Reducing Communication to a Buffer and Queue Model

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

March 6, 1991

Abstract

In our work, we seek to reduce the communication problem to its critical concepts and build a simple, efficient, and general communication paradigm based on these concepts. We present the Buffer and Queue Model, which contains a set of communication abstractions and primitives which is simple and general, rigorous and flexible, low-level and extensible. We show how this model seeks to utilize the efficiencies of shared memory communication while providing a universal communications interface between various types of entities across a wide spectrum of environments. We give examples of how various complex communication facilities may be developed from this low-level communication system.

1 Introduction

As computers have grown more powerful, their capabilities have become more complex and system software has mushroomed in response. Such growth has been driven both by the increasing variety and sophistication of the hardware interface, and by the devolution of application level software packages into standard interfaces for yet higher layers of applications software [Sche86]. 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.

The modularization of such software was initially typified by the concept of layered structure as embodied in the ISO-OSI Reference Model for network protocols [Zimm80]. As the move to hardware independence and open systems became stronger, complete procedural definitions of layers and standard interfaces have emerged, such as the System V Interface Definition [ATT85] and similar POSIX standards [IEEE88]. More recently, a need for dynamic flexibility in the type and arrangement of layers has suggested the concept of a much more self-contained and independent module. AT&T STREAMS is an example of such a concept [ATT87].

However, as system components become more modular and independent so that even the hardware and software base of the system is being changed regularly, 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 between modules becomes a more significant component of such tasks, efficiency coupled with versatility become ever more important in such communication.

Over the years, many researchers have worked on various aspects of communication, some in developing new communication paradigms or protocols, others in attempts at improvements or performance optimizations. Recently, there have been some attempts to provide a single consistent programming interface that bridges many underlying techniques, which would reduce the programmers' concern of dealing with the conceptual partitioning of interfaces. 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 have targetted a specific or limited application area, often without a base of general communication abstractions to start with.

In this paper, we present the Buffer and Queue Model, which is a specific implementation of the generic communication model developed in [Hong91]. 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 seeks to utilize the efficiencies of shared memory communication while providing a universal communications interface between various types of entities across a wide spectrum of environments. Further, we show how various complex communication facilities may be developed from this low-level communication system.

2 Communication Issues in Distributed Systems

We define *communication* as the transfer of data between two or more entities. The definition involves three important concepts: data, entities, and transfer (delivery and synchronization). In this section, we identify a number of issues associated with each concept in order to motivate our Buffer and Queue model.

2.1 Data

Data (or messages) may be passed from one entity to another either by reference (*i.e.*, a pointer to the data is passed) or by value (*i.e.*, a copy of the actual data is transmitted). Passing by reference, realized in a shared memory environment, is more efficient both in time and space since it does not involve operations on individual bytes of data. Passing by reference also allows the structure of data (no matter how complex) to be preserved through the communication. However, the same does not necessarily hold true when data must be copied because communication occurs between disjoint protection domains. When copying data, its structure must be understood by the copier, which often limits such structures to a single contiguous sequence of bytes. More elaborate schemes can require an extra protocol layer to provide a structural description of the data to be transferred, which is necessary if a shared-memory communication paradigm is to be provided across the system.

Most communication implementations involve multiple protocol layers with control information (*i.e.*, headers and trailers) associated with the user data for each protocol. Flexibility 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 from one layer to another. Such a structure should also allow data fragments to be reassembled at an arbitrary layer,[1] which can result in less strain on system resources.

Buffer memory management refers to management of memory resources containing message components such as user data, protocol headers and trailers, control data, and descriptors that describe any of these message components. The low-level aspects of memory management, such as dealing with how and when memory is allocated and deallocated should not be the concern of communication software but rather of the user of the communication subsystem. However, the communication subsystem must support memory management conventions that allow the efficiencies of shared memory to be exploited by avoiding copies wherever possible, and minimizing the need for expensive dynamic memory allocation and deallocation. For example, the communication subsystem should return control of memory to the owner when it is no longer needed. The principle involved is that owner of the memory should be the only entity that can destroy it and that destruction should only take place when control is returned. This is a reasonable and necessary restriction needed to maintain order in resource management.

