Remote Interprocess Communication and its Performance in Team Shoshin

Donald Acton

Technical Report 85-16
November 1985

Department of Computer Science
University of British Columbia
Vancouver, British Columbia
Canada V6T 1W5

REMOTE INTERPROCESS COMMUNICATION AND ITS PERFORMANCE IN TEAM SHOSHIN

By

DONALD WILLIAM ACTON

B.Sc., University of British Columbia, 1982

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES
(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming
to the required standard

[illegible]

[illegible]

THE UNIVERSITY OF BRITISH COLUMBIA

October 1985

Abstract

Team Shoshin is an extension of Shoshin, a testbed for distributed software developed at the University of Waterloo. Part of the functionality of Shoshin can be attributed to its transparent treatment of remote interprocess communication. This is accomplished by having a special system process, the communications manager, handle the exchange of messages between machines. Shoshin's new hardware environment is significantly different from what it was originally designed on. This thesis describes the problems the new hardware presented and how those problems were overcome. Performance measurements of the time required for both local and remote message exchanges are made and compared. Using this empirical data, a simple model of the remote message exchange protocol is developed to try and determine how to improve performance. The software and hardware enhancements made to Shoshin have resulted in an improvement in system interprocess communication performance by a factor of four. Finally as a demonstration of Shoshin's interprocess communications facilities a simple UNIX based file server is implemented.

[illegible]

Contents

List of Tables

List of Figures

Acknowledgement

I would like to thank my supervisor, Dr. Son Vuong, for his patience, guidance and advice on this thesis and Dr. Sam Chanson for reading the final draft.

This whole thesis would not have been possible without the help of my fellow student and friend, Huay-Yong Wang, who was responsible for the porting of the Shoshin kernel. Thanks are also in order to my officemate, Jim Robinson, for the many discusions on this work and other topics. I would also like to thank my fellow graduate students for making the time at UBC more than just a pure academic endeavor.

Chapter 1

Introduction

1.1 Goals

This thesis describes the implementation in the Team Shoshin distributed operating system of a piece of software, the *Communications Manager*, which is responsible for handling interprocess communications between machines running Shoshin. These machines are interconnected via a Local Area Network (LAN) and the communications managers exchange messages on behalf of user processes using the protocol developed at the University of Waterloo. This protocol is implemented and extended to take into account the differing aspects of the two operating environments. Performance measurements of the time taken for both remote and local message exchanges are made and compared. Using this empirical data, the protocol is examined to try and determine how best to improve the performance of remote communications. As a useful demonstration of the message passing capabilities, a file server residing on a machine running 4.2BSD UNIX[1] is implemented. This provides simple file access facilities to the Team Shoshin operating system running at the University of British Columbia.

1.2 Motivation

Once computer systems started supporting more than one process concurrently it became necessary to provide services to facilitate the exchange of information between processes and to control or synchronize access to shared resources. Such services can be classified as either providing mutual exclusion or as a mechanism for providing synchronization between processes. The implementation methodology for the various types of interprocess communications (IPC) facilities can be broken down into two distinct categories [ANDR83,NABE82]:

1. Procedure oriented systems which depend upon some form of shared memory which is generally accessed through procedure calls.
2. Message oriented systems where communicating processes exchange information through the explicit use of messages.

Initially, these IPC methods were developed on uni-processor systems. However, with the proliferation of small inexpensive machines that can be interconnected easily with high speed local area networks, the problem has expanded to include the exchange of information between processes and the synchronization of processes on different machines. With only minor modifications the concepts embodied by message oriented

[1]UNIX is a trademark of AT&T Bell Laboratories

systems developed for uni-processor environments can easily be extended to accommodate a distributed environment. Such an extension cannot easily be made for the shared memory/procedure oriented models, consequently message oriented systems are currently favoured in distributed environments to accomplish synchronization and the exchange of data.

The sections which follow provide background material on some of the various IPC forms. This information serves two purposes:

1. It provides a brief history of how the concepts surrounding interprocess communication have developed.
2. It demonstrates the types of techniques that can be used to solve the synchronization and mutual exclusion problems. This is important since these same problems exist in the communications manager. Knowing what tools and approaches are available to implement a solution can greatly facilitate completing the task at hand.

1.2.1 Procedure Oriented systems

As indicated previously, procedure oriented systems rely on procedures and some form of shared memory to implement a set of IPC primitives. Typically, a shared "variable" contains some information which is used to determine if a certain sequence of statements can be executed. If the variable does not have the appropriate value to allow execution to continue then the process must wait until it does.

In its simplest form the process just performs a busy wait continually checking to see if the value of the variable has changed such that the execution of useful statements should be allowed to continue. (This, of course, assumes some form of processor multiplexing, otherwise the process doing the busy wait would run forever and the variable's value would never be changed by another process.) The busy wait model can easily handle the class of problems that just require some form of synchronization or event signalling. However, the task of implementing mutual exclusion is somewhat more difficult and requires elaborate entry and exit protocols for the critical region. These protocols are difficult to understand, design and verify as correct [DIJK65]. Additionally, a large number of processor cycles could be put to better use by running other processes instead of being wasted while the process repeatedly checks the shared variable.

A more reasonable approach would be to check the variable only once and if it does not have the desired value suspend the process until it does. This is the general approach that Dijkstra [DIJK68] proposed and implemented with the **P** and **V** semaphore operations.

A semaphore is a special purpose integer shared among the cooperating processes that execute simultaneously. The semaphores are initialized to nonnegative integers before the processes start execution. The performing of the **P**(sem) operation causes the value of the associated semaphore to be decremented by one. If the result of this action is nonnegative the process continues to execute, otherwise it is blocked and put on the queue of processes waiting for that particular semaphore.

Should a process perform a **V**(sem) operation then the value of the indicated semaphore is incremented by one and if the result is positive no further action is taken. If that is not the case then a process waiting on that semaphore is unblocked and allowed to compete for the processor.

It should be noted that both the **P** and **V** operations must themselves be implemented as indivisible pieces of code. In addition there is no requirement that a process which issues a **P** operation post a corresponding **V** operation. That is, at any one time it is not possible to determine which processes are executing under the authority of any one semaphore. This leads to the possibility that one process can issue the **V** command on another process's behalf. Since there is no way to determine who has a semaphore, a nasty problem develops when the holder of a semaphore dies before releasing it. A similar problem exists if the poster of a **P** operation *forgets* to perform the matching **V** operation and another process does not do it on the poster's behalf.

An attempt to try and eliminate some of the more serious problems with semaphores resulted in the development of concepts like the monitor [HOAR74]. A monitor consists of the grouping of local administrative data and the associated procedures and functions required to manipulate the data into an entity which is often referred to as a schedule. The local data typically consists of the information and local state variables that are to be shared among the processes. An additional requirement is that only one program may succeed in entering a monitor procedure at a time.

In Hoare's solution to the "bounded buffer problem" the local data consists of the buffer, a pointer into the buffer, a count of the characters in the buffer and a condition variable indicating the state of the buffer. Access to this local data can only be achieved through the routines *append*, which adds characters to the buffer, and *remove*, which removes characters from the buffer. These concepts are very similar to the techniques Parnas espouses for decomposing software systems and for performing data abstraction [PARN72].

On occasion a program which is in a monitor procedure may discover that the resource it is after is in use. In such situations the monitor procedure can issue a *wait* on a particular condition which will result in the program becoming suspended while it releases its mutual exclusion on that monitor. The program will remain suspended until a procedure in the same monitor completes a *signal* operation on that condition. When a signal operation is executed one of the suspended programs waiting on that condition are selected for immediate execution. If no program is waiting on a condition when a signal is issued then the signal is ignored.

