A Generic Paradigm for Efficient Distributed Communication

James W. Hong and James P. Black
Department of Computer Science
University of Waterloo

July 29, 1991

Abstract

In our work, we seek to reduce the complexity of communication in distributed systems. We present the Buffer and Queue Model, which consists of 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 among various types of paradigms, abstractions, and entities. The Buffer-Queue protocol extends this consistent programming interface transparently to a distributed environment. We give examples of how various complex communication facilities may be developed using the Buffer and Queue paradigm.

1 Introduction

As computers are connected via networks to share resources and to increase performance, communication among them has become an essential part of modern computing. With the rapid increase in the power and capabilities of modern computers as well as in users' needs, communication software has mushroomed in response.

Over the years, various communication paradigms, abstractions and protocols have been developed independently to support specific system and application needs. As a consequence, today's programmers must deal with a complex, incompatible set of standards, semantics and interfaces. Currently, most communication software implementations are done by programmers using their own *ad hoc* approaches, yielding a diverse and unwieldy programming environment. In such an environment, there is a desperate need for a single consistent and efficient communications programming interface, which can be used by all types of entities for all types of communication, and which can be extended transparently across a distributed system.

Recently, there have been some attempts to provide a single consistent programming interface to reduce the complexity of such programming. AT&T STREAMS [ATT87], BSD sockets [Leff88], the *x*-Kernel [Hutc88], Choices Conduits [Zwei90] and TACT [Auer90] are some such attempts. Unfortunately, most of these efforts fail to support new forms, or as many forms, of communication as one might wish. In our estimation, the main reason for this failure is that most have *a priori* targeted a specific or limited set of areas and thus are not based on sufficiently general communication abstractions.

We address the communication problem in distributed systems by seeking answers to the following three related questions:

- How can a programmer write code to communicate with the provider of a 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 we achieve this independence even when the communicating entities have different characteristics?

In this paper, we present the Buffer and Queue Model as an efficient and versatile solution to the problem, and as a specific implementation of the generic communication model developed in [Hong91]. We use the object-oriented design framework [Meye88] in the development of the model, since it allows us to define or specialize the tools as needed. The Buffer and Queue Model contains a set of communication concepts and primitives which is simple and general, rigorous and flexible, low-level and extensible. We show how this model uses the efficiencies of a single memory domain while providing a universal communications interface (called the *internal communications interface*) between various types of entities across a wide spectrum of environments. The internal communications interface can be used to replace *ad hoc* approaches currently being employed between various entities, high-level abstractions and implementation mechanisms to provide users with a single consistent programming interface.

The rest of the paper is structured as follows. Section 2 lists a set of design constraints for a simple, efficient and versatile communication paradigm. Section 3 presents the Buffer and Queue Model. Section 4 presents the Buffer-Queue Protocol, which extends the internal communications interface transparently to a distributed environment. Section 5 demonstrates how various complex communication facilities may be easily and efficiently developed using Buffers and Queues and Section 6 presents some conclusions.

2 Design Constraints

We define *communication* as the transfer of data between two or more entities. The definition involves three fundamental concepts: data, entities, and transfer (delivery and synchronization).

A survey of various communication paradigms in distributed systems and a detailed discussion of important issues related to the fundamental concepts of communication are given in [Hong91]. Based on this discussion, design constraints are derived for a simple, efficient and versatile communication model that provides a single consistent communications programming environment. Some of those constraints are:

- a standard data-object structure is required, which all levels and types of communication can use efficiently and consistently;

- a universal communications interface is required, which can be used between various types of entities across a wide spectrum of environments;
- for efficiency reasons, the model is required to use the conceptual efficiencies of a single memory domain by passing a descriptor (or control) of data objects rather than data objects themselves;
- for resource management and other purposes, a data object must always be returned to its owner at completion of the operation;
- an identification scheme that can uniquely identify data or interface objects across a system is required; and
- for transparent distribution of data objects, a protocol is required to transfer a structural description of data objects as well as control information for peer-to-peer interactions.

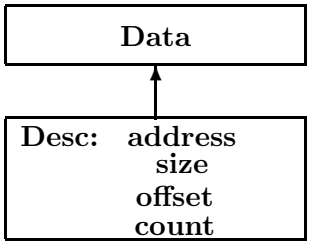
3 The Buffer and Queue Model

We have applied the design constraints mentioned in the previous section and developed an efficient and versatile communication model called the Buffer and Queue Model, which consists of three communication abstractions, namely delivery/synchronization, Buffers, and Queues, corresponding to the transfer of data among entities.