2.2 Communicating Entities

Two environments are involved in most communication paradigms: a *working environment* and a *target environment*. The working environment refers to an environment where the communications processing takes place. In the working environment, communications processing usually takes place within a system protection domain involving global shared memory. The target environment, on the other hand, may be either local (*intra-node*) or remote (*inter-node*). Intra-node communication, where the working and target environments are the same, generally involves mostly software and primary storage in a single or few layers. On the other hand, inter-node communi-

[1]Conventional peer-to-peer protocols require that the data fragments be reassembled at the same layer on the receiving side as they were fragmented on the sending side.

cation, where the working and target environments are different, generally involves hardware with protocol-specific device interface and more protocol layers.

In this kind of communication environment, there is a need for a universal interface between all entities whether in the working or target environment, an interface that can be used by different types of partners to achieve simple and efficient communication. In particular, there is a need for a universal interface between internal communication, employing the efficiencies of shared memory, and network communication. The interface each layer presents in network communication should be that of internal communication. Hence, all layers could be linked in a uniform fashion and we could easily interface directly to any sub-layer in network communication (*e.g.*, to TCP, IP, *etc.*) from any other. Distributed communication, which may involve multiple remote machines, can still enjoy some of the efficiencies of shared memory if the internal communication interface can be naturally extended to remote machines.

Another important aspect related to communicating entities is *connectivity*. Connection-oriented communication (*e.g.*, streams) involves intermediate entities which form a logical pipeline to transfer data between two communicating endpoints. The construction of such pipelines should be flexible so that they can be setup either at system configuration time or at runtime. Thus, standard operations and interfaces that will connect entities together and disconnect them on demand must be provided. While these operations are not necessarily required in connection-less communication (*e.g.*, datagram), in both types routing information is implicitly or explicitly maintained within each endpoint. Even when no logical communication connections are present, note that there is a need for "connection" between the different protocol layers that may be involved in providing, for example, remote procedure call or datagram communication.

2.3 Delivery and Synchronization

Data delivery techniques vary greatly in different communication paradigms. Some examples of delivery techniques include transmission via stacks or globally referenced variables in procedure calls, writing into and reading from shared memory in shared memory IPC, or copying data from the source to destination in communication between disjoint protection domains. Closely

associated with data delivery is a signalling mechanism for transferring the control of data as well as notifying the receiver of the arrival of data. This is necessary for all delivery techniques, since the receiver must have some indication that data has arrived and is available for access. As with 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 delivery and signalling techniques for various communication paradigms, a generic delivery and signalling technique is desirable.

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, an acknowledgement or status is expected on completion of the operation. Even when buffers (or data objects) that carry user data and control information travel only in one direction, owners of these data objects usually expect control of the data objects to return to them when the operation is complete. Thus, if we can utilize the principle of always returning (control of) data objects to their owners when the operation is complete, the operation status can be returned at little or no additional cost by piggybacking the status on the returning data object.

Synchronization in communication can be broken down into two aspects: synchronization of user execution (or user blocking semantics) and synchronization of data object control (or blocking semantics of data objects). That is, when is it safe for the owner to dispose, modify or reuse the data object? In synchronous communication the user is blocked while waiting for the completion of the operation and return status. Since the user is blocked, the user does not have access to the data object and hence the contents of it are quite secure. In asynchronous communication, however, the user is not blocked after initiating the data transfer, and thus can potentially access the data or modify it accidentally.

3 The Buffer and Queue Model

3.1 The Buffer Abstraction

In this section, we present details of the Buffer abstraction and the associated solutions to various communication problems related to data. We also develop an efficient, versatile Buffer structure that all levels of communication can use.

3.1.1 Generic Buffer

A Buffer is an abstraction of an area of memory. The simplest form of Buffer is one that describes 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.[2] It consists of four elements: the starting address of a block of memory used by the Buffer, the size of Buffer, and the starting and ending addresses of valid data. Having two variables to specify the valid data as opposed to having just a single counter is useful when the data fragments may be written and read concurrently, as in the *bounded-buffer problem* [Pete83]. The basic operations common to all Buffers write data into and read data from Buffers, and set and return Bufd information. A generic Buffer is shown graphically in Figure 1 (a).