An example of a monitor that controls access to a reusable resource such as a tape drive is given in [Fig. 1.1]. A program attempts to obtain the tape drive with a call to the monitor routine **get_drive**. The procedure checks to see if the drive is free and if it isn't then a wait on the condition variable **nonbusy** is performed. The wait will result in the requesting program being blocked while relinquishing its exclusion on

```
tape_drive: monitor
begin
    busy : Boolean;
    nonbusy: condition;

procedure get_drive;
begin
      /* If someone has tape drive wait for it */
    if busy then nonbusy.wait;
      /* indicate that tape drive is in use */
    busy := true;
end;

procedure release_drive;
begin
    if current holder of drive then
       begin
           /* indicate tape drive is not in use */
         busy := false;
           /* signal a waiting process that drive is free */
         nonbusy.signal;
       end;
end;
  /*Initially drive is not in use */
busy := false;

end tape_drive;
```

Figure 1.1: Monitor for a reusable resource

the monitor. When the process currently using the tape drive wants to release it, it does so with a **release_drive** request. The signal performed on the condition variable **nonbusy** will allow one of the waiting programs to resume execution and acquire the tape drive.

1.2.2 Message Oriented Systems

Message oriented systems distinguish themselves from the procedure oriented systems by using the exchange of messages to transfer information and achieve synchronization. There is no requirement that processes wanting to send and receive messages between themselves share memory. Synchronization is accomplished by the fact that a message cannot be received before it has been sent.

To send a message a primitive like **send** is invoked with parameters to identify the destination process and the information to be sent. The corresponding reception primitive is something like **receive** which has parameters to indicate where to place the the received message and what process(es) to receive messages from.

Unlike procedure oriented systems which can use well known names to identify the shared resource, message oriented systems have the more difficult task of identifying a destination process in the turbulent and dynamic system environment where processes are constantly dying and being born. On any one machine the processes there must be uniquely identified, not just for message passing but to allow for the smooth functioning of normal system activities. Expanding an IPC message system to allow messages to be exchanged between machines further requires that process ids be unique system wide. One relatively simple approach is to combine the id of a process on a particular machine with the unique host identifier assigned to that machine. With processes uniquely identified, the destination process can then be selected by using its full process id which implicitly names the machine and process. Since it generally cannot be reliably determined in advance what the process id of a given server or application might be, some form of mapping from a logical name to an actual process identifier is required. Depending upon the system, this mapping may have to be done explicitly by the process through repeated calls to some sort of nameserver, while in other implementations it may be done "auto-magically" by the system.

Message oriented systems tend to come in two flavours; systems with UNIX like pipes [RITC74] which view their messages as a constant stream of data and systems like Thoth [CHER79a] and its descendent Verex [CHER79b,CHER79c] in which messages are individually addressed discrete blocks of data. Stream oriented systems provide some mechanism for establishing a connection between processes whereas the discrete message implementations require the sender to address each message individually. Although the systems mentioned are all confined to a single machine, this is more of an implementation restriction then a conceptual one. All of these IPCs require some means of identifying the IPC destination process. Extending this identification mechanism to work across machines introduces no new problems to the IPC users. To a process running under one of these systems it is irrelevant where the process it is communicating with actually is as long as both the remote and local IPCs provide the same class of service.

The message oriented systems described above have all been extended to accommodate information transfer between machines. In 4.2BSD UNIX this has involved adding support for TCP/IP connections between machines. This provides for the guaranteed reliable delivery of bytes between processes and a connection appears to the user just like a pipe or any other UNIX I/O device.

The V Kernel and Shoshin have taken the message passing paradigm of VEREX like systems and extended them across machines. To processes making use of these facilities the processes appear to be on the same machine.

By establishing the appropriate relationships between processes the synchronization and mutual exclusion problems presented under the procedure oriented systems can be solved. For example to control access to a resource like a tape drive a process can be established as the owner of the device. To use it a requester sends a message to the owner and then waits for a message indicating the status of the operation. A similar construct can be used to allow the requester to return the resource.

1.3 Thesis Summary

In this section a brief description of the contents of the following chapters is provided. Chapter two provides background information on the various aspects of the implementation environment. The new hardware that Shoshin is destined for is contrasted with the initial hardware and some of the problems this introduces are discussed. A brief overview of the Shoshin software capabilities is also provided.

Chapter three provides a description of the implementation problem and of the tools and methodology that will be used to solve the problem. The fourth chapter then describes how the problem was actually solved. A description is provided of the remote IPC protocol implementation in both the Team Shoshin and UNIX environments.

Chapter five presents the final results and analysis of the remote IPC performance. An informal model of how the communications manager operates is developed to describe the transmission of messages. This model then serves as the basis for proposing ways in which the performance of remote IPCs might be improved. In the final chapter the results of this thesis are summarized and possible topics for further research stemming from the work are presented.

Chapter 2

System Architecture

The hardware that Shoshin was originally developed on at the University of Waterloo is vastly different from the hardware it was ported to here at UBC. Although the two sets of hardware differ significantly the software environment presented to the end user remains essentially the same. During the porting effort the opportunity was taken to provide Shoshin with some new features. This chapter shall contrast the two hardware environments and briefly describe the Shoshin operating system and the extensions that have been made to it.

2.1 Hardware

2.1.1 Original Hardware

Shoshin was originally developed at the University of Waterloo on a set of LSI 11's. These machines were interconnected with a piece of hardware dubbed the *Schoolbus* [TOKU83a]. The *Schoolbus* is a tailor made high speed bus yielding a "port" to "port" data transfer rate of 2 megabits/second with 32 ports on the bus. The bus is operated by a centralized programmable bus controller which allocates access to the bus using synchronous time-division multiplexing. A port is an access point to the bus and is implemented physically as a bus interface unit (BIU) which is responsible for sending and receiving the data for a particular port. A data frame on the bus can vary from two bytes to 256 Kbytes. For the interprocess communication protocol the maximum frame size was set to 544 bytes.

As previously mentioned the bus interface unit is actually responsible for the sending and receiving of packets on the bus. The BIU allows a port to send to a specific BIU, a subset of the BIUs or by using its broadcast facilities to all the BIUs. Unfortunately broadcast packets are limited to two bytes. The receiving functions consist of the ability to receive from any port, a specific port, a specified BIU which is sending to a subset of the BIUs and/or broadcast packets. The combined hardware design of the BIU and *Schoolbus* allows the sender to determine whether or not the packet was received by the destination BIU. There are four possible outcomes that the hardware can report when trying to send a packet.

- SUCCESS: packet was successfully transmitted and received
- BIU_DEAD: destination BIU does not exist
- CPU_DEAD: destination BIU exists, but CPU not active
- BUS_ERR: some sort of transmission error

All of these return statuses, except the first one, are treated as fatal errors indicating that communication to the desired BIU cannot be completed. As will be seen later the ability to determine what happens to a packet that is sent is important to the design and implementation of a protocol.

Each LSI 11 had 248 Kbytes of main memory with the additional constraint that no process's virtual address space could exceed 64 Kbytes in size. A DEC PDP 11/45 running UNIX was used to provide a network file and boot server to the Shoshin machines. The UNIX machine was not dedicated to Shoshin and ran other user applications simultaneously.

2.1.2 New Hardware

The operating environment that Shoshin was ported to is significantly different from that of its birthplace. Shoshin was ported to a set of SUN Workstations[1] connected to an Ethernet.[2] Each SUN Workstation is based on a 10 megahertz Motorola 68010 [MOTO84] with from one to three megabytes of main memory. Shoshin takes advantage of the SUN's memory management hardware to provide separate virtual address spaces for each process or team of processes. A one megabyte limit is placed on the size of a process by Shoshin, but the hardware provides for much larger processes. In message passing systems the tendency is to have a large number of small communicating

[1]SUN Workstation is a trademark of Sun Microsystems Inc.

