies of optimistic blast protocols for supporting high throughput user-to-user bulk data transfer [Cart89, OMal90]. Here, these protocols are briefly examined because they provide special ways of handling long messages in a distributed environment. Since the Buffers and Queues provide an efficient solution for transferring long messages in a distributed environment, these specialized protocols should also be easily supported. Thus, a solution is proposed, which

will achieve the same goal, based on the Buffers and Queues. This exercise will also demonstrate the flexibility of the Buffer and Queue Model, which can be used to support very specialized protocols such as the optimistic blast protocols.

These optimistic blast protocols basically increase throughput by avoiding the copy operation of data from the kernel address space to the user address space and vice-versa, with the help of the *scatter-gather* capability of the network controllers. These protocols are based on Zwaenepoel's blast protocol [Zwae85], where the sender fragments large messages into multiple network packets and sends all the packets, one immediately after the other, without waiting for each packet to be acknowledged. The receiver sends an acknowledgement only after all the fragments have been received. Carter and Zwaenepoel's optimistic blast protocol [Cart89] supports only a single transport protocol, namely the V-kernel's. On the other hand, the optimistic blast protocol of O'Malley *et al* [OMal90] can support multiple protocol suites.

These optimistic blast protocols all require a *scatter-gather* capability on the network controllers. They can avoid the copy operation of data by having advance knowledge of the likely destination of data in the next incoming packet. On the sending side, the user provides the source address of the data by passing its address to the device driver. The sender also provides the destination address of a Buffer for the reply message in the case of a request/receive/reply interaction. On the receiving side, the device driver can obtain the address of the receiver Buffer from the information contained in the sender's network frame. In Carter and Zwaenepoel's protocol, the receiving network device driver peeks at the header of the network frame to detect the first fragment of a blast sequence, and thus obtains the destination address of the message fragments. O'Malley *et al*'s facility transmits an explicit control frame to signal the start of a blast sequence. They support two

optimistic blast protocols: *padded blast* and *by-request blast*. The padded blast protocol is similar to Carter and Zwaenepoel's in that it uses a fixed header size. In contrast, the padded blast protocol supports variable-length headers by padding each header to the same fixed length. The control frame sent to signal the start of a blast sequence does not contain any header information, since the receiving device driver knows the header size and hence can figure out where the user data starts. The by-request blast protocol does not use the fixed header size but relays the pertinent information (such as total size of the data, the size of the first header, *etc.*) in the control frame to the receiving device driver.

Both research groups have used some interesting memory management "tricks" as well as network controllers that have the *scatter-gather* capability. Some of these "tricks" involve the fixed header size for supporting only a single transport protocol or a fixed header size that is big enough to handle any existing stacks of network communication protocols, and requiring that user data be page-aligned. These optimistic blast protocols can be easily supported using the Buffers and Queues with the same "tricks".

Here, a solution based on Buffers and Queues is presented, that will support multiple protocol suites without the constraints imposed on O'Malley *et al*'s protocols (see Figure 5.8). Since the Buffer structure (described in the Buffer abstraction) is already suitable for avoiding the copy operation as it traverses the network communication layers, if a *scatter-gather* network controller is used, any copy operation of data can be avoided on the sending side. In this case, the BQP protocol can transmit the Bufd information necessary for the proper interpretation of subsequent data. Recall that the BQP protocol is responsible for transmitting all the headers, data, trailers and the structural information of a Buffer.

Their optimistic blast protocols are handled at the lowest layer, the network

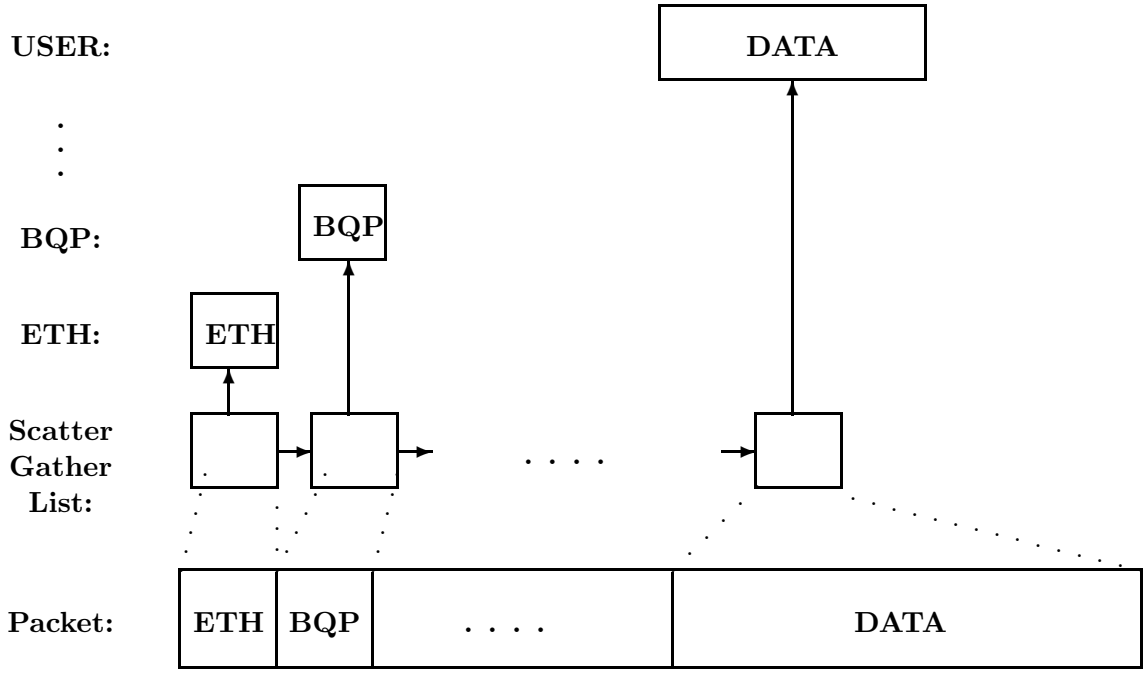

Figure 5.8: A B&Q Solution for Optimistic Blast Bulk Data Transfer

device driver, so that they can avoid any protocol processing which requires the copy operation of inbound network frames. For the same reason, the same functionality should be built in the network device driver in the Buffer and Queue solution. In the earlier examples, the BQP protocol was placed between the transport and IPC layers. Here, the layer immediately above the network device driver handles optimistic blasts with special commands to configure the network device driver reception buffers. The device driver places not only all the headers, trailers, and data, but also all the Bufds above the device layer into the outgoing network packet. This first network packet will thus contain the headers, trailers, Bufds, zero or more bytes of user data, and a flag (*e.g.* Ethernet packet type) indicating the start of a blast sequence. If the headers span more than one packet, then whatever number of initial packets are required will be handled by normal protocol processing at the start of the blast. When the header packet or packets arrive at the receiving device driver, then the BQP Buffer can be reconstructed immediately. Through normal layered processing, the incoming message must be matched with an outstanding Buffer. At this point, all the necessary information is available to configure the network controller to scatter incoming data fragments. If no outstanding Buffer has yet been supplied by the receiver, any of the solutions proposed in [Cart89, OMal90] can be chosen.