3.1.2 Simple Buffer

Communication based on Buffers and Queues involves transferring Buffers among Queues. This requires extra information be maintained in Bufds in addition to data, namely identification information and status information related to Buffer operations. Identification fields include *type*, *bid*, *owner* and *returnQ*. Since we envisage various types of Buffers for various types of communication, we need to specify the type. *Bid* uniquely identifies the Bufd across the system under consideration. The *owner* field specifies who created the Bufd. *ReturnQ* specifies the Queue to which the Buffer is supposed to return when the requested operation is complete. Status fields include

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

currentQ and *return_status*. *CurrentQ* points to the present location of the Bufd. *Return_status* contains the status of the most recent operation on the Bufd. The Bufd also contains a pair of pointers, *q*, that are used when Buffer objects are attached to some Queue. These pointers can also be used to link together a chain of Bufds to form a more complex Buffer, for example, to support message fragmentation and reassembly. The structure of a simple Buffer is given in Figure 1 (b).

3.1.3 Recursive Buffer

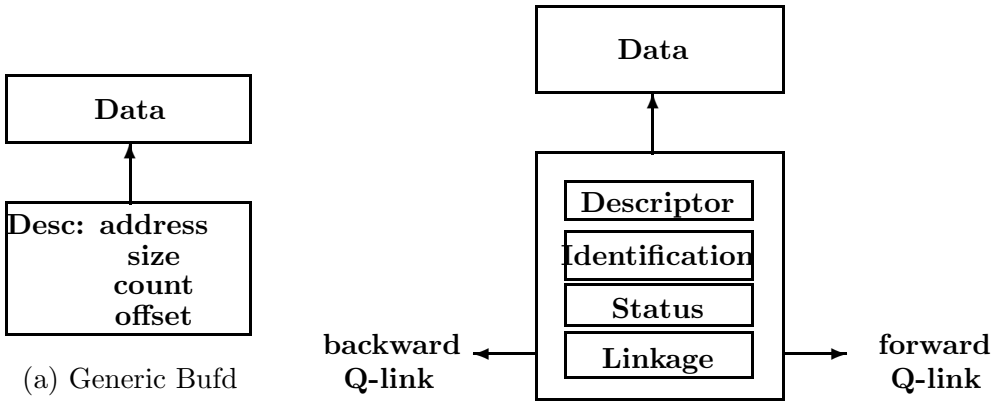
A data structure such as the simple Buffer presented above has been demonstrated to be inefficient for protocol processing [Hutc88, Zwei90]. A more desirable structure is one that can handle a hierarchy of Bufds, each Bufd capable of describing multiple blocks of memory for header, trailer and data (or another Bufd). Such a structure is shown in Figure 1 (c). It contains several pointers and a flag. The first points to a protocol header field, the second to a protocol trailer field, and the third to either data or another Bufd (*i.e.*, recursive Bufd structure). The flag in the Bufd indicates which memory blocks exist and specifies whether this Bufd contains a data block or a pointer to another Bufd.

The recursive Bufd 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. If all levels use this common structure, a uniform and simple software interface can be designed for each layer that permits complex interactions without regard for details of any of its higher or lower layer interfaces.

The framework that we have used to develop hierarchical Buffer structures above can be used to modify existing or develop new Buffer structures as needed. For example, Buffers for various network communication protocols (*e.g.*, TCP Buffers, IP Buffers) can be created by adding protocol-specific data structures and operations to the basic recursive Bufd structure.

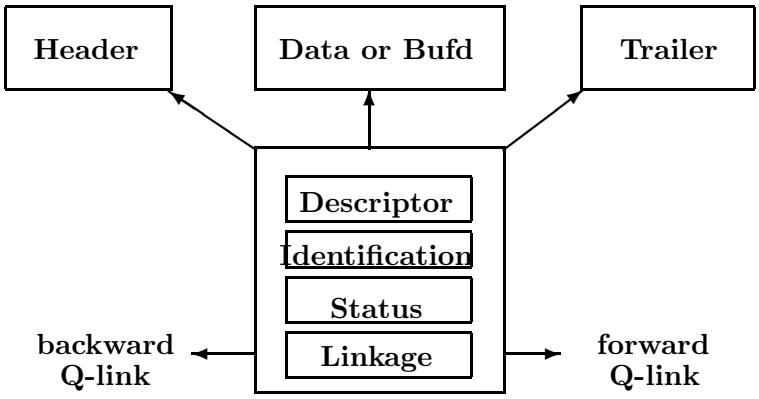
3.2 Queue Abstraction

In this section, we present the details of the Queue abstraction and the solutions to various problems related to communication interfaces. We also



(a) Generic Bufd

(b) Simple Bufd



(c) Recursive Bufd

Figure 1: Buffer Structures

present the development of a generic interface that all communicating entities can use.