[2]Ethernet is a trademark of Xerox Corporation

processes cooperating to solve a problem instead of one large monolithic program, consequentially the limit on the size of a process should not be a serious restriction. Additionally, a 10 megabit/second Ethernet is used to interconnect the SUNs and other local machines which consist of two VAX 750s and a VAX 780 [Fig. 2.1]. The SUNs

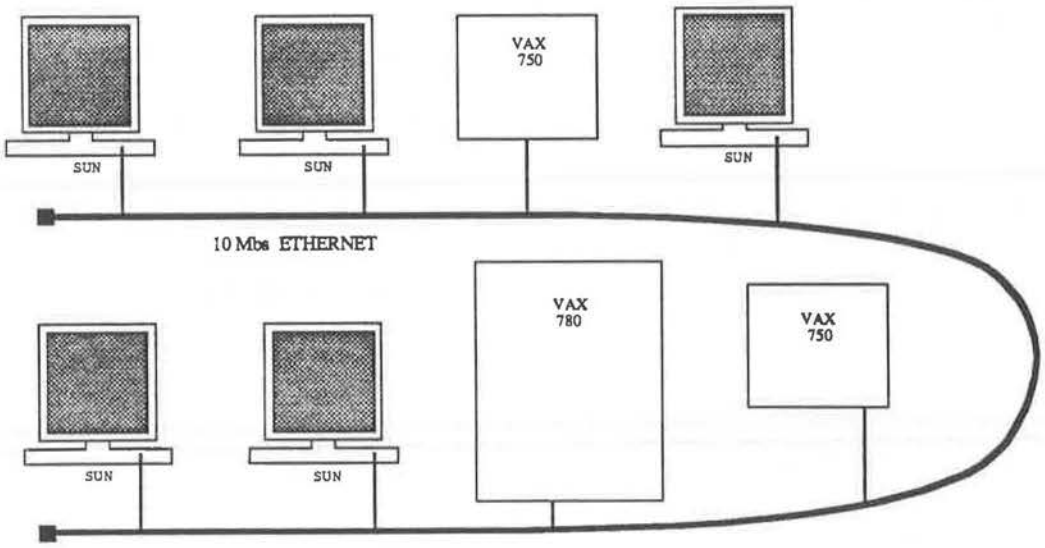

Figure 2.1: Network Configuration

can be divided into two classes of machines depending upon the type of controller used to provide access to the Ethernet. The initial implementation, as discussed in this thesis, was done on the machines using the 3Com Ethernet controller. The remaining SUNs use a controller based on the Int$_{e}$l 82586 local communications controller chip. Other minor differences between the two types of SUNs exist but they do not affect the communications aspect of Shoshin.

The Ethernet is a Carrier Sense Multiple Access Network with Collision detection (CSMA/CD) [METC76] and its operating characteristics differ significantly from those of the *Schoolbus*. One of the fundamental differences concerns how the network bandwidth is allocated among the sending units.

By using synchronous time division multiplexing to determine which BIU can have the bus, the *Schoolbus* guarantees that after a finite period of time a given BIU will have an opportunity to transmit. On an Ethernet the sender waits until there is no activity on the Ether before initiating a transmission. If a collision occurs during the transmission then the sender backs off and tries again. In the worst case the sender may never get to send the packet or else a very long time my elapse before sending. Such operating characteristics imply that there is no upper bound on how long a sender may have to defer before acquiring the Ether. In practice, however, Ethernet loading conditions which would cause such behavior are seldom encountered [LIU82,STALL84]. When the offered load on the Ether is low then a sender will not have to defer very long, if at all, before transmitting. However, as the offered load increases the delay will be greater and more variable. It would therefore be expected that under heavy offered loads the *Schoolbus* would provide more consistent response times when compared to the Ethernet.

When a packet is transmitted on the Ethernet there are no assurances that the packet will actually be received by its destination; there is just a high probability that it will. Unlike the protocols that use the *Schoolbus*, those using the Ethernet must assume that occasionally a packet will be missed or corrupted during transmission. With the *Schoolbus* either a packet is delivered or it isn't and attempting to retransmit the packet when it cannot be delivered will not alleviate the problem. The Ethernet's lack of a hardware guaranteed acknowledgement required the remote IPC protocol to be modified to accommodate the possible loss of packets.

The 3Com Ethernet Controller has one transmit and two receive buffers. All these buffers support the standard 1514 byte packets used on the Ethernet [DEC80]. The first fourteen bytes of the packet are reserved for the source, destination and type fields that make up the standard Ethernet header with the remaining 1500 bytes devoted to user data. A 16 bit CRC checksum is computed automatically by the sending hardware and appended to the outgoing packet. Before the packet is accepted by the receiving hardware, it computes and verifies the checksum against that produced by the sender. The controller supports the reception and transmission of multicast, broadcast and individually addressed packets. If a receiver's buffers are full when a packet arrives for it then that packet is ignored. Packets which are too small, (less than 60 bytes) too large, or have frame alignment errors are also ignored.

2.2 Software

Shoshin[3] is an operating system designed as a distributed software testbed at the University of Waterloo. The Shoshin porting effort has provided the opportunity to modify and extend the original Shoshin system. However, the original goals of having a simple, easily reconfigurable system have not been sacrificed. Too achieve these goals Team Shoshin combines a simple yet powerful set of IPC primitives with a versatile and flexible series of process creation functions.

2.2.1 Interprocess Communication

The IPC model selected expounds the philosophy of simplicity and ease of reconfigurability by providing a simple yet extensible facility as opposed to a powerful general purpose IPC implementation. The direct message passing paradigm meets this criteria while at the same time providing a mechanism to support, if it is desired, more complicated IPC models.

Part of the power associated with Shoshin's message passing scheme is its uniform treatment, at the user level, of both local and remote communications. In all situations the process simply supplies the process identifier (PID) of the process it desires to communicate with to the appropriate IPC primitive. The PID consists of two parts, the host identifier (HID) which identifies which host to send the message to, and the

[3]Shoshin is a Japanese word meaning beginner's mind.

local id (LID) of the destination process on that host. When using the IPC primitives both parts of the PID must be supplied. The size of a message that can be sent is arbitrary with the maximum size being dictated by the buffer size of the receiving process or the system imposed maximum of 32Kbytes.

The Team Shoshin IPC model supports both blocking and non-blocking receive primitives but only blocking type send primitives. The inclusion of non-blocking sends would introduce an added degree of complexity into the implementation of the IPC primitives since the system would now have to manipulate and maintain message buffers. Non-blocking sends also result in more complicated user programs since a user program can no longer be assured that the destination process got the message. As an example a process may die or exit before it takes delivery of a message in its queue, yet the sender has no way of knowing this unless some protocol between the two processes is specified in advance. With blocking sends, as soon as the sender resumes execution it knows that the message has been delivered (assuming no error is reported) and it can proceed accordingly.

In Shoshin, a process wanting to send a message has the following primitives at its disposal:

nr = **request**(topid,&msg,m,&buf,n,mtag)

ns = **reply**(topid,&msg,m)

ns = **bsend**(topid,&msg,m,mtag)

The destination PID is specified by the "topid" parameter while "&msg" and "&buf" specify, respectively, the addresses of the message to send and the buffer area to place a received message in. The parameter "m" indicates the size, in bytes, of the message to send and in the **request** primitive the "n" specifies, again in bytes, the size of the buffer area for the received message. The parameter "mtag" is used for matching selective receives [TOKU83b] and sends. The IPC primitives are all defined as functions and they return the actual number of bytes sent (ns) or received (nr).

Using the **request** and **reply** primitives a client-server relationship can easily be constructed and implemented by the programmer. Once a process sends a message with **request**, the process remains blocked until the receiver issues a matching reply. When using the **bsend** primitive a process remains blocked until the receiver accepts the message.