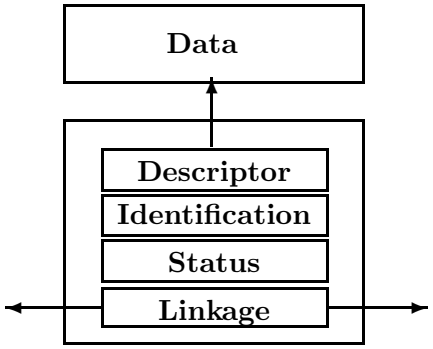
3.1 Buffers

A Buffer is an abstraction of a memory object consisting of an ordered sequence of bytes. The simplest form of Buffer is one that describes a single contiguous block. A Buffer consists of two parts: data and operations. The data portion of a Buffer is referred to as the Buffer descriptor or *Bufd*.[1] It 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. Operations required in a Simple Buffer (shown in Figure 1 (a)) write data into and read data from Buffers, and set and return Bufd information.

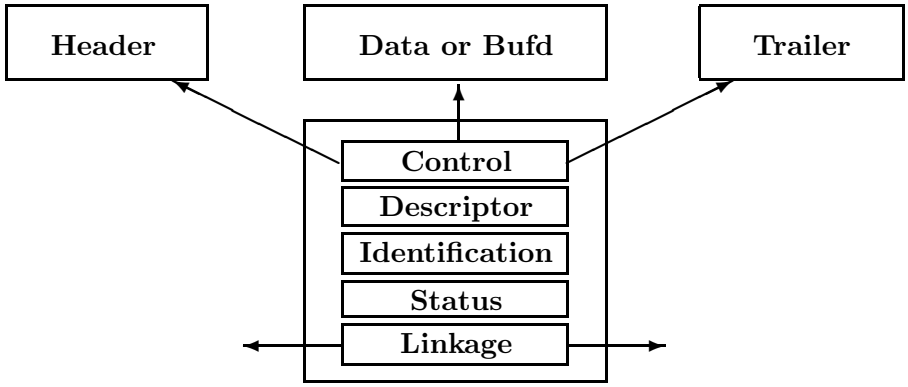
[1]Throughout this paper, when we use the term *Buffer*, we will mean the object in the object-oriented sense [Cox86], while by *Bufd*, we will mean only the area of memory occupied by the data corresponding to a particular instance.



(a) A Simple Buffer



(b) A Segmented Buffer



(c) A Recursive Buffer

Figure 1: Buffer Structures

Communication based on Buffers and Queues involves transferring Buffers among Queues. This requires that extra information be maintained in Bufds in addition to data, namely identification and status information. Identification fields include type, identifier and owner of a Buffer, as well as where the Buffer should be returned at the completion of an operation (the *returnQ*). Status fields include the present location (a Queue), and the status of the most recent operation. During communication, messages may be fragmented into smaller pieces and reassembled. Thus, a pair of pointers is used for handling a sequence of Bufds as a doubly-linked queue. The structure of a Segmented Buffer is given in Figure 1 (b).

The basic operations defined in the Simple Buffer can be inherited by the Segmented Buffer. However, they need to be redefined to handle a list of memory blocks. Moreover, operations to fragment a large data item into two smaller pieces of data and to reassemble them back into a single large data item are required.

A data structure such as this Segmented Buffer has been demonstrated to be inefficient for protocol processing [Hutc88, Zwei90]. 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 indeed, (a pointer to) another Bufd. Such a structure, shown in Figure 1 (c), can handle arbitrary layers of protocol headers and trailers. Operations on recursive Buffers include *push_blk* and *pop_blk*. *Push_blk* is used in packetization to insert a header or trailer by linking the appropriate Bufd pointers to either block. *Pop_blk*, on the other hand, is used in depacketization to separate a header or trailer from user data. Further, the *read* and *write* operations need to be modified to traverse a recursive Bufd structure and access individual blocks.

The recursive Buffer structure and functional interface can be used as a "standard" structure for all levels and types of communication. If all levels use this common structure, a uniform, simple and efficient software interface can be designed for each layer that permits complex interactions without regard for details of any higher or lower layer interfaces. Further, the framework that we have used to develop hierarchical Buffer structures can be used to modify existing or develop new Buffers as needed.