3.2.1 Simple Queue

We use the conventional concept of the queue abstract data type. The simplest queue consists of a pair of pointers, head and tail, and the operations *enqueue* (to add elements) and *dequeue* (to remove elements). Using the common convention that elements are added to the tail of queues and are deleted from the head (*i.e.*, FIFO), one can order or sequence pieces of data being delivered. Although FIFO will be the accepted access discipline in Queues, we assume other access modes will be useful in such operations as extracting fragments of a single message from a Queue holding interleaved fragments of a number of messages.

3.2.2 Buffer Queue

We call the specialization of queues to a queue of Buffers a *Buffer Queue* (or *BufQ*).

In addition to queue pointers, extra information is needed in BufQs for system management and communication, namely identification and status information. Identification fields include *type*, *qid* and *owner*. *Type* specifies the type of BufQ. *Qid* uniquely identifies the BufQ across the system under consideration. The *owner* field specifies who created the BufQ. Status fields include *flags* and *status*. *Flags* can be used for setting various bit options (such as whether a process is blocked on a dequeue operation of a Buffer object from a Queue). The *status* field contains the most recent status of a Queue operation.

In our Buffers and Queues 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, 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 resource management, synchronization, demultiplexing of long messages and other interference aspects. We do not see that building this code into BufQs is necessarily appropriate with the current level of experience.

3.2.3 Network Queues

Network communication involves a Buffer being passed through multiple layers of communication protocols. At each layer, protocol-specific processing is performed such as adding header information and updating state information. Since the header 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 object. In keeping with the object-oriented design philosophy, we should store the state information 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, we include a protocol specific socket structure as part of the Network Queue, which is a specialized BufQ.

Operations are required to connect protocol software modules and initiate a pipeline so that messages can flow or to disconnect them when they are no longer needed. Since a BufQ can represent a protocol layer, we can add the operations, *connect_above*, *connect_below*, *disconnect_above* and *disconnect_below* to BufQs. Invoking any of these operations will cause the owner of the BufQ to perform appropriate actions. For example, when responding to *connect_below*, the owner of a BufQ would check whether the connection is valid (*e.g.*, placement of TCP below IP should not be allowed), and then complete the necessary steps. These four operations are sufficient to support dynamic pipeline configuration.

The framework that we have 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 adding protocol-specific data structures and operations.

3.3 Delivery and Synchronization Abstraction

In this section, we present the details of the delivery and synchronization abstraction as well as the solutions to communication problems related to the dynamic functionality of transporting the data.

3.3.1 Delivery

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 the data since the receiver has no way of knowing if the data is available. Therefore, a *Signal* function must necessarily be incorporated into the enqueue operation to notify the receiver of the Buffer transfer. The Signal function is defined as part of the Queue definition, and can be implemented differently in different communication systems. 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 necessarily need to know the details of the Signal function defined for any receiver's Queue.

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 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 these threads. In conventional communication systems, this capability is generally difficult to achieve. However, Buffer returnQs (the destination of returning Buffers) coupled with the capability to look through nested higher layer Bufds inside a Buffer at 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. Further, returnQs are also useful for resource management and synchronization as will be discussed below.

3.3.2 Synchronization

Synchronization in distributed systems can be subdivided into *user synchronization* and *buffer synchronization*. User synchronization is concerned with supporting various user blocking semantics. Buffer synchronization is concerned with control of Bufds. Below, we describe how both types of synchronization are handled in the Buffer and Queue model.

A Buffer is blocked (*i.e.*, the user should not access it) when it is enqueued on some BufQ other than the sender's or is in the control of some other entity. A Buffer is unblocked when it is dequeued from the sender's returnQ. The return of the Buffer indicates that whichever entity had control of it does not need it any more or has completed an operation on it.

In asynchronous or non-blocking IPC, the user invokes a send or receive operation which merely signals its intention, and then continues execution. When it wants to discover the status of the operation it can simply check or block on the dequeue operation of the returned Buffer from its returnQ. The Buffer, however, is considered blocked until it is returned to the sender's returnQ. In synchronous or blocking IPC, the initiating user could block on the returnQ immediately after initiating the operation.