The following four basic primitives can be used for the reception of messages:

nr = **brec**(frompid,&buf,n,mtag)

nr = **nrec**(frompid,&buf,n,mtag)

xpid = **brecany**(&buf,n,mtag)

xpid = **nrecany**(&buf,n,mtag)

The two functions **brecany** and **nrecany** are used to receive messages from any process while the other two primitives require the explicit naming, through the parameter "pid", of the process to receive a message from. When one of the receive any primitives is used a structure known as an extended PID (xpid) is returned. This structure contains the PID of the process that was selected for reception and a byte count of the number of bytes received. All the parameters and return values not described here have the same usage as the ones presented with the send primitives.

The non-blocking receives (the primitives beginning with n) return immediately if there is not a message ready to be received. This ability allows processes that require a polling type mechanism to be implemented. As their name implies, the blocking receives remain blocked until a message meeting the constraints specified by their various parameters arrives.

The original Shoshin allowed a receiving process, if it desired, to set aside some space using **Setflcb** (set flow control buffer) so that messages from specified senders could be buffered. Using the **fsend** [TOKU83b] primitive the sender could then send messages to this process without blocking as long as the receiver had buffer space available. In Team Shoshin this facility has not been implemented. Since buffer space tends to be finite, there will always be the possibility that a sender will block so it must be prepared for that contingency even in allegedly non-blocking situations. It therefore seems reasonable to have all sends block since the sending applications program must always be prepared for this event anyway. Send primitives that always block can, to the same degree as buffered receives, easily be made to be non-blocking by setting up a group of processes that act as receive buffers. The sender simply sends to these processes in a fixed order thus accomplishing buffering. The processes then relay the received information to the intended receiver when it is ready. The extra context switches and scheduling overhead of this method undoubtably results in a less efficient implementation of "non-blocking" sends then could achieved by providing it as a basic service.

Team Shoshin provides numerous facilities, as the next section will describe, for the quick and efficient creation of processes thereby making it reasonable to have processes act as buffers.

2.2.2 Process Management Facilities

Team Shoshin has taken advantage of the nice hardware environment provided by the SUN Workstations to extend the original process creation primitives and to add new ones. In particular the concept of teams has been introduced at both the kernel and user level, hence the name Team Shoshin.

A team is a group of processes that share a common address space, code and global data areas[CHER79a]. Teams have the advantages that:

- Process creation can be quick because the appropriate code is already loaded.
- Message exchanges are more efficient since there are fewer context switches
- Data can be shared between team members thus reducing the size and number of messages that need to be sent to solve a problem. However, it then becomes the programmer's responsibility to ensure data consistency.

The term team implies more than just sharing an address space, it also imparts a certain connotation on the approach to be used when solving a problem in a distributed programming environment. In our day to day lives we typically think of a team as a group of individuals who are cooperating in a very close manner to try and solve a problem. By transplanting this notion to the program development environment an accurate portrayal of multi-process structuring using teams is achieved. Thus solutions using teams typically consist of several processes which can logically execute concurrently, yet require close interaction with other team members. This interaction is provided through messages and shared data.

When a new process is started it is considered to be a team root. Team members can then be created with a system call that takes the name of a function in the currently running process. This function then becomes a team member that shares the code and global data spaces with the other team members but which has its own local stack. In addition to team creation Team Shoshin supports several other methods of process creation.

Team Shoshin has supplemented the original **create** system call with the **teamcreate** and **execv** system calls. These routines take the following format:

pid = **create**(pname,f_tree,rel_pri,at)

pid = **teamcreate**(func,rel_pri,ssize,nargs,arg1,arg2,...)

execv(pname,argv)

The create primitive is identical to the one in the original Shoshin. The name of the load module of the process to be created is given by "pname". The parameter "f_tree", by taking on the value ATTACH or DETACH, specifies whether or not the created process remains attached to the family tree. This becomes important when process destruction is considered. The priority of the new process is given by specifying, in "rel_pri", the priority, relative to that of its parent's, of the created process. The machine that the created process is to reside on can be specified with the "at" parameter. In Team Shoshin a process created with the **create** primitive is a team root.

The **teamcreate** primitive is used to create a process on a team. This type of creation is fast since a separate address space doesn't need to be created nor does the file containing the code and initialized data have to be located. The entry point of the function to execute as a process is specified in "func" and a stack of "ssize" bytes is allocated to the process. The "nargs" parameter specifies the number of arguments passed to the function while "arg1","arg2"... are the actual arguments. Processes created with **teamcreate** are always attached to their parent and destroyed when the parent exits. Both **teamcreate** and **create** return the process id of the process created.

The **execv** call is identical to its UNIX namesake and results in the executing process being replaced by the one named in the call. Arguments are passed in typical C fashion [KERN78] with "argv" being an array of pointers to character strings. When

execv is successfully executed the process will be a team root.

Any discussion of process creation is incomplete without some mention of how processes terminate and return their resources, such as memory, to the system. A normal program termination usually consists of a program making an explicit call to the exit routine or an implicit one by "falling off its code." A process's existence can also come to an end if it is named in the **kill** system call by another process. As its name suggests, **kill**(PID), causes the process specified by PID to be killed. Although a PID implies both a host and local identifier the kill primitive currently is only supported for processes on the same machine. Program errors such as segment and access violations or illegal instructions, which on a UNIX system result in the all too familiar message "core dumped", will also result in a process being terminated. Regardless of how a process terminates, a process destruction wave is initiated that results in all of a process's attached descendents being killed.

2.2.3 Miscellaneous System Services

Interprocess communication and process creation primitives provide the basic framework for the development of distributed systems and programs. However, other facilities to look after some of the more mundane aspects of a programming environment are also required. These system services really belong to two category:

1. Those functions which can be used by any process in the system.

2. Those functions which can only be executed by privileged processes such as the communications manager.

In the first category are such things as **pexist**, which checks for the existence of the specified process; **malloc**, **calloc** and **free** which dynamically acquire and free memory for the user process (**calloc** allocates memory and sets it to zero); **delay** which delays the process for a specified period of time; **whois** which returns the process ID of well know system processes; **gettime**, **gettimeofday** and **settimeofday** which read or write various system time of day counters; and, of course, there are routines to read and write data. In designing and naming these services the approach has been, wherever possible, to adopt the names of the similar UNIX functions. By doing this a certain degree of user program portability is maintained across systems since the most basic and commonly used functions are identical both in name and action. It should be kept in mind that although the functions names may be the same the implementation methodology within the two systems certainly is not.

Just as in most systems, Shoshin provides a class of functions which can only be executed by privileged processes. Typically these functions result in the manipulation or reading of some kernel specific data structures. Many of these functions were developed specifically for the communications manager process; however, an attempt has been made to make the functions general purpose in nature so that at a future date other processes providing system services can make use of them.

The privileged functions can be placed into the categories: hardware related functions, message related functions, and miscellaneous functions. The hardware oriented functions allow a process to determine specific information about the hardware configuration of the machine the process is executing on. In the communications manager, these routines are used to locate the Ethernet controller and then to map a virtual location in the process's address space to the actual physical location of the controller. The message related functions are used to coordinate the exchange of messages between remote processes and are meant strictly for the use of the communications manager. The miscellaneous functions do not categorize themselves so neatly and their functions tend to be more general purpose in nature. For example, one miscellaneous function can be used to transfer data between the virtual address spaces of processes. Although this function is of great use to the communications manager for moving messages around, its usefulness is not restricted to this particular application. All of the privileged functions outlined here are typical in that they are extremely powerful and provide services that the normal user process would not be expected to use. Since the use of these functions in a malicious manner would be detrimental to the system, it is extremely important that these functions only be executed by well known and *trusted* processes.

Chapter 3

The Design Problem and Approach

The design of Shoshin stresses a small kernel providing only the basic functions like memory allocation, synchronization, process scheduling and local IPC facilities. More complicated services, including remote communication, are realized through system server processes.