5.7 Semaphores

Next, implementation of traditional semaphores [Dijk65] using the Buffers and Queues is presented. Implementations of binary semaphores, counting semaphores (also called general semaphores), and distributed versions of these are described. A semaphore is a protected variable whose value can be accessed and altered only

by the operations P() and V(). Binary semaphores can assume only the value 0 or 1. Counting semaphores can assume only nonnegative integer values. In order to permit distributed semaphore operations while respecting the restrictions that only the owner may dequeue, the implementation is more complex than would be required to support only local semaphores.

The basic idea is to use Buffer descriptors (Bufds) as tokens or rights to access a shared object, and a SemaphoreQueue (SQ) as a place where these tokens are initially stored. A token is passed from a SQ to a client when the P() operation is invoked and a token is returned to the SQ when the V() operation is invoked. The proposal presented here is not new; the notion of "token passing" has been used in ISIS for implementing distributed semaphores [Birm87a]. However, what I hope to achieve through this exercise is to demonstrate the versatility of the Buffer and Queue model by demonstrating an implementation of various semaphores without too much overhead or effort.

In the current conceptualization of Buffer and Queue model, data elements (*e.g.*, Bufds) in a Queue are only dequeueable by the Queue's owner. This restriction forces one to implement a semaphore as a server entity or at least an object with the code to accept and process semaphore operations by the *signal* operation within the enqueue operation. In the latter case, the client's thread of control would be extended to execute the semaphore object's code when the P() and V() operations are invoked. For simplicity, the implementation using a semaphore server entity is described here. An implementation of binary semaphores is described first, followed by counting semaphores, and distributed versions of these.

For the P() operation of binary semaphores, the client sends a Bufd containing the semaphore request operation code (C-Bufd) to the SQ and the client is blocked on dequeueing the token Bufd (S-Bufd) from its receiveQ. The SQ owner replies to

the client with a token Bufd if and when the token Bufd is available for the client. It keeps the client's Bufd until the token Bufd is returned. Note that keeping the client's Bufd by the SQ owner allows the SQ owner to keep track of who currently holds the token and to recover from lost tokens and unfaithful P()'ers. It also serves as a vehicle to acknowledge the return of the token by a subsequent V(). When the token Bufd is enqueued onto the client's receiveQ, the client is unblocked and has now obtained the semaphore.

In a normal request-reply-ack transaction, the client sends an ACK back to the server to complete the transaction. However, here the client P() request is satisfied by the SQ sending a token. The latter can also be thought of as a request by the SQ to retrieve its token. The V() operation by the client is the reply to this request, and the return of the original client Bufd is the final ACK terminating the extended operation. Thus, the P() and V() operations of binary semaphores can be considered as an overlap of two request-reply-ack transactions, which involves the exchanges of four packets rather than the normal six packets. The timing diagram for the exchange of two Bufds is given in Figure 5.7.

So far, an implementation of binary semaphores has been described, where the use of a single token Bufd and a single client Bufd per client is sufficient. Counting (or general) semaphores work essentially as above, except that there would simply be more token Bufds managed by the SQ server. For example, for N-counting semaphores N token Bufds are required in the SemaphoreQueue. These token Bufds would be sent out to clients as they perform P() operations and returned as they perform V() operations. When there are no more token Bufds left in the SQ, the clients would be blocked until a token Bufd is available.

Distributed semaphores allow entities or processes located on different machines to synchronize. An implementation of distributed semaphores can be accomplished

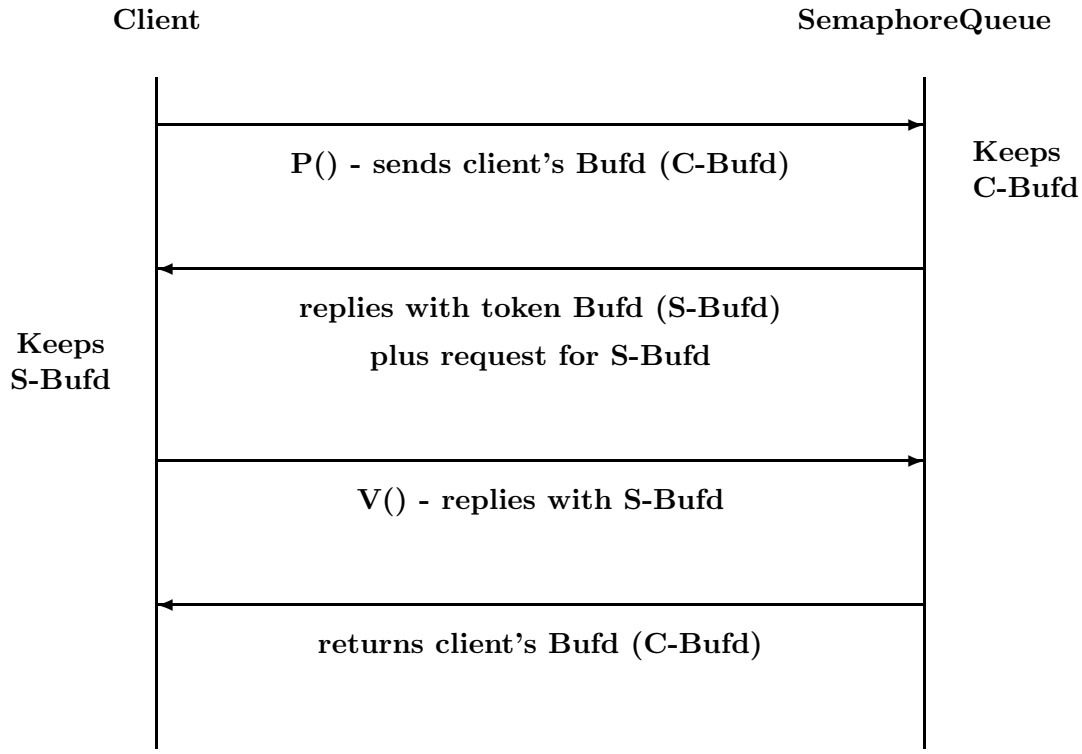

Figure 5.9: The Timing Diagram of the Exchange of Client's and Token Bufds

using Buffers and Queues with the aid of the BQP. Token request and return aspects of the distributed semaphore implementation work essentially the same as in centralized binary or counting semaphores except that the BQP is used for semaphores to work transparently over remote machines.

The BQP was originally designed to encode/decode Buffer structuring information as well as to transfer dynamic control information for remote Buffer and Queue operations. Since the delivery of data associated with Bufds is not a concern in supporting semaphores, the BQP can be used to transfer only the token aspect of Bufds and the dynamic control information associated with them. Thus, the encoding/decoding aspect of the BQP can be simplified substantially. However, since the Bufds may travel from one node to another with important information such as its SemaphoreQueue (or returnQ), requested operation code and status, they must be delivered reliably. One may implement the BQP on top of a reliable transport mechanism or utilize the reliability capability of the BQP if necessary.