3.2 Queues

Queues are the second fundamental concept in our model: the endpoints of communication which represent the entities exchanging information. We use the conventional concept of the queue abstract data type. The simplest queue consists of a pair of pointers for storing data objects, and the operations *enqueue* (to add objects) and *dequeue* (to remove objects). Although FIFO is the normal access discipline for queues, we assume other access modes will be useful, for example, to extract fragments of a single message from a Queue holding interleaved fragments of a number of messages, or for handling priority data. A position cursor and a data-object counter are useful information to maintain in a Queue for efficient access to data objects.

We call the specialization to a queue of Buffers a *Buffer Queue* (or *BufQ*). In addition to queue pointers, identification and status information is needed in BufQs for system management and communication. Identification fields include the type, identifier and ownership of a Queue. Besides specifying the type of a Queue, the type field may be used to indicate the delivery method (*e.g.*, via procedure call, wake-up signal). Status fields include a count of the number of Bufds using this particular BufQ as their returnQ, and flags for various purposes (such as whether a process is blocked on a dequeue operation of a Buffer from a Queue).

Network communication usually involves a Buffer being passed through multiple layers of communication protocols. At each layer, protocol-specific processing is performed such as manipulating control information and updating state information. Since the control information and the operations for manipulating it are stored within a Buffer, it is the Buffer that gets modified as it travels downward or upward. However, state information to manage each protocol layer should not be stored in the transient Buffer but rather in endpoints or protocol modules. Thus, a Network Queue is a specialized Buffer Queue that can potentially include a protocol-specific socket structure.

The universal interface (*i.e.*, a Buffer Queue) developed in this section can be used between various types of entities. It provides generic delivery and signalling techniques and a storage capability for data objects (Buffers). The framework that we have used to develop hierarchical Queue structures can be used to tailor existing Queues or develop new Queues as needed.

3.3 Delivery and Synchronization

The *enqueue* and *dequeue* operations of Queues are used for delivery of Buffers. 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 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 can be implemented differently in different communication paradigms. For example, the signal function might simply be a call instruction in the procedure-call paradigm, or a wakeup call for a blocked process in the message-passing paradigm. 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.

Transferring a Buffer is achieved by enqueueing a Buffer onto the BufQ that represents the destination entity. Receiving a Buffer is achieved by dequeueing it from a BufQ. Note that Buffer flow and ownership are orthogonal to data flow: there is no distinction in the model between full and empty buffers, but rather, an emphasis on the management of the memory areas containing the data.

Thus, a Buffer may be enqueued on any BufQ in the system. However, we restrict dequeue operations to the owner of a BufQ. This is a reasonable restriction which serves to maintain order in Buffer management. If more flexible enqueue and dequeue semantics are required, one can build that capability 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 such issues as resource management, synchronization, demultiplexing of long messages and other interference aspects.

The Buffer and Queue Model provides a simple but versatile mechanism for returning data delivery or Buffer operation status. The returnQs coupled with the capability to access nested higher-layer Bufds inside a standard Buffer from any level provide an elegant solution. Any receiver can either return the Bufds in the reverse direction of the delivery path (*i.e.*, removing the Bufd it created and passing the rest to its sender) or return them

all directly to their 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 since each Bufd always contains its own returnQ information. Finally, returnQs are useful for resource management and synchronization.

By convention, a Buffer is blocked (*i.e.*, the user does not have access to it) when it is enqueued on some BufQ other than the sender's or it is in the control of (has been dequeued by) 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 can be reused or reclaimed.

Using this convention, we can easily support both synchronous and asynchronous communication, which are the basic forms of communication required in most modern computer systems. The Buffer and Queue communication paradigm is inherently asynchronous: the user invokes enqueue to initiate the operation, and then resumes its execution. When it wants to discover the status of the operation, it can simply poll or block on the dequeue operation of the Buffer from its returnQ. In synchronous or blocking communication, the sender must enqueue the Buffer to an appropriate Queue and then block immediately on the dequeue operation of the Buffer from its returnQ.

Although the principle of returning the Buffer to its sender when the operation is complete is intended mainly to assist Buffer resource management, it can be used as a vehicle for several other useful mechanisms. For instance, it can be used to transport acknowledgements back to the requesting entities upon completion or failure of an operation. In turn, this results in the signal function being invoked, which can be used to unblock any blocked user entities. Another use is in synchronization of data control; the return of a Buffer signals that it can be modified without fear of disrupting some communication in progress.