When a process invokes an IPC primitive the kernel examines the host id portion of the destination id supplied to determine if it is a remote or local request. If it is a local request then the kernel completes the request itself otherwise it informs a special system process, the communications manager, that there is a remote IPC to deal with. The communications manager will then complete the remote request on behalf of the kernel.

3.1 Communications Manager

Before considering just what form the communications manager will take it is important to understand its operating environment and the types of events that it will have to handle. The communications manager is started by the kernel at boot time and is responsible for all remote communications. To provide this remote communications ability the communications manager must also interface with the hardware controller for the Ethernet. Functionally, the communications manager is a server process [GENT81] that accepts messages, which can be viewed as events, and takes actions based on these messages. The occurrence of an event, as indicated by the arrival of a message, signifies that certain actions are to be performed. It is essential for maximum concurrency in remote messages that the communications manager never block other than while waiting for a message. The communications manager will be expected to deal with the following three types of externally generated events:

1. A message from a clock process indicating that a fixed amount of time has expired.
2. A message issued by the kernel on behalf of a user process which indicates that a user process wants to perform a remote IPC.
3. A message from a kernel process indicating that a physical I/O operation on the network has completed.

As previously mentioned, the communications manager is totally responsible for Team Shoshin's interface to the network. To accomplish this, the communications manager must interact directly with the hardware. The communications manager is notified, in a message from a kernel process, whenever the Ethernet controller generates an interrupt. Although the kernel process detects the interrupt the actual servicing of it remains the responsibility of the communications manager.

A user request, as forwarded from the kernel, can be viewed as a request for access to the network resource which the communications manager owns. Currently this request can only have the form of a remote interprocess communication and it therefore becomes the communications manager's responsibility to guarantee delivery of this message. The transfer of this message across the network is accomplished through a well defined communications manager to communications manager protocol known as the remote IPC protocol.

In porting the communications manager to the new hardware a major goal was to try and make as few changes as possible to the original software. However, the vastly different underlying hardware environments and the resulting treatment of message exchanges dictated that fundamental changes in the remote IPC protocol would have to be made. In substituting the 10 Mbs Ethernet for the *Schoolbus* and its hardware generated packet acknowledgements (ACKs) a considerable amount of information pertaining to the state of a message exchange was lost. In particular a software solution had to be found to convey the information previously provided by the ACKs.

The primary function of the communications manager is to implement the remote IPC protocol. This was accomplished by designing the portion of the communications manager that deals with the remote IPCs as a finite state machine. Each active remote message has a message descriptor which, among other things, defines the current state of that message exchange. When an event occurs the communications manager locates the appropriate message and, usually after some action, updates the state of the message. For each message type (bsend, brec, etc) there exists a protocol state matrix which when indexed by a message's current state and event specifies what action to take and what state to change to. The format of a protocol state matrix can be seen in Figure 3.1. The use of the state tables makes it particularly easy to test new or

Events	Protocol Status		
	State 1	State 2	State 3
Event 1	action, new state	action, new state	action, new state
Event 2	action, new state	action, new state	action, new state
Event 3	action, new state	action, new state	action, new state
Event 4	action, new state	action, new state	action, new state
Event 5	action, new state	action, new state	action, new state
Event 6	action, new state	action, new state	action, new state

Figure 3.1: Protocol State Matrix

modified protocols since all the protocol designer has to do is update the protocol state matrix and provide the new functions. Regardless of how simple and straight forward the implementation strategy may seem, the majority of the work revolves around the non-trivial exercise of designing and verifying the protocol. In developing the new remote IPC protocol the intention was not to implement a totally different protocol but to extend and modify, as much as possible, the original remote IPC protocol to accomplished the task at hand.

3.2 Design Philosophy

The term communications manager itself does not embody any information about the way in which the communications manager should be implemented or structured. In an operating environment like Shoshin there are several possible approaches that could be used. These range from a single process to a large number of small processes coordinating themselves through messages. Typically a solution will range someplace between the two extremes depending upon the advantages and disadvantages of each approach in a particular situation.

By employing multi-process structuring, in an extreme case, it would be possible for the designer to decompose the actions of the communications manager in such a manner that function calls could be replaced by the exchange of messages between processes. Such an approach would be expected to extract a severe performance penalty since the the cost of sending a message versus that of a subroutine call is relatively high. These additional costs can be attributed to the extra process switches and data copying between address spaces that ultimately takes place.

A moderation of this approach would result in some of the processes being replaced by function calls. In doing this a certain amount of speed is gained due to the reduced system overhead but there is a possible loss in concurrency of execution among the processes comprising the communications manager. If poor choices are made the gain in speed may not be sufficient to overcome the reduced concurrency. For example, some "bookkeeping" type functions may not need to be done immediately and could be done while the main portion of the communications manager waits for a new event. If these bookkeeping functions were pulled into the main processing loop, then work that was previously done while waiting for a request (event) would become part of the event processing. Depending upon the complexity of the bookkeeping functions, the overall effect might be that the total elapsed time for processing a remote IPC would increase.

Shoshin was originally designed as a distributed testbed, therefore the emphasis for all parts of the system was weighted more towards providing a convenient environment for the testing of ideas in distributed computing rather than speed. Since this implementation of the communications manager is a port from Waterloo one can take advantage of hindsight to try and improve the system. To this end there has been a greater concentration on improving the performance aspects of the system than to unfettered flexibility. This has resulted in the elimination of seldom used services such as **fsend** as described in section 2.2.1.

3.3 Testing Methods

For any piece of software an important stage in its development is the test phase. It is essential that any new software be tested and, to the best of ones ability, verified to perform in a "correct" and consistent manner. In developing the communications manager the difficulty of the test phase is compounded by the fact that diagnostic tools like symbolic debuggers have not yet been developed for Shoshin. However, these tools do exist for UNIX and this provided a considerable impetus for also implementing the communications manager on a UNIX system. By porting the communications manager to UNIX it was possible to make use of debugging tools which are considerably more powerful than a fleeting thirty line audit trail on a monitor. For example under UNIX it is possible to log the audit trail to a file so that a more extensive record of events exists. This allows one to detect subtle problems and system state changes that manifest themselves over a time frame that is longer than what can reasonably be captured on a small screen.

The ability to use the UNIX symbolic debugger, *dbx*, also allows one to arbitrarily monitor the progress of a process and to examine variables and memory locations. The debugger can also be used to perform a post mortem examination of the core files that a program produces when some sort of execution error takes place. This is a very useful feature for determining just what went wrong in an allegedly working program.

A further advantage of having a UNIX version of the communications manager occurs when trying to improve performance. Under 4.2BSD UNIX there exist two execution time code profilers, *prof* and *gprof*, which provide information of how many times during a given session each function is called and the average duration of the call. Using this information it is possible to determine which routines are executed the most and then concentrate on improving their performance.

The profilers can also produce an execution time call graph which is a record of which routines were called, how often they were called and from where. A close examination of this call graph can then help to detect any abnormal execution patterns. An example of how to apply this technique occurred when the communications manager kept crashing and reporting it could not get any more memory. By studying the execution profiles it was possible to determine that memory allocation routines were being called much more than the memory freeing routines. (There should have been a one to one correspondence of calls except for a a constant number of allocations that were made to set up some of the initial data tables.) This mismatch between the memory allocation and freeing routines inevitably resulted in the process running out of memory. A further examination of the profiles helped to determine, based on where the memory was being allocated, where to look to see why the allocated memory wasn't being freed. If one had tried to solve this problem strictly on a Shoshin system the magnitude of the task would have been much greater and the time required to solve the problem much longer.