Thus, Buffers and Queues can be used to implement semaphores (both binary and general) and distributed semaphores in a simple and efficient manner with the BQP.

5.8 One-Way Communication

Next, a solution for supporting an efficient one-way communication is presented, where the sender does not require an acknowledgement from the receiver(s). Broadcast and multicast basically provide one-way flow, as opposed to bidirectional flows of information. Also, datagram protocols (*e.g.*, UDP) does not obviously map well onto the transaction-based Buffer and Queue Model. One can always send an extra packet and drop or ignore it, but it might be useful to build a component into

Buffers and Queues that handles such things as part of the protocol and thus can avoid redundant steps in intermediate operations and optimize efficiency.

Clearly, once an entity has copied the datagram message from a Buffer, the Buffer can be deleted or returned. After reception, if one passes back a Buffer to a local intermediate entity such as the BQP server, and it is known that the originating remote Buffer was a datagram Buffer, then there is no need to acknowledge it or clean-up previous elements of the chain, and it can be grounded (*i.e.*, no acknowledgement is sent over the network). However, it is useful to retain the shared-memory transaction-like behavior within a particular domain (*e.g.*, local group multicast) for the purpose of resource management. By recognizing datagram Buffers, it is possible to avoid a packet exchange, unblock the sender as soon as transmission is complete, but still use Buffer and Queue facilities to manage system resources on both nodes. A simple and efficient solution to achieve this optimization is to have the sender set a flag in the Bufd. After transmission, the Buffer is returned immediately to higher layers. On reception, the intermediate receiver would check this field upon processing a returned Buffer and act accordingly. From the sender's perspective, this is an example of partially synchronous blocking, or ACK within a single domain only.

To elaborate slightly, when a one-way remote data transfer is initiated by a local entity, local resources will be allocated appropriately to construct required Bufds as explained in previous examples involving network communication. The "prepared" Buffer is passed to a local BQP server, which in turn flags the Buffer to be one-way communication, and then passes it down the layers of network protocols. When the network device driver transmits the content of the Buffer, the Buffer can be returned to the layers above. Each layer above the device layer can reclaim its Bufd and memory associated with it, and pass the remaining Buffer structure

upward as explained in the network communication example. This process can go on recursively until all Bufds are returned to its appropriate places including the user since it is a one-way communication requiring no ACK or reply. In two-way communication, the Bufds above the BQP layer will not be returned to their owners until an ACK or reply message is received from the remote BQP layer.

On the receiving host, when data is delivered to the user, Bufds will be returned to appropriate places as well, this time in the downward direction. When the returning Buffer reaches the BQP layer, using the information stored in the BQP layer, the BQP server will detect that this particular transfer was one-way and will not generate a reply message to the sender BQP server[2]. The remaining Bufds will then be returned to their appropriate places.

5.9 Distributed Shared Memory

Recently, two terms, *distributed shared memory* and *distributed virtual memory*, have been used in the literature with no clear distinction emerging between the two. For the purposes of this thesis, distributed shared memory (DSM) is defined as a way of extending shared memory to a distributed environment for communication. Mirage [Flei89] and Agora [Bisi87] are two examples of DSM. Distributed virtual memory (DVM), on the other hand, is an extension of virtual memory in the operating system sense to a distributed environment.

Distributed shared memory thus refers to logical sharing of memory across a distributed computing environment. It gives the illusion to users that the memory being shared (whether it is physically local or remote) is local memory. The un-

[2]In two-way communication, the return of local Bufds will trigger the transmission of a reply message to the sender BQP server.

derlying system supporting distributed shared memory is responsible for handling issues such as *coherence*, *access synchronization*, *address space structure*, *block size* and *replacement policy* to provide that illusion [Tam90]. The first two issues, namely coherence and access synchronization, can be easily supported by the Buffers and Queues. However, the latter three issues are really part of memory management and thus outside the scope of the Buffer and Queue Model.

The data abstraction in the Buffer and Queue Model is flexible enough to describe various granularities of memory objects. For example, it could represent memory of a single byte, pages of an address space, or permanent memory objects (files). Coherence, which refers to consistency control of write and read operations on various memory objects, can be supported by the owner or home concept of Buffers. Write or page request operations on shared memory are sent to the owner node, which would then make appropriate changes to the master copy or reply with the requested page. The owner would also take appropriate actions to make sure that the readers currently holding the stale data get the updated data.

Access synchronization is another important issue in supporting distributed shared memory, and it ensures that no more than a single writer accesses the shared memory. A simple solution using the Buffers and Queues is that the token aspect of Bufds can be used to control access synchronization as presented earlier in the semaphore example. The writer must obtain the token (that is, a Buffer descriptor) for any particular segment of memory before he can make changes to that segment.

The Buffer and Queue model, as demonstrated here, can easily and efficiently support important issues of distributed shared memory. However, since the low-level memory management issues such as the structure of the shared address space, page replacement policy, and block size are outside the scope of the Buffer and

Queue model, it must be integrated with memory management to provide full support for distributed shared memory

5.10 Summary

In this chapter, implementations of various communication paradigms and facilities were given using Buffers and Queues. Several examples of simple internal communication, which enjoy the efficiencies of the shared-memory paradigm, were presented. These internal communication examples showed that Buffers and Queues can be used between the simplest communicating entities such as procedures and coroutines, and more complex entities such as threads in a single domain. The local interprocess communication example showed interactions between local processes, each with its own protection domain. Network communication as well as distributed communication examples demonstrated the use of a uniform data structure and a universal interface between different types of entities.

The shared-memory communication paradigm is extended transparently to remote communication by inserting the BQP layer, as demonstrated in the two distributed communication examples. The bulk data transfer example demonstrated that Buffers and Queues with the help of the BQP can implement specialized protocols easily and efficiently. Other examples also presented were implementations of conventional and distributed semaphores, one-way communication and distributed shared memory. Although an explicit solution was not presented I feel that more complicated communication mechanisms such as Psync[Pete87] can be easily implemented using Buffers and Queues.

Chapter 6

Conclusions

6.1 Summary

This research in communication in distributed systems has focused on managing the complexity of communication. In conventional systems, there exist various types of communication paradigms, abstractions and entities. Communication between different types of paradigms, abstractions or entities is very difficult (if not impossible). This research has focused on developing a single efficient and consistent programming interface that can be used by programmers to implement various communication interactions both within and between various paradigms, abstractions and entities. Furthermore, a hierarchical framework has been provided that can be used to develop both existing and future communication paradigms and protocols.

In this study, a survey was done on various existing communication paradigms and the issues involved in distributed systems as a means to identify and categorize some of the fundamental aspects of all communications. In particular, aspects of intra-process communication (communicating within the bounds of a protection

domain), inter-process communication (both within and across protection boundaries), and network communications (general device and kernel-to-kernel communications) were examined for similarities and differences. The fundamental aspects identified in Chapter 2 were examined more fully, and were redefined and organized them into a set of communication abstractions, namely *data*, *node*, and *delivery/synchronization*. A framework has been built based on this set of abstractions. The result is a generic communication model, which provides a single efficient and consistent programming interface that all communication paradigms, abstractions and entities can deal with for various types of interactions. Further, the generic communication model can be used to construct arbitrary communication systems.