4 The Buffer-Queue Protocol

The Buffer and Queue Model presented thus far views a system as a single node, where efficiencies of a single memory domain can be enjoyed. That

is, from the user's point of view, communication takes place within a single global domain. In reality, however, there are a number of separate nodes and protection domains in such a system and we must provide some tools to bridge this gap if we wish to provide a truly transparent distributed Buffer and Queue paradigm.

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. However, communication across protection domains requires a *relay* or a *pipeline* through which the data must be copied from the sender's protection domain to the receiver's. In order to provide transparent Buffer and Queue communication across protection domains (both within a single node and between nodes), the structural information of Buffers as well as control information for Buffer and Queue operations should be delivered along with the content. The Buffer-Queue protocol (BQP) is intended to do just this. Note that the data structure encoding (which we call *linearization*) as opposed to the control aspects of BQP could easily be replaced by external data representation standards such as XDR [SUN87] or ASN.1 [ISO87] at the possible expense of some efficiency.

Between protection domains within a single node, a privileged entity such as the kernel performs the copy operation (both structure and content) on behalf of the sender and receiver. The kernel can employ memory management "tricks" such as memory mapping [Russ89] to improve the efficiency of such operations. Across nodes, however, no such third party exists: the structural information must be encoded and transmitted along with the content by the sender, and there must be some entity on the remote node that can decode and deliver it to the appropriate receiver.

In a layered communication system, an implementation of BQP would result in a separate, thin layer in a stack of protocol layers. Since Buffers must be transferred reliably between nodes, the BQP must either reside on top of a reliable transport protocol or have the reliability aspect built into the protocol itself. We chose to build the reliability features (*e.g.*, sequencing, error recovery) into the protocol so that it can be placed at an arbitrary level in a protocol stack. These features only need to be activated when necessary.

In the Buffer and Queue Model, the structure of a Buffer is contained in a Bufd or nested Bufds. Thus, the BQP transfers structural information contained in higher-layer Bufds to the remote node. The remote BQP module uses this information to reconstruct the original Buffer, with its internal

structure intact, so that it can be delivered within the remote node as if enqueued locally. A Bufd also contains the operational information such as the returnQ and status as the requested operation completes. This information must also be transferred along with the structural information. We refer to these two types of information (structural and B-Q operational) as *essential* Bufd information that needs to be transferred in order to support the single-domain communication paradigm across the system.

However, the Bufds also contain information that is useless in remote nodes (*e.g.*, local virtual memory addresses), and which need not be transmitted. One alternative 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 unused data, we prefer it because of its simplicity and uniformity, and because the receiver need not re-allocate any extra memory for Bufds, but may simply strip them from the inbound packet and overwrite fields as appropriate.

The remaining problem is how 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 Bufds themselves. As discussed at length in [Hong91], we assume that low-level packet assembly is modified to also include Bufds marked for transmission – specifically those 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.

In summary, we claim that the Buffer and Queue Model coupled with the BQP 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 section.

5 Communication Examples Using Buffers and Queues

In this section, we demonstrate how one can easily and simply implement complex communication facilities using Buffers and Queues. We first show a simple example of internal communication, which we then extend to implement distributed communication which involves BQP on top of conventional network communication.

As stated earlier, *internal communication* refers to communication between entities within a single protection domain, where messages are passed by reference and copying of messages is avoided as much as possible. Our internal-communication example, shown in Figure 2, involves two entities, A and B. A receiveQ associated with the entity B is a BufQ onto which incoming Bufds are enqueued and where they remain until dequeued by the owner of the BufQ. 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.

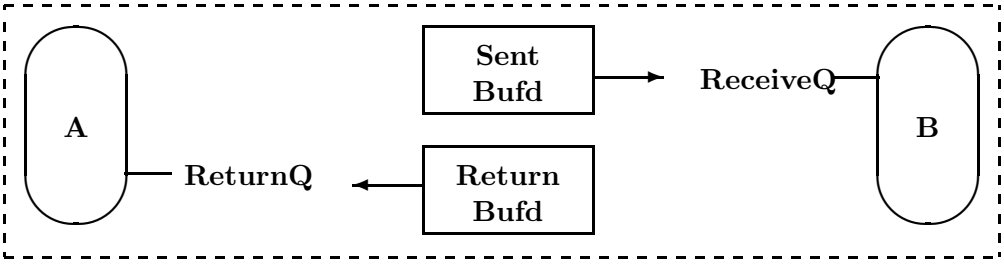


Figure 2: An Example of Internal Communication

Entity A transfers a Bufd to entity B by enqueueing it onto B's receiveQ, and invoking its *signal* function as a side effect. Entity B retrieves the Bufd 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 Bufd from its returnQ and examining the *return_status* flag. Since this internal communication example takes place

within a single protection domain, neither the structural information nor the content of the Buffer need to be copied.