A communications manager operating on UNIX provides more than just a a way of debugging and monitoring the code. It also provides a mechanism through which Shoshin based processes can communicate with UNIX based ones. Such a facility provides Shoshin with a large number of services which invariably increases the functionality and usability of the Shoshin environment. Having such facilities available is extremely important in the early stages of system development since it allows users to build and provide some complex, yet required, services without having to worry about the very low level details of the system. Access to a file system is an example of this.

Chapter 4

Implementation Details

The implementation of the communications manager consists of several distinct problems each of which requires its own unique set of design decisions. However, each of these parts does not operate independently and must provide a well defined interface to the other conceptual entities making up the communications manager. The choice of the format of this interface and where it occurs determines how easily extensions or design changes can be implemented in the software. The major and most crucial design decisions were made in the following areas:

- That portion of the communications manager which deals with the implementation of the remote IPC protocol and the protocol itself.
- That portion of the communications manager which manages the Ethernet hardware controller.
- The UNIX implementation of the communications manager.

Although work proceeded in parallel on these topics it is rather difficult to present the solutions in a similar fashion without causing a great amount of confusion to the reader. Instead each of the above topics will be dealt with on an individual basis as much as possible.

4.1 The Remote IPC Protocol

The remote IPC protocol is really one level in a hierarchy of protocols that makes use of several protocol layers below it [Fig 4.1]. Each one of the protocol layers has

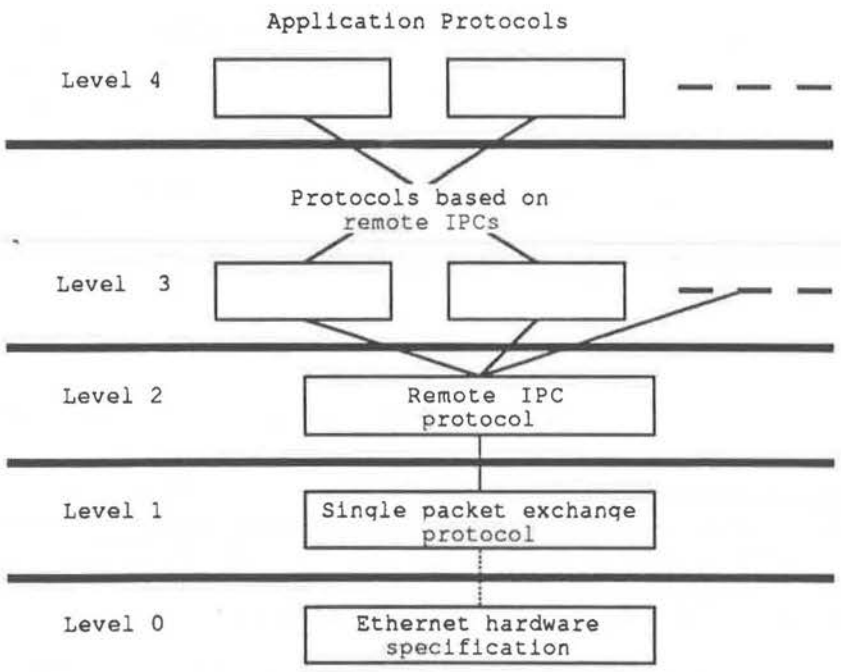
Application Protocols
Level 4
Protocols based on remote IPCs
Level 3
Level 2
Remote IPC protocol
Level 1
Single packet exchange protocol
Level 0
Ethernet hardware specification

Figure 4.1: Protocol Layers

a specific function and communicates only with those layers either directly above or below it.

4.1.1 Protocol Layers

Level 0

The level zero protocol describes the hardware that allows for machines to physically communicate with each other. The protocol, which is really based in hardware, specifies how access to the net is accomplished and what the functional characteristics of the network are. The level zero protocol for the Ethernet also specifies such things as the amount of data that can be transmitted, when a station can transmit and what actions to take when a collision is detected. It is important to recognize the operational properties of the network since these characteristics indicate the types of network transmission problems that the higher protocol layers might have to detect and correct. For example the hardware computed and verified polynomial CRC is sufficient, given the low error rate of the Ethernet, to guarantee that corrupted packets can be detected and ignored. Consequently, the higher level protocols rely on this CRC to detect transmission errors and they do not have generate checksums of their own.

Level 1

The remote IPC protocol is based on the level one protocol which provides a specification of how to move a single packet (datagram) between hosts. This basically consists of adding the Ethernet header, which identifies the source and destination machines and the underlying protocol being run, to the outgoing packet. (If the packet is destined for a UNIX machine then additional headers to allow the packet to get to the the desired process on that machine are added.) This level also ensures that the packets concerned are transmitted and hence delivered (assuming that they are not lost) in the sequence that they are sent. By guaranteeing something about the order in which packets are transmitted and processed it is possible for the higher level protocols to make certain assumptions about the types of packets that may be expected in any one protocol state This greatly reduces the complexity of the protocol processing.

Level 2

The level two protocol is the remote IPC protocol which provides for reliable message transfers between processes across the network. Access to this service is transparent and provided through the normal Shoshin IPC primitives.

Level 3

In many respects the protocols of level three are analogous to the protocols of the presentation layer in the seven layer ISO reference model [TANE81]. The function of this level is to provide commonly used specialized services to the user processes. Services for data compression, data encryption, file transfer and virtual terminal emulation typically reside at this level. Although these functions may not be used by a user program, conceptually there is still a protocol layer present and it can be viewed as a layer which passes data through unaltered.

Currently, in Shoshin, the only level three type service is the simple file server that has been implemented between the Shoshin and UNIX machines. In a more complete implementation of the Shoshin system a distributed name server, which would provide a mapping between process names and services, would also be a candidate for inclusion at this level. This would allow processes to request services from other servers without having to know that server's PID in advance.

Level 4

Level four is the applications layer and serves as the boundary between user programs and system provided services. The protocols at this level are designed and implemented by the user and implemented between user processes. These user processes will typically make use of some of the special system services like the name and file servers although this is not required. Ultimately the user protocols are based on and constrained by the remote IPC functions which the system provides.

4.1.2 Communications Protocol

The proper functioning of the remote IPC protocol is essential to ensuring that both parties involved in a remote message send perform consistently and agree on whether or not a message is successfully exchanged. This is accomplished by modifying the original Shoshin IPC protocol [WOOD82] to handle the new operating environment.

As pointed out earlier, the original Shoshin system used the specially designed *Schoolbus* to transfer messages between machines. In this implementation the *Schoolbus* has been replaced by an Ethernet which does not support hardware generated acknowledgements (ACK) of the transmitted packets. The Ethernet [METC76] provides only a best effort attempt at packet delivery. Consequently any protocol using the Ethernet must be prepared to handle the occasional loss of a packet. The other major problem area that the protocol must be able to handle is the failure of processes, machines and/or the network [RAVI85].

To overcome the lack of hardware generated ACKs the new communications protocol has replaced most of the hardware ACKs with implicit ones based on the next type of packet expected in the message exchange. Under certain conditions explicit ACK packets are generated by the remote site. Since it is possible for the original packet or the ACK (implicit or explicit) to be lost, an extensive set of timeout facilities has been integrated into the protocol to allow for the retransmission of unacknowledged packets.

When a sender wants to transmit there are several states that the remote process/host may be in. If the host is dead or the destination process is non-existent then, obviously, the message cannot be delivered. Network and processor malfunctions can be detected with a high degree of accuracy through the repeated lack of software generated ACKs. When a communications manager receives a properly formed packet it immediately verifies existence of the target process. If the process exists then the packet is processed in the normal fashion otherwise a rejection packet indicating that the process does not exist is returned to the sending communications manager. Should the sending communications manager not see an ACK after repeatedly transmitting its own packet then the assumption is made that the network is down or that the destination host is dead. By using this approach to failure detection it is possible to incorporate a process checking mechanism into the regular activities of the remote IPC protocol.