The principle of returning the Buffer to its owner when the operation involving the Buffer is complete also provides a basis for buffer memory management. When the Buffer is no longer needed, the entity that used or held it enqueues it on its returnQ, where the owner may recover and dispose of or reuse it. Although this mechanism is intended mainly to assist buffer memory 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 when the requested operations are complete. Directly related to acknowledgements is synchronization of user execution. Return of acknowledgements can be used to unblock user entities. Another use is in synchronization of data control. The return of a Buffer signals its availability, as discussed above.

4 Examples of Communication Paradigms Using Buffers and Queues

In this section, we present several examples of how Buffers and Queues can be used to implement different communication paradigms. We first show a simple example of internal communication, which we then extend to implement local IPC and network communication efficiently.

4.1 Internal Communication

As stated earlier, *internal communication* refers to communication between entities within a single protection domain, where a shared memory paradigm is applicable, where messages can be passed by reference, and where 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 Buffers are enqueued and where they remain until dequeued by the owner. A returnQ associated with the entity A is also a BufQ, to where Buffers are returned after completing their journey to one or more communication endpoints.

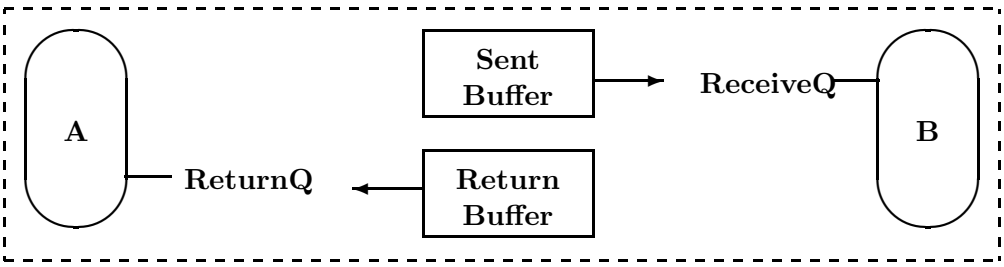


Figure 2: An Example of Internal Communication

Entity A transfers a Buffer to entity B by enqueueing it onto B's receiveQ and, as a side effect, invoking its Signal function. 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 returned to the source's (A's) returnQ. Entity A discovers the status of the data transfer by dequeue-

ing the returned Buffer from its returnQ and examining the *return_status* flag.

Note that so far, we have not mentioned the direction of data flow, but only that of Buffer flow. The direction of data flow and the direction of Buffer flow are orthogonal in the Buffer and Queue communication 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 to A's returnQ. It is also possible to generate bidirectional data flow by having the returning Buffer filled with the B's user data.

This simple shared-memory use of Buffers and Queues is used as a building block in the following examples.

4.2 Local Interprocess Communication

Next, we give an example of interprocess communication between two local processes or entities that do not share an address space. Since the communicating entities do not share a common area of memory, data transfer involves a copy operation. The copy operation is usually carried out by a 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. 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 by enqueuing the IPC Buffer to the IPC server's BufQ as shown in Figure 3. 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: a *sendQ* and a *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.

We note here that the user memory provided by the receiver does not

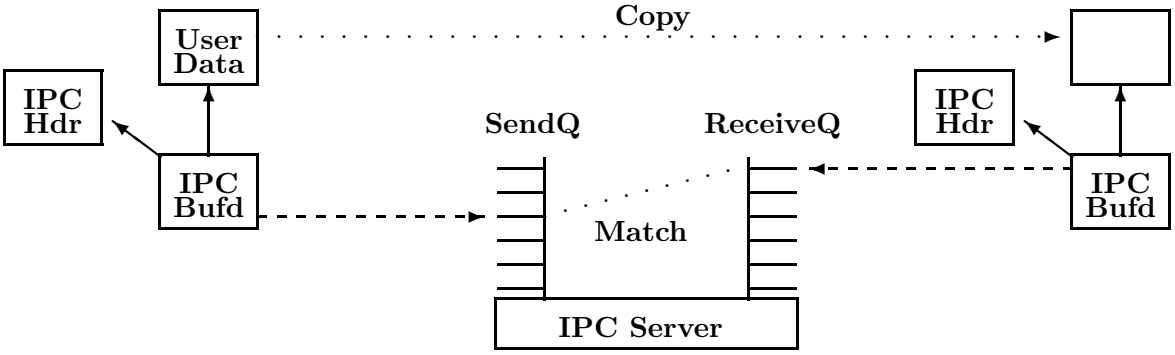


Figure 3: An Example of Local Interprocess Communication

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. By the same token, the sender's user memory also only 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 interfaces as the IPC user interfaces in such Buffer and Queue communication [Shew90]. Thus, user interface and basic communication are orthogonal components in the Buffer and Queue communication model.