A specific instantiation of the generic communication model called the Buffer and Queue Model was developed, which consists of three communication abstractions, namely Buffer, Queue and delivery/synchronization abstractions. These abstractions are combined to implement a communication model for distributed systems, which is simple and low-level but powerful and efficient, and which satisfies the constraints of the generalized paradigm developed in Chapter 3. The Buffer-Queue protocol was used to extend the programming interface to a distributed environment so that the conceptual efficiencies of a single memory domain can also be enjoyed between remote entities.

The versatility and generality of the Buffer and Queue Communication Model were demonstrated by presenting implementation examples of a variety of communication paradigms and communication-related problems using the model. In particular, examples of internal communication, local interprocess communication, network communication, distributed communication and bulk data transfer were presented. Further, examples of implementing distributed and conventional semaphores, one-way communication (*e.g.*, multicast) and distributed shared memory were also pre-

sented.

6.2 Discussion of Related Work

A common problem in the other attempts at providing a single consistent programming interface (such as Berkeley sockets, AT&T STREAMS, Conduits, and the x-Kernel) is that their concepts do not extend well to remote nodes. They certainly have merits of their own for supporting local IPC or network communication. However, they all have to go out of their abstraction to extend their concepts nicely to distributed environments. First, they all lack global identifiers that can be used to identify and name their abstract objects across a system. Second, I do not believe they have considered how to mix different types of entities as extensively as I have. Last, but not least, they have not developed a specific protocol that can allow transparent remote operations. On the other hand, the Buffer and Queue abstractions view a distributed system as a single node, where efficiencies of a local-memory environment can be enjoyed. That is, there is no node dependence in the model. In reality, however, there exist separate nodes and protection domains. The transparency which allows one to view the entire system as a single shared domain has been achieved through an appropriate standardization and hierarchical development of the abstractions, and through the linkage provided by the BQP protocol as shown in Chapter 4.

As mentioned in Chapter 1, TACT [Auer90] and this work have a lot of similarities in both approaches and goals. Both abstract complex problems in communication into a set of simple concepts and standards that is more manageable. Both strive to provide a universal interface between different types of endpoints. In TACT, Auerbach introduced five canonical forms in the transport level as well as

the tools to convert a specific type of data flow to one of these forms. The conversion may actually involve one or more abstract conversions between canonical forms themselves before achieving the form required by the destination. In my work, I go one step further and provide a standard interface and a standard data structure for various types of entities at all levels (not just in the transport level as in TACT) across a wide spectrum of environments. Some local conversion may be involved, which depends on and is handled by each local entity. However, all entities deal with a uniform data structure through a uniform interface. Thus, I claim that the communication programming interface developed in this thesis is "truly" universal.

In the *x*-Kernel, protocol and session objects attempt to provide a uniform interface for each communication protocol. Their interface provides what I call a "procedure-call interface" as discussed in Section 3.2.2. However, the communication interface presented in this thesis is more generic than theirs, since it can support both the procedure-call interface and the "self-contained objects" interface (also discussed in Section 3.2.2). Furthermore, they have designed their interface for network communication protocols, whereas I have designed it to be used for any type of communicating entities. Thus, I believe that my communication interface is more generic and universal than theirs.

Another major difference between the *x*-Kernel work and this work is that they have included low-level memory management aspects into communication abstractions whereas I have not. Although memory management is definitely an essential aspect of communication, I strongly believe that low-level aspects of memory management such as how memory is allocated and deallocated should not be a concern of the communication system but that of users of the communication system or the memory management subsystem. Although memory management is intentionally (and correctly) separated from the generic model and the Buffer and Queue

communication model, I have incorporated principles and tools that are useful in maintaining sound memory management.

6.3 Future Work

One of the obvious extensions of this research is to do a full implementation of the Buffer and Queue communication mechanism based on the class definitions presented in this thesis. Some *ad hoc* implementations for a couple of communication paradigms have been demonstrated by Dave Shewchun using the Buffers and Queues concepts in the Shoshin distributed software testbed [Shew90]. Although he did not take full advantage of the Buffer and Queue class hierarchies presented in this thesis, he followed the general concepts of Buffers and Queues and demonstrated that it was possible to implement a serial streams capability on top of Buffers and Queues.[1] He further demonstrated that local message-passing could be handled with the addition of a kernel communication manager called the *PostMaster*.

The full Buffer and Queue implementation would involve filling in the detailed code for various operations which are part of the class definitions as well as incorporating any system-dependent details. With appropriate modifications to the PostMaster and kernel Buffer and Queue management subsystem, supporting local message-passing, streams, shared memory, and mixed interactions among them should be relatively easy. The next step could be to implement the Buffer-Queue protocol layer, to allow distributed communication (*e.g.*, remote message-passing, streams) via conventional network communication. In order to truly take full advantage of the Buffer and Queue paradigm, the next step could be to imple-

[1] The implementation was done in C.

ment various network protocols for various types of network communication (*e.g.*, connection-oriented, connection-less) using Buffers and Queues. Other examples of communication facilities and protocols presented in Chapter 5 could also be implemented.

In Chapter 4, three different ways of transmitting the structural information in Buffers to remote nodes were presented as part of the BQP protocol. One involved a physical structural change to the internal Buffer structure so that the information required to be transmitted was arranged as if it were user data. The other two schemes involved a change not in the physical internal structure but in the way the read operation traverses the Buffer structure to transmit the necessary information. Also, two ways of sending Bufds to remote nodes were discussed. One is to send only the *essential* information, and the other is to send the whole Bufd. An interesting study would be to implement various combinations of these choices and make performance comparisons.

Other possible future work involves developing other communication systems or models based on the generic communication model. The Buffer and Queue Communication Model is just one instantiation of the generic model. I envisage that other specific systems or models (such as Berkeley sockets, the x-Kernel, Choices Conduits, and AT&T STREAMS) can be easily developed from it. For example, I believe that one can easily apply the concepts of Conduits and ConduitMessages to the generic model to develop the Choices Conduit communication system. Also, one can apply the concept of STREAMS Queues and Messages to the generic model to develop AT&T STREAMS. I believe that the generic model is sufficiently general to develop most existing and future communication systems and models. This type of extension would presumably involve incorporating system-wide identifiers, a protocol for remote operations, and serious consideration of how to mix different

entity types.

Another possibility involves applying the hierarchical framework that was used in this research to other disciplines of operating systems. Memory is an abstraction for data storage. Thus, one might also apply the same framework for the data abstraction to memory objects. There are various types of memories: main memory, secondary memory, cache memory, virtual memory, distributed virtual memory, *etc.* Managing various types of memory is becoming as complex as managing communication in distributed systems. By an appropriate hierarchical framework, one would hope to reduce the complexity of memory management to a more tractable and universal form.