A message exchange is initiated by a process issuing a request, through one of the IPC primitives, to exchange information with a process on a remote host. If the remote process exists then it is either currently waiting to receive the message or it isn't. (This also includes the deadlock situations when both processes want to either send or receive a message between each other.) Should the remote process not be ready to exchange a message then the protocol enters a process checking phase which results in this initial packet being repeated at a regular interval. This will continue until the destination process is ready to exchange the message or until one of the previously outlined failure detection mechanisms is triggered. This process checking phase is just one portion of three more general phases of the remote IPC protocol which are enumerated below.

1. A setup phase.

2. A data transfer phase.
3. A termination phase.

Setup Phase

The setup phase consists of verifying, in the above described manner, that the desired process is actually running. As an example of how this works consider what happens when a process, the sender, issues a **bsend** [Fig 4.2]. Once the communications man-

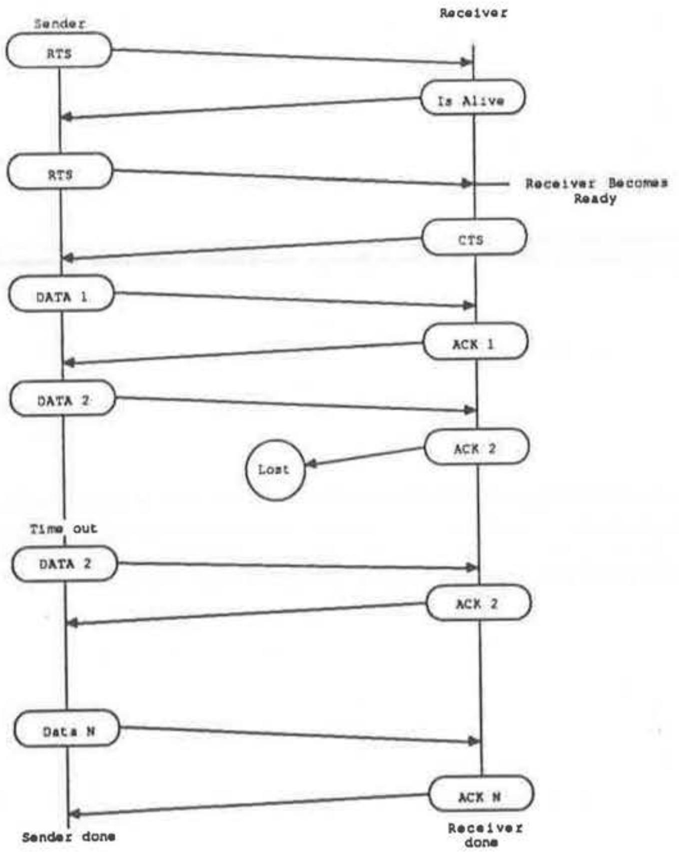

Figure 4.2: Typical message exchange

ager on the sender's side is notified of the bsend it issues a request to send (RTS) packet. It expects to receive either a process is alive or clear to send (CTS) packet in response. (The repeated lack of a response or a response packet indicating a protocol or system error would simply result in the protocol terminating.) The RTS packet is repeated at a regular interval until the CTS packet is received. The CTS packet indicates to the sender that the destination process has issued a receive primitive and consequently is blocked waiting for the actual data of the message. Should a receive IPC be issued before the corresponding send IPC then the communications manager will issue periodic checks with an are-you-there (AYT) packet to verify that the remote process is alive. Once the sender process issues its IPC the normal exchange sequence of RTS and CTS packets are expected. The CTS packet indicates that the protocol is to enter the data transfer phase.

Data Transfer Phase

The data transfer phase is that portion of the remote IPC protocol which accomplishes the reliable transfer of data between machines. Since the IPC primitives allow very large messages to be sent, it is possible that a message may exceed the maximum sized packet that the network can accept. In such a situation large messages will have to be fragmented into several packets. Each one of the fragmented data packets has a packet sequence number which is used by the receiver to make sure that duplicate packets are ignored and that packets are not missed.

To insure the reliable delivery of the data packets some form of acknowledgement had to be incorporated into the data transfer portion of the message exchange. The initial attempt consisted of sending all the data packets comprising the message as fast as possible once a clear to send had been received and then waiting for a response packet indicating how the exchange went. However, for large multiple packet messages ($>$ 8 packets) the receiver had trouble keeping up with the sender and often became swamped with the result that back to back packets were often dropped. This loss of packets meant that the sender had to either timeout and retransmit and/or wait for some acknowledgement from the remote site indicating what the next expected packet was so that the retransmission phase could be entered.

During the retransmission phase the same swamping problem inevitably made its reappearance and packets would again be missed. The net effect of this was that a lot of retransmissions were required for multi-packet messages. Once it was noticed that retransmissions were required it was not totally unexpected that the same problem would appear during the retransmission since whatever situation caused a packet to be missed would probably manifest itself again. Packets are most likely to be dropped when the receiver's buffers are full. This situation will develop if there is lots of broadcast traffic or if the receiving machine is active with several message exchanges.

To avoid this swamping problem the simple and proven solution of explicitly acknowledging each data packet was adopted. Should the sender not receive an ACK for the last packet then after a "suitable" time period the data packet would be re-transmitted. The receiver still uses the sequence number to make sure that duplicate packets, which could result if the receiver were a little slow with the ACK or if the sender missed the ACK, are ignored. With this stop and wait protocol a single bit, either zero or one, can be used to establish the correct order of transmitted packets and a full packet sequence number is not required. However, the full sequence number has been kept in case future protocol enhancements should require it and to aid in the debugging effort.

In this protocol implementation up to 1432 bytes of user data can be sent in a data packet, thus the majority of remote IPCs involve the sending of only one such packet. In the situation where only one data packet is sent some form of ACK is mandatory to ensure to the sender that the recipient got the packet. The simplicity of the stop and wait type protocol allows it easily to cater to messages that involve the transmission of single or multiple data packets. It would be possible to extend this protocol to a sliding window type but the additional code complexity and overhead currently cannot be justified given the small number of message transfers in which performance might improve.

Termination Phase

A termination phase is required by the protocol to insure that both parties to the transfer agree that the message exchange is complete. In the error free case when the last data packet is sent the receiver transmits the ACK and the sender receives it. In this situation there is no problem since the arrival of the ACK confirms that the last data packet got there. This means that both communications managers can report to the kernel that the message transfer is complete thus allowing the kernel to unblock the processes. The world, however, is seldom ideal and inevitably on some occasion the last ACK will be lost. Waiting for ACKs of ACKs doesn't help since once again the last ACK could be lost and the original problem has reappeared in a slightly masked form.

The solution to this dilemma was to have a communications manager report a message exchange done as soon as it knows for sure that all the sent data has been accepted. For the receiver this means as soon as the last data packet has been processed. (The last data packet can be determined since the amount of data to be sent and how much data can be placed in one packet are known and, furthermore, the last data packet is of a different packet type then regular data packets.) When the ACK to the last data packet is received then the sending site can release its process too. If the ACK is lost then the sender will retransmit the last data packet to a destination process that was unblocked and hence not expecting this packet. There are three states of interest that this destination process could be in:

1. The process may no longer exist.
2. The process may be executing its code.
3. The process may be blocked on another remote IPC.

In these situations the receiving communications manager reports the destination processes state and lets the sending party decide what to do. If the reported status indicates that the process doesn't exist or is executing then the assumption is made that the last data packet was delivered. Should the process not exist then it was either killed while the last packet was received or right after it received the packet. Since there is no way to determine this when an ACK is lost the assumption is made that the packet was delivered and then the process died. If the returned status indicates the process is executing, then the assumption is made that it is executing because the "last data" packet resulted in the process being unblocked.