Returning to our example, 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 (we do not discuss any specific matching algorithm since it is a function of the IPC protocol and outside the scope of our 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* flags 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 in streams IPC, several Buffers may actually be used in one half cycle to fill or empty a single Buffer in the partner cycle. A stream is closed by flagging the last Buffer with an "end-of-data" bit, which forces both half cycles to terminate cleanly. 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 interprocess communication in library or kernel routines for the convenience of user processes.

4.3 Network Communication

Next, we use an X11 client-server communication example 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 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. We also assume the use of a stack of TCP, IP and Ethernet protocols under the IPC layer in our example.

Presumably, the invoked X library routine constructs an IPC Buffer and prepares it as process A did in the local IPC example above. It then 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 transported 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 our 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

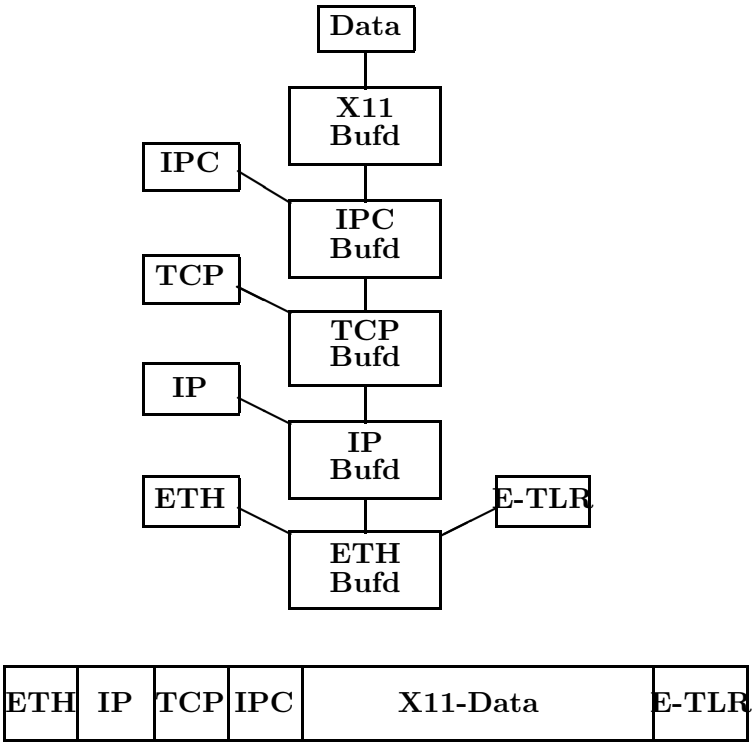


Figure 4: The Internal Structure of an Ethernet Buffer and the Corresponding Packet

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. 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.

In this way, the Buffers can be passed by reference from the user layer to the device layer, avoiding expensive copy operations as much as possible. When the Ethernet controller transmits the packet 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.

5 Conclusion and Future Work

In this paper, we have examined various communication issues in distributed systems as a means to identify the basic requirements for a set of communication abstractions that can form the basis for a generalized communication paradigm. We have presented the Buffer and Queue Model, which meets the constraints of the generic communication model developed in [Hong91]. We have demonstrated how the Buffer and Queue Model strives to provide a uniform communications interface between various types of communicating entities while enjoying the efficiencies of shared memory. Finally, we have demonstrated how various types of existing communication paradigms can be implemented using Buffers and Queues.

Although not presented in this paper, we also have simple, efficient solutions using Buffers and Queues for other problems such as distributed Buffers and Queues, bulk data (or long message) transfer [Cart89, OMal90], distributed and conventional semaphores [Dijk65], and distributed shared memory [Li86].

The Buffer and Queue model is intentionally simpler than previous proposals for communication abstractions or objects. The advantage of this simplicity is that the model can be applied to a wider variety of communication needs, increasing the apparent uniformity among the growing variety of communicating entities in a complex computer system. However, a disadvantage may be that less support is provided by the model for certain stylized but important types of communication. We believe that this criticism can be addressed by appropriate further specialization or subclassing of the basic objects described here.

Acknowledgement

Thanks to Ross Wetmore for numerous useful discussions and comments.

References

[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.

[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.

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

[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.

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

[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.

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

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

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

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

[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.

[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.