Somewhat closely related but a more concrete application of memory management is in supporting *memory objects* in the sense of Mach memory objects [Acce86], Clouds objects [Dasg87], and Choices memory objects [Camp87] using the Buffer and Queue concepts. The Buffer concept abstracts an ordered sequence of bytes. I can envisage implementing a disk storage subsystem by extending the concept to include permanent Buffers. After all, a disk represents an ordered sequence of bytes with particular internal structures (*e.g.*, blocks, cylinders, sectors, *etc.*). Furthermore, structured memory objects with different types of segments (*e.g.*, data segment, code segment, *etc.*) might also be handled using a form of recursive Buffers.

I envision memory object servers with one or more Queues as interfaces, where memory objects are stored and retrieved. These memory objects would be efficiently transferred from one Queue to another within a protection domain or across protection domains. The concept of memory objects is often being used as a basis for supporting distributed virtual memory [Youn87, Russ89], and thus supporting distributed virtual memory using the Buffers and Queues falls out quite naturally.

I believe the Buffers and Queues possess a set of tools that is simple, low-level but powerful and versatile to support these and other modern concepts in distributed systems.

6.4 Contributions

This research has focused on reducing the complexity of communication in distributed systems. Currently, programmers use *ad hoc* approaches in developing and implementing communication paradigms and protocols. There were two goals in this research. One was to provide a framework upon which arbitrary communication systems or models can be developed. The other was to develop a set of simple standards and tools that can provide a single efficient and consistent programming interface that can be used to implement various communication interactions both within and between various paradigms, abstractions, and entities.

In this study, the fundamental aspects of communication in distributed systems were examined and organized into a set of abstractions; this forms the basis for a generic communication model. This model satisfies both of the goals. That is, the generic communication model provides a single programming interface that the communications programmers can use to implement interactions both within and between different types of paradigms, abstractions and entities. This interface allows its users to cross boundaries of protection and temporal domains without having to deal with the conceptual partitioning of interfaces based on the type of target module. Further, the hierarchical framework can be used to develop arbitrary communication systems or models. Thus, a major contribution this thesis has made is the development of such generic communication model.

Another major contribution this thesis has made is the development of the Buffer

and Queue Model, which meets the constraints of the generic communication model, and thus provides a single consistent programming environment which utilizes the conceptual efficiencies of the shared-memory paradigm. The Buffer-Queue protocol was developed to extend the Buffer and Queue abstractions transparently to a distributed environment. The Buffer and Queue model is simple and low-level so that various communication systems and utilities can easily be built on top. The generality and versatility of the Buffer and Queue Model were demonstrated by presenting implementation examples of various existing communication forms and utilities.

The main problem intended to solve in this thesis was to find answers to the following set of questions:

- How can a programmer write code to locate some service, and then communicate with the provider of that service?
- How can this code be made independent of the memory domain, execution control regime (procedures, threads, processes, device drivers *etc.*), and physical location of the service?
- Can this independence be achieved even when the communicating entities have different characteristics?

The Buffer and Queue communication paradigm provides a simple and efficient solution for the above problem, as shown by the code segment in Figure6.4.

The name server returns the QID of the service in *Dest_Q* (assuming such a service exists). The client sends the request by simply enqueueing the Buffer filled with the message onto the service provider's Queue. The client then retrieves the status of the request (or a reply message) by dequeueing the returning Buffer from

```
simple_op() {
       Buffer_Queue  *Dest_Q, Return_Q;          // create Qs
       Buffer        *Bufp, Buf( Return_Q );     // create Bs

       Ask_Name_Server( ‘‘Service’’, Dest_Q );  // locate service

       Dest_Q->enqueue( Buf );                   // send request
       Return_Q.dequeue( Bufp );                 // get reply or ack
}
```

Figure 6.1: The Code for a Buffer and Queue Communication Example

its returnQ. The beauty of this code using Buffers and Queues is that it can be used independent of memory domains, execution control regime, physical location of the service and types of entities. For example, the client and server can be of different types of entities, located locally or remotely, executed by a single thread or by separate threads, in a single memory domain or in separate memory domains or any combinations of these.

Using the Buffer and Queue communication programming paradigm, the systems programmers can implement various communication systems efficiently. Since the programmers are using a single consistent programming interface, supporting communication between different types of paradigms, abstractions and entities is not any more difficult than supporting communication within a single paradigm. Further, these communication systems will provide the same universal interface to user level applications, and thus allowing an efficient and truly universal programming environment in both system and application levels. I believe the Buffer and Queue programming paradigm can not only solve a lot of problems in current systems but also in the future systems, in which heterogeneity will undoubtedly increase.

Finally, the following two aspects are minor contributions of this thesis. First, the systematic design approach used for developing the generic communication model appears to be applicable to reducing complexity of other operating system disciplines as well, and it would be interesting to explore the possibilities in memory management and process management. Second, the survey of current communication paradigms and issues involved as a first step in this research can be used as a comprehensive reference to the literature of communication in distributed systems.

Appendix A

Buffer Classes

```
class Simple_Buffer {
   ADDR       address;        // start of buffer
   int        size;           // buffer size
   OFFSET     offset;         // offset of start of valid data
   int        count;          // number of valid data bytes
                              // (offset + count <= size)
 public:
   Simple_Buffer();           // Buffer constructor
   ~Simple_Buffer();          // Buffer destructor
   // write data into and read data from buffer
   virtual int    write_data( ADDR data, int len );
   virtual int    read_data( ADDR data, int len );
   // move the pointer to specified location in buffer
   virtual int    seek_data ( int type, OFFSET off );
   // get info related to valid data
   virtual OFFSET get_offset();
   virtual int    get_count();
};
```