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 communicate by procedure call, however, the enqueue and signal functions become asymmetric and more complex.

Consider a procedure A (the calling procedure) invoking a procedure B. We assume that there are some parameters transmitted by A to B and results returned from B to A. Further, we assume that the process of saving and restoring the context of the calling procedure will be handled by the normal procedure-call mechanism. 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 B. The signal function invokes the dequeue operation of B's receiveQ to obtain the input parameters, before finally calling procedure B. After the execution of B, the results are appropriately stored in the Buffer again and returned to the calling procedure by enqueueing on A's returnQ. This enqueue operation also calls the signal function, but that version of signal is a null operation: the effect of the signal is achieved through the normal procedure call mechanism as a number of stack frames are popped. Execution continues at the next statement in A, which dequeues the returned Buffer to access values sent by B.

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, which would return the Buffer with an appropriate error status; this status 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 following, more elaborate example.

To demonstrate how Buffers and Queues can be used across protection domains within a single node (*i.e.*, local IPC between two virtual address spaces) and across nodes (*i.e.*, remote IPC), consider a remote X11 [Sche88] client-server communication example. An X11 client wishes to send a request to a server located on another node on the network. It creates an X11 Buffer and fills it with the appropriate information such as the request op-code and its arguments, and 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. We also assume the use of a stack of TCP, IP and Ethernet protocols under the IPC layer.

Presumably, the X library routine requests the kernel to create an IPC Buffer. The X library routine prepares the IPC Buffer and passes it to the IPC server by enqueueing it on the server's BufQ. Note that a typical IPC server would be equipped with two BufQs, one for storing Buffers containing messages to be sent (sendQ) and another for storing Buffers that are to receive arriving messages (receiveQ). At this point, the server checks whether the Buffer is destined to a remote or local entity. If it is local, the server would try to find a match from Buffers that are enqueued onto its receiveQ, transfer data to the receiver's Buffer and complete the local IPC by returning the Buffers to their corresponding returnQs. In this case, the IPC server acts as the relay between two protection domains by copying the data from the sender's protection domain to the receiver's. The user processes communicate with the IPC server's queues through a kernel (procedure) call, while the kernel returns Buffers to the user processes through blocking queues associated with the process control blocks.

If the destination is remote, the IPC server must transfer the Buffer to the remote IPC server. This is an example of a pipeline between nodes. Since there is no third party that can assist in relaying Buffers as in the local IPC example, the peer BQP modules are responsible for linearizing and delinearizing, transferring Buffers reliably, and synchronizing the parties involved. Thus, the IPC server passes the Buffer to its lower layer, the BQP layer, again by enqueueing it to the BQP server's sendQ. In this case, the IPC and BQP servers may be special system processes communicating in a shared-memory domain through kernel Queues. The IPC servers on the two systems use the BQP to relay between their disjoint memory domains.

The BQP server dequeues the Buffer and fills in the protocol information (such as source and destination QIDs, total message length, number of Bufds, *etc.*) in the BQP header. Since the BQP sits on top of a reliable stream (TCP in our example), activating the reliability features (*e.g.*, sequencing, checksum) is unnecessary unless BQP is multiplexing several concurrent operations and thus fragmenting and sending Buffers piecemeal itself.

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 using internal communication between layers whenever possible. In this way, the Buffers are passed by reference from the user layer to the device layer, avoiding copy operations. The resulting Buffer structure before transmission is shown in Figure 3. In conventional network communication, the transmission traversal scheme would involve visiting all Bufds and transmitting only the headers, data and trailers but not Bufds. In distributed Buffer and Queue communication, however, Bufds above the BQP layer must also be transmitted as explained earlier. The corresponding Ethernet packet transmitted is shown in Figure 4.

After the Ethernet controller transmits the packet to the remote node, the Bufds in the Ethernet Buffer can be returned recursively to higher layers, or directly to individual returnQs by the Ethernet or any intermediate layer as appropriately flagged. 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 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 of the relevant Bufds will be caught and redirected to the remote node.