Figure A.1: The Simple Buffer Class Definition

```
class Segmented_Buffer : Simple_Buffer {
   // linkage
   QDHDR      q;                // doubly-linked Queue pointers
   // identification information
   BTYPE      type;             // Buffer type
   BID        bid;              // Buffer identifier
   int        sequence;         // sequence field for memory block
   PID        owner;            // owner of Bufd
   QID        returnQ;          // return queue
   // status information
   QID        currentQ;         // current queue
   int        ref_count;        // Buffer reference count
   int        return_status;    // Buffer operation status
   FLAG       more_blocks;      // more horizontal Bufds if set
   // data
   Simple_Buffer* databuf;      // data block
 public:
   Segmented_Buffer( BufQ returnQ );   // constructor
   ~Segmented_Buffer();                // destructor
   virtual int write_data( ADDR data, int len );
   virtual int read_data( ADDR data, int len );
   virtual int seek_data( int type, OFFSET off );
   virtual OFFSET get_offset();
   virtual int    get_count();
   // operations for fragmentation and reassembly
   virtual int   fragment_data( OFFSET off );
   virtual int   reassemble_data( Seg_Buf b1, Seg_Buf b2 );
   // operations for setting and getting Bufd fields also go here
   virtual BTYPE get_type();          // get Buffer type
   virtual BID   get_bid();           // get Buffer identifier
   virtual int   get_sequence();      // get sequence field
   virtual PID   get_owner();         // get owner of Bufd
   virtual QID   get_returnQ();       // get return queue
   virtual QID   get_currentQ();      // get current queue
   virtual int   get_ref_count();     // get Buffer reference count
   virtual int   get_return_status(); // get Buffer operation status
   virtual FLAG  is_last_block();     // check if last block
};
```

```
class Recursive_Buffer : Segmented_Buffer {
   // a subclass of Segmented_Buffer
   FLAG       linearized;       // send Bufd if this flag set
   FLAG       which_blks;       // indicates which memory blks exist
   // Bufd structure information
   Simple_Buffer* header;       // protocol header
   union struct {
       Recursive_Buffer*  bufdptr; // recursive Bufd pointer
       Simple_Buffer*  databuf; // data buffer
   } nextptr;
   Simple_Buffer* trailer;      // trailer
 public:
   Recursive_Buffer( BufQ returnQ );  // constructor
   ~Recursive_Buffer();               // destructor
   virtual int read_data( ADDR data, int len, int type, int level);
   virtual int write_data( ADDR data, int len, int type, int level);
   FLAG which_blks_exist();     // which memory blks exist?
   FLAG check_linearized();     // check if Bufd linearized
   int  push_blk (addr, len, type) // insert control block
   int  pop_blk (addr, len, type)  // remove control block
};
```

Figure A.2: The Recursive Buffer Class Definition

Appendix B

Queue Classes

```
class Simple_Queue {            // the simplest Queue
   QDHDR    head;               // doubly-linked queue pointers
   int      q_pos;              // current position in queue
   int      q_cnt;              // number of elements in the queue
 public:
   Simple_Queue();              // constructor
   ~Simple_Queue();             // destructor
   virtual int enqueue();       // add data elements
   virtual int dequeue();       // remove data elements
   virtual int seek_pos();      // move queue position pointer
   virtual int get_q_pos();     // get current position in queue
   virtual int get_q_cnt();     // get no. of elements in the queue
};
```

Figure B.1: The Simple Queue Class Definition

```
class Buffer_Queue : Simple_Queue {  // Qs for handling Buffers only
   // BufQ identification
   int   type;                // Queue type
   QID   qid;                 // global Queue identifier
   PID   owner;               // owner process of this Queue
   // BufQ status
   int   flags;               // flags
   int   status;              // status
   int   ref_count;           // reference counter
   virtual int signal();      // signal the receiver
 public:
   // operations
   Buffer_Queue();            // constructor
   ~Buffer_Queue();           // destructor
   virtual int enqueue();     // enqueue Buffer
   virtual int dequeue();     // dequeue Buffer
   // operations for getting BufQ fields go here
   virtual int get_type();       // get Queue type
   virtual QID get_qid();        // get qid
   virtual PID get_owner();      // get owner
   virtual int get_flags();      // get flags
   virtual int get_status();     // get status
   virtual int get_ref_count();  // get reference counter
};
```

Figure B.2: The Buffer Queue Class Definition

```
class Network_Queue : Buffer_Queue { // a subclass of Buffer_Queue
   Protocol protocol_descriptor;     // protocol descriptor
   virtual int signal();             // signal the receiver
 public:
   Network_Queue();                  // constructor
   ~Network_Queue();                 // destructor
   virtual int enqueue();            // enqueue network Buffers
   virtual int dequeue();            // dequeue network Buffers
   // protocol specific operations go here
};
```

Figure B.3: The Network Queue Class Definition

Appendix C

The BQP State Transition Table

The list of communication types, protocol states and actions are listed below. Figure C.1 is a table containing all possible states, events and actions. The rows are labeled by current states, and the columns by events (or the messages received). The intersections of rows and columns represent actions. (Due to space constraints, the events and their corresponding actions in the table have been wrapped around and stored in two rows for each current state.)

- Receiver Event/State/Action:
  - R_WDD : Wait for data from sender
  - R_REP : Wait for user reply
  - R_SDD : Send data to sender
  - U_REPLY : User reply
- Sender Event/State/Action:
  - S_DONE : Sender all done

	P_NUL	R_WDD	R_REP	R_SDD	U_REPLY ...
	S_DONE	S_SDD	S_WDD	U_SEND	X_ABORT
P_NUL	P_IGN	P_ERR	P_ERR	P_ERR	P_ERR ...
	P_IGN	P_ERR	P_ERR	S_INIT,S_SDD	X_ABORT
R_WDD	P_IGN	P_ERR	P_ERR	P_ERR	P_ERR ...
	P_ERR	R_WDD	P_ERR	P_CLR	P_CLR
R_REP	P_IGN	P_ERR	P_ERR	P_ERR	R_SDD ...
	P_ERR	R_REP	R_REP	P_ERR	P_CLR
R_SDD	P_IGN	P_ERR	P_ERR	P_ERR	P_ERR ...
	R_DONE	R_SDD	R_SDD	P_CLR	P_CLR
S_DONE	P_IGN	P_ERR	P_ERR	S_DONE	P_ERR ...
	P_ERR	P_ERR	P_ERR	S_INIT,S_SDD	P_CLR
S_SDD	P_IGN	S_SDD	S_WDD	S_WDD	P_ERR ...
	P_ERR	P_ERR	P_ERR	P_ERR	P_CLR
S_WDD	P_IGN	P_ERR	S_WDD	S_WDD	P_ERR ...
	P_ERR	P_ERR	P_ERR	P_ERR	P_CLR
X_ABORT	P_IGN	P_ERR	P_ERR	P_ERR	P_ERR ...
	P_ERR	P_ERR	P_ERR	P_ERR	X_ABORT

Figure C.1: The BQP State Transition Table

  - S_SDD : Send data to receiver
  - S_WDD : Wait for data from receiver
  - U_SEND : User initiated send

- Other Event/Actions:
  - X_QEXIST : Queue existence query
  - X_ABORT : Abort Buffer operation

- Internal State/Actions:
  - R_DONE : Decommission receiver
  - R_ACK : Ack S_SDD
  - R_INIT : initialize receiver
  - S_INIT : initialize sender
  - P_NUL : Null/Reset state
  - P_IGN : ignore/drop packet
  - P_CLR : clear transaction and retry
  - P_ERR : protocol error - reset and transmit Abort

Bibliography

[Abra73] N. Abramson and F. Kuo (eds.), "The ALOHA System", *Computer-Communication Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[Acce86] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Summer Conference Proceedings 1986*, USENIX Association, 1986.

[Ahuj88] S. Ahuja, N. J. Carriero, D. H. Gerlernter, and V. Krishnaswamy, "Matching Language and Hardware for Parallel Computation in the Linda Machine", *IEEE Transactions of Computers*, Vol. 37, No. 8, pp. 921–929, August 1988.

[Andr83] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming", *Computing Surveys*, Vol. 15, No. 1, pp. 3–43, March 1983.

[ATT85] AT&T, "System V Interface Definition", AT&T Customer Information Center, Indianapolis IN, Spring 1985.

[ATT87] AT&T, "UNIX System V Streams Programmer's Guide", Prentice-Hall Inc., Englewood Cliffs, NJ, 1987.

[Auer90] J. Auerbach, "TACT: A Protocol Conversion Toolkit", *IEEE Journal on Selected Areas in Communications*, Vol. 8, No. 1, pp. 143–159, January 1990.

[Bal88] H. E. Bal and A. S. Tanenbaum, "Distributed Programming with Shared Data", *Proc. of 1988 International Conference on Computer Languages*, pp. 82–91, Miami Beach, Florida, October 1988.

[Bal89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming Languages for Distributed Computing Systems", *Computing Surveys*, Vol. 21, No. 3, pp. 261–322, September 1989.

[Balz71] R. M. Balzer, "PORTS - A Method for Dynamic Interprogram Communication and Job Control", *Proc. of AFIPS Spring Jt. Computer Conf.*, AFIPS Press, Vol. 38, pp. 321–328, Atlantic City, NJ, May 1971.

[Bara85] A. E. Baratz, J. P. Gray, P. E. Green, J. M. Jaffe, and D. P. Pozefsky, "SNA networks of small systems", *IEEE Journal on Selected Areas in Communications*, Vol. SAC-3, No. 5, pp. 416–426, May 1985.

[Birm87a] K. Birman, T. Joseph, and F. Schmuck, "ISIS System Documentation, Release I", Technical Report 87-849, Dept. of Computer Science, Cornell University, July 1987.

[Birm87b] K. P. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, pp. 47–76, February 1987.

[Birm88] K. P. Birman and T. A. Joseph, "Exploiting Replication", Technical Report 88-917, Dept. of Computer Science, Cornell University, June 1988.

[Birr84] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39–59, February 1984.

[Bisi87] R. Bisiani and A. Forin, "Architectural Support for Multilanguage Parallel Programming on Heterogenous Systems", *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 21–30, Palo Alto CA, October 1987.

[Booc86] G. Booch, "Object-Oriented Development", *IEEE Trans. on Software Engineering*, Vol. SE-12, No. 2, pp. 211–221, February 1986.

[Camp74] R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions", *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 16, pp. 89–102, New York, 1974.

[Camp87] R. Campbell, V. Russo, and G. Johnston, "The Design of a Multiprocessor Operating System", *Proc. of the USENIX C++ Workshop*, pp. 109–125, Sante Fe NM, November 1987.

[Cart89] John. B. Carter and Willy Zwaenepoel, "Optimistic Implementation of Bulk Data Transfer Protocols", *Proc. 1989 ACM SIGMETRICS and PERFORMANCE '89: International Conference on Measurement and Modeling of Computer Systems*, ACM Press, pp. 61–69, Berkeley, CA, May 23-26, 1989.

[Cher84] D. R. Cheriton, "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, Vol. 1, No. 2, pp. 19–42, April 1984.

[Cher85] D. R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel", *ACM Transactions on Computer Systems*, ACM, Vol. 3, No. 2, pp. 77–107, May 1985.

[Cher86a] D. R. Cheriton, "Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design", *The 6th International Conference on Distributed Computing Systems*, IEEE Computer Society, pp. 190–197, Cambridge MA, May 1986.

[Cher86b] D. R. Cheriton, "VMTP: A Transport Protocol for the Next Generation of Communication Systems", *Proc. of the SIGCOMM '86 Symp. on Communications Architectures and Protocols*, ACM, pp. 406–415, New York NY, 1986.

[Cher86c] D. R. Cheriton and T. P. Mann, "A Decentralized Naming Facility", Technical Report 86-1098, Department of Computer Science, Stanford University, February 1986.

[Cox86] B. J. Cox, *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, Reading MA, 1986.

[Cyps78] R. Cypser, *Communications Architecture for Distributed Systems*, Addison-Wesley, Reading MA, 1978.

[Dasg87] P. Dasgupta, R. J. LeBlanc Jr., and W. F. Appelbe, "The Clouds Distributed Operating System: Functional Descriptions, Implementation Details and Related Work", Technical Report GIT-ICS-87/42, School of Information and Computer Science, Georgia Institute of Technology, 1987.

[Dijk65] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM*, Vol. 8, No. 5, pp. 569, September 1965.

[Dijk68] E. W. Dijkstra, "Cooperating Sequential Processes", *Programming Languages*, F. Genuys (Ed.), Academic Press, New York, 1968.

[Flei89] B. D. Fleisch and G. J. Popek, "Mirage: A Coherent Distributed Shared Memory Design", *Proc. of the 12th Symp. on Operating Systems Principles*, pp. 211–223, Litchfield Park, Arizona, December 1989.

[Flow90] K. Flowers, "Programming With the Motif Toolkit", *UnixWorld*, Vol. 7, No. 11, pp. 135–144, November 1990.

[Gele85] D. Gelernter, "Generative Communication in Linda", *ACM Trans. on Programming Languages and Systems*, Vol. 7, No. 1, pp. 80–112, January 1985.

[Gent81] W. M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", *Software-Practice and Experience*, Vol. 11, No. 5, pp. 435–466, May 1981.

[GM89] H. Garcia-Molina and A. Spauster, "Message Ordering in a Multicast Environment", *The 9th Int. Conf. on Distributed Computing Systems*, pp. 354–361, Newport Beach CA, 1989.

[Hama84] V. C. Hamacher, Z. G. Vranesic, and S. G. Zaky, *Computer Organization*, McGraw-Hill, New York NY, 1984.

[Hoar72] C. A. R. Hoare, "Towards a Theory of Parallel Programming", *Operating Systems Techniques*, Academic Press, pp. 61–71, New York, 1972.

[Hutc88] N. C. Hutchinson and L. L. Peterson, "Design of the x-Kernel", *Proc. of the ACM SIGCOMM '88 Symposium*, pp. 65–75, Stanford CA, August 1988.

[IBM86] IBM, "Token-Ring Network PC Adapter Technical Reference", 1986.

[IEEE83] IEEE, "IEEE Project 802, Local Network Standards", 1983.

[IEEE88] IEEE, "Portable Operating System Interface (POSIX) for Computer Environments", IEEE, 1988.

[Inte84] Intel, "LAN Components User's Manual", 1984.

[ISO84] ISO, "Information Processing Systems - Open System Interconnection - Transport Service Definition", 8072, International Organization for Standardization, 1984.

[ISO87] ISO, "Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1)", International Organization for Standardization and International Electrotechnical Committee, International Standard 8824, 1987.

[Koch89] S. G. Kochan and P. H. Wood, *UNIX Networking*, Hayden Books, Carmel IN, 1989.

[Krak88] S. Krakowiak, *Principles of Operating Systems*, The MIT Press, Cambridge MA, 1988.

[Lau86] F. C. M. Lau, "Policies and Mechanisms for Distributed Clusters", PhD Thesis, Dept. of Computer Science, University of Waterloo, 1986.