The BQP server then passes the IPC Buffer to the IPC server by enqueueing it to the IPC 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

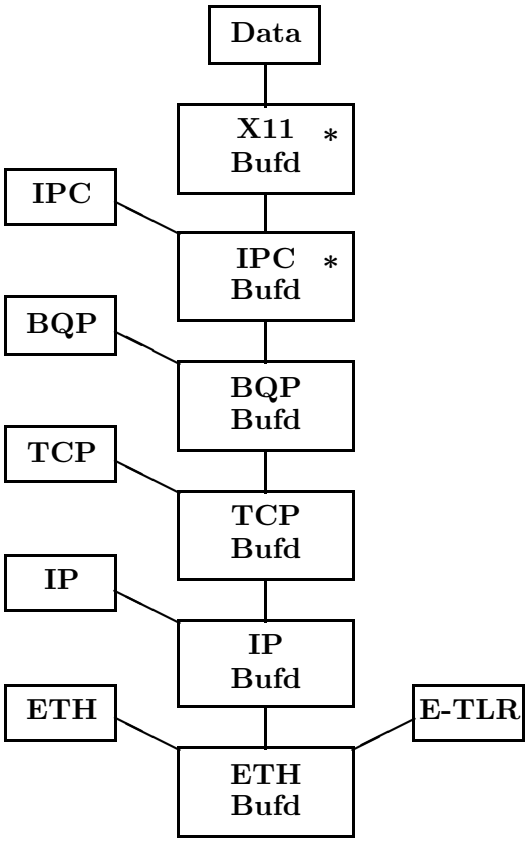


Figure 3: The Internal Structure of an Ethernet Buffer with the Flagged Bufds

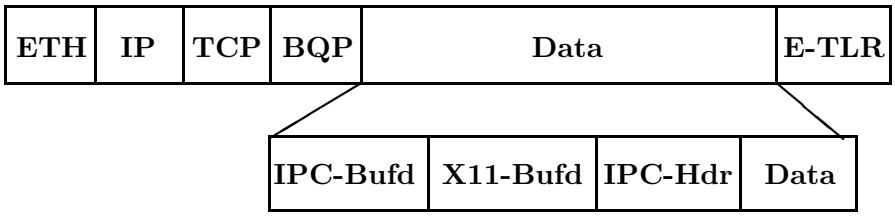


Figure 4: The Transmitted Ethernet Packet Containing Flagged Bufds

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 clean-ups carried out.

This example demonstrates not only how one can implement conventional network communication efficiently, but also how different types of entities (*e.g.*, user processes, kernel routines, kernel processes, devices) can communicate using a uniform data structure as well as a uniform communications interface. Further, the BQP extends the internal communication interface to a distributed environment.

6 Conclusion

Our work has focused on reducing the complexity of communications programming in distributed systems. Currently, programmers use *ad hoc* approaches in developing and implementing communication paradigms and protocols. Our main goal is to develop a set of simple standards and tools for a uniform, efficient and consistent programming interface that can be used to implement various communication interactions both within and among different paradigms, abstractions, and entities.

As a simple and efficient solution to the communication problem, we propose the Buffer and Queue Model, which meets the constraints of the generic communication model developed in [Hong91] and thus provides a single consistent programming environment which uses the conceptual efficiencies of the local-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. We demonstrated the generality and versatility of the Buffer and Queue Model by presenting implementation examples of various existing communication forms. Additional examples we have investigated include the use of Buffers and Queues for conversion to and from stream- and message-based communication, for efficient processing of long messages in a distributed environment, for optimistic blast protocols, and for one-way (broadcast) communication.

Using this model, the communication 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.

The main problem we intended to solve in our work was to find answers to the set of questions raised in the introduction. Consider the following code segment.

```
simple_op() {
       Buffer_Queue  *Dest_Q, Return_Q;      // create Qs
       Simple_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
}
```

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 its returnQ. This code using Buffers and Queues is independent of memory domain, execution control regime, physical location of the service and types of entities. The client and server can be 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.

References

[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, Atlanta, GA, 1986.

[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.

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

[Hong91] James W. Hong, "Communication Abstractions for Distributed Systems", *PhD Thesis*, Research Report CS-91-43, Dept. of Computer Science, University of Waterloo, 1991.

[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.

[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.

[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.

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

[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.

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

[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.

[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.