[Lau87] F. C. Lau, J. P. Black, and E. G. Manning, "Naming of Objects in the Cluster System", *Proc. IFIP WG 10.3 Working Conf. on Distributed Processing*, pp. 107–111, Amsterdam, October 1987.

[Leff88] S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman, "Interprocess Communication", *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison-Wesley, 1988.

[Li86] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems", *Proc. of 5th ACM SIGACT-SIGOPS Symp. of Principles on Distributed Computing*, pp. 229–239, Calgary Alberta, August 1986.

[Lian90] L. Liang, S. T. Chanson, and G. W. Neufeld, "Process Groups and Group Communications", *IEEE Computer*, Vol. 23, No. 2, pp. 56–66, February 1990.

[Lint87] M. A. Linton and P. R. Calder, "The design and implementation of InterViews", *Proc. of the USENIX C++ Workshop*, pp. 256–267, Santa Fe NM, November 1987.

[Mats88] S. Matsuoka and S. Kawai, "Using Tuple Space Communication in Distributed Object-Oriented Languages", *Proc. of the OOPSLA '88*, pp. 276–284, San Diego CA, September 1988.

[Metc76] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Vol. 19, No. 7, pp. 395–404, July 1976.

[Meye88] B. Meyer, *Object-Oriented Software Construction*, Prentice Hall International, New York, 1988.

[OMal90] S. W. O'Malley, M. B. Abbott, N. C. Hutchinson, and L. L. Peterson, "A Transparent Blast Facility", *Journal of Internetworking*, September 1990.

[Pete81] G. L. Peterson, "Myths about the mutual exclusion problem", *Inform. Procss. Lett.*, Vol. 12, No. 3, pp. 115–116, June 1981.

[Pete83] J. Peterson and A. Silberschatz, *Operating System Concepts*, Addison-Wesley Publishing Company, Reading MA, 1983.

[Pete87] L. L. Peterson, "Preserving Context Information in an IPC Abstraction", *Proc. of the 6th Symp. on Reliability in Distributed Software and Database Systems*, pp. 22–31, Williamsburg VA, March 1987.

[Pete90] L. Peterson, N. Hutchinson, S. O'Malley, and H. Rao, "The x-Kernel: A Platform for Accessing Internet Resources", *IEEE Computer*, Vol. 23, No. 5, pp. 23–33, May 1990.

[Post80] J. Postel, "User Datagram Protocol. Request For Comments 768", *USC Information Sciences Institute*, Marina del Rey, CA, August 1980.

[Post81] J. Postel, "Transmission Control Protocol. Request For Comments 793", *USC Information Sciences Institute*, Marina del Rey, CA, September 1981.

[Powe83] M. L. Powell and D. L. Presotto, "PUBLISHING: A Reliable Broadcast Communication Mechanism", *Proc. of the 9th Symp. on Operating Systems Principles*, pp. 100–109, October 1983.

[Radi90] S. R. Radia, "Names, Contexts, and Closure Mechanisms in Distributed Computing Environments", PhD Thesis, Department of Computer Science, University of Waterloo, 1990.

[Rama89] U. Ramachandran, M. Ahamad, and M. Y. A. Khalidi, "Coherence of Distributed Shared Memory: Unifying Synchronization and Data Transfer", *18th International Conference on Parallel Processing*, St. Charles, IL, August 1989.

[Reed78] D. P. Reed, "Naming and Synchronization in a Decentralized Computer System", Research Report MIT-LCS-TR-205, MIT, Cambridge, MA, September 1978.

[Ritc84] D. Ritchie, "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, pp. 1897–1910, October 1984.

[Ross89] F.E. Ross, "An Overview of FDDI: The Fiber Distributed Data Interface", *IEEE Jour. Selected Areas Communications*, Vol. 7, No. 7, pp. 1043–1052, September 1989.

[Russ89] V. Russo and R. Campbell, "Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques", Technical Report 89-13, University of Illinois at Urbana-Champaign, Urbana, Ill, April 1989.

[Sand90] R. M. Sanders and A. C. Weaver, "The Xpress Transfer Protocol (XTP) – A Tutorial", *Computer Communication Review*, ACM SIGCOMM, Vol. 20, No. 5, pp. 67–80, October 1990.

[Sche86] R. W. Scheifler and J. Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, pp. 79–109, April 1986.

[Sche88] R. W. Scheifler, "X Window System Protocol, MIT X Consortium Standard, X Version 11, Release 4", Laboratory for Computer Science, MIT, 1988.

[Shew90] Dave Shewchun, "A Streams Design for a Distributed Operating System", Term Report, Dept. of Computer Science, University of Waterloo, April 1990.

[Stan80] T. A. Standish, *Data Structure Techniques*, Addison-Wesley, Reading MA, 1980.

[Stro86] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, Reading MA, 1986.

[SUN87] SUN, "XDR: External Data Representation Standard", Request for Comments 1014, DDN Network Information Center, SRI International, June 1987.

[Swic88] R. Swick and M. S. Ackerman, "The X Toolkit: More Bricks for Building User-Interfaces or Widgets for Hire", *USENIX Winter Conference*, pp. 221–228, Dallas Texas, February 1988.

[Tam90] M. C. Tam, J. M. Smith, and D. J. Farber, "A Taxonomy-Based Comparison of Several Distributed Shared Memory Systems", *ACM Operating Systems Review*, Vol. 24, No. 3, pp. 40–67, July 1990.

[Tane85] A. S. Tanenbaum and R. van Renesse, "Distributed Operating Systems", *Computing Surveys*, Vol. 17, No. 4, pp. 419–470, December 1985.

[Tane88] A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Toku83] H. Tokuda, E. G. Manning, and S. R. Radia, "Shoshin OS: a Message Based Operating System for a Distributed Software Testbed", *Proc. of 16th Hawaii International Conference on System Science*, pp. 329–338, January 1983.

[Vasu87] R. Vasudevan, "Network Transparency in Multi-Process Structuring", Ph.D. Thesis, Dept. of Computer Science, University of Waterloo, 1987.

[Wats81] R. Watson, "Identifiers (Naming) in Distributed System", *Distributed Systems - Architecture and Implementation*, pp. 191–210, Springer-Verlag, 1981.

[Youn87] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System", *Proc. of the 11th ACM Symp. on Operating Systems Principles*, ACM SIGOPS, pp. 63–76, Austin TX, November 1987.

[Zimm80] H. Zimmermann, "OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection", *IEEE Trans. on Communications*, Vol. COM-28, No. 4, pp. 425–432, April 1980.

[Zwae85] W. Zwaenepoel, "Protocols for Large Data Transfers over Local Area Networks", *Proceedings of the 9th Data Communications Symposium*, pp. 22–32, Whistler Mt., BC, September 1985.

[Zwei90] J. M. Zweig and R. E. Johnson, "The Conduit: a Communication Abstraction in C++", *Proc. of 1990 USENIX C++ Conference*, San Francisco CA, April 1990.