The problem is slightly more complicated when the destination process is unblocked and then immediately issues another remote IPC resulting in it again being blocked. To distinguish this new IPC from the old one a unique session number is assigned to each communication and forms part of the Shoshin protocol header. In the situation in which the receiver immediately does another remote IPC the sequence numbers will be different and an appropriate response is returned. The assumption is made that for the same process to be doing another remote IPC it must have completed its previous one. By their very nature blocking type IPC primitives imply that each process may have only one outstanding communication at any one time.

In some situations the assumptions about message completions may not provide a sufficient guarantee of message delivery. Under these circumstances it would be expected that some level four applications type protocol would be implemented.

4.1.3 Shoshin Packet Format

To accomplish the reliable exchange of messages and to detect the situations described above each outgoing datagram has a Shoshin header affixed to it. The header format is identical to that used in the original Shoshin [TOKU84] except for the addition of session numbers [Fig 4.3].

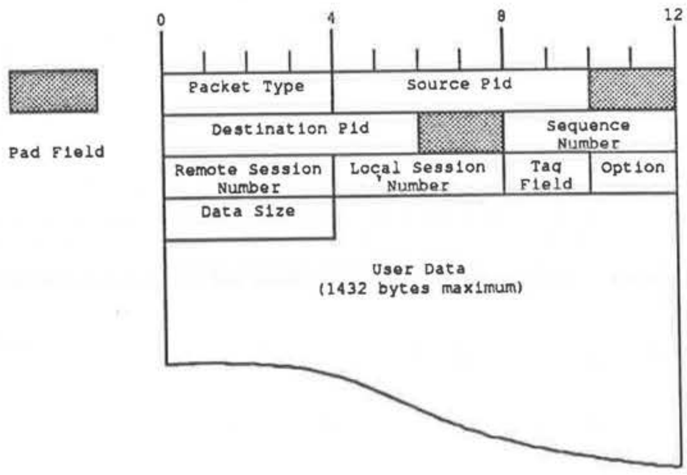

Figure 4.3: Shoshin Header

The header is 40 bytes long and consists of the following fields:

- The packet control field that identifies the type of packet. (4 bytes)
- Source and destination process identifiers. (6 bytes each)
- The packet sequence number. (4 bytes)
- Session numbers used by the source and destination hosts to ensure that they agree on what message is being sent. (4 bytes each)
- The tag field from the IPC primitive. (2 bytes)
- The amount of data being sent. (4 bytes)

An additional six bytes of padding are needed to maintain compatibility between the header packets generated by the communications managers running on the SUNs and VAXes. This is required because the C compilers used on the two machines treat the alignment of halfwords (2 bytes) differently. Within each field the most significant bit is always the leftmost bit. The last pad field is labeled as an options field and is used to provide information about the capabilities or lack of them with respect to the various communications manager implementations. Initially this information was not used but as more machines, some with different communications manager, entered service the options field came into use.

4.2 The UNIX Implementation

As indicated in section 3.3 it was decided to implement a communications manager on UNIX to take advantage of development tools such as debuggers and profilers. Also mentioned was the advantage of providing processes running on Shoshin with the ability to communicate with a UNIX system. However, to accomplish this a method had to be found for propagating the packets a Shoshin system places on the Ethernet up to the UNIX communications manager.

Since user processes don't have direct access to the Ethernet they must rely on the networking software to accept and deliver the incoming packets to them. The approach to solving this problem was based on using the standard networking facilities available in 4.2BSD UNIX.

This arrangement forces a number of restrictions on the access to the net since there are a limited number of protocols which are supported by the networking software. [Fig 4.4]. The protocol identification field of the Ethernet header is used to

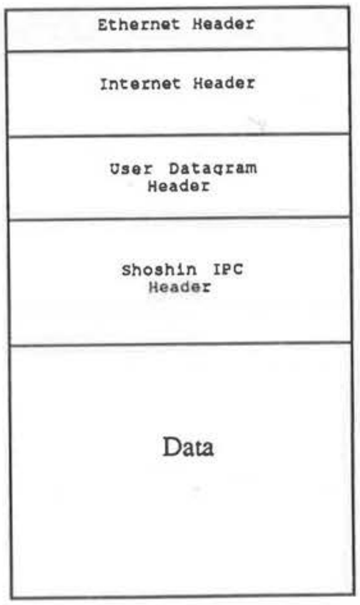

Figure 4.4: Protocol headers used

inform the receiving software as to how it should interpret the data (protocol) which follows. In a pure Shoshin to Shoshin environment the protocol type will identify the following packet as a Shoshin packet. However, to a UNIX system the Shoshin type is meaningless so any packets destined for such a machine must use a protocol that the 4.2 BSD systems are conversant in. An additional problem on the UNIX system is ensuring that the arriving datagram is delivered to the correct process which, in this situation, would be the UNIX based communications manager.

The basic datagram service in 4.2 systems [LEFF83] is provided by the Internet Protocol (IP) [POST81] which is the datagram protocol used by the ARPANET. The ARPANET is a long haul network consequently much of the information contained in an IP header is directed at solving the problems unique to that type of a network. This also means that most of the information contained in this header is not required for a local area network. Among other things, the IP header contains source and destination addresses along with a protocol type field. In many respects this is much like the Ethernet header except that the significance of the address field is slightly different. The Ethernet addresses really specifies one part of the route to take to get to a physical machine. It does not imply that the ultimate destination is at that host although on a local area network it usually is. (In fact a pure Shoshin to Shoshin exchange assumes that it is.) The address in the internet protocol header is slightly different in that it identifies the final destination host and not an intermediate destination site which, in a long haul network, is what the next site might be. Just as in the Ethernet header the protocol identification field identifies the protocol being used in the data which follows.

Although the internet protocol provides a mechanism for identifying which host a message came from and where it is going to, it says nothing about which process on the destination or source machine the packet is associated with. By introducing a port number the the User Datagram Protocol (UDP) [POST80] provides a mechanism that allows a datagram to be associated with a process. If a process wants to use UDP under UNIX then that process issues the **socket** and **bind** [LEFF83] system calls to associate the process with a port. A process can specify the port it wants to use or it can have the system select one from a pool of port numbers currently not in use. The one to one mapping between processes and ports is enforced by allowing only one process to be associated with each port. To allow the communications managers on the UNIX machines to interact with their Shoshin based counterparts a well known port number is always selected by the UNIX communications manager. In addition to making it easier for the Shoshin communications manager to determine where the the communications manager on UNIX is, it also provides a mechanism that ensures that only one communications manager will be running at each UNIX site since only one process can be associated with each port.

The final header is the Shoshin header which takes the form described earlier. The Shoshin header is then followed by up to 1432 of user data. The 1432 byte limitation is arrived at by determining the maximum amount of header overhead and subtracting that from the the maximum size of an Ethernet packet. Numerically this is $1432 = 1514(standardpacket) - 14(Ethernet\ header) - 20(IP\ header) - 8(UDPheader) - 40(Shoshinheader)$. In a pure Shoshin environment the 28 bytes of IP and UDP header are not transmitted and could be replaced with user data. The penalty for not doing this is that on occasion a message may be fragmented into two packets when one would have been sufficient. Since most messages tend to be small there would be a very limited performance improvement at the expense of simplicity in both the code and understanding in the data transfer portion of the protocol.

As the Shoshin development effort proceeded a mixture of communications managers with different requirements began to appear on the network. On one occasion a machine would be running a pure Shoshin system while on the next it would be running a communications manager under UNIX. One implication of such a dynamic environment is that if a Shoshin machine wants to communicate with a UNIX machine then it must know which machines are running what operating system so that the appropriate protocol headers can be placed on the outgoing packets. The simple solution would be to run all the protocols all the time but that places a needless performance penalty on a pure Shoshin to Shoshin communication. Instead, one of the original pad fields in the Shoshin header was turned into an options field so that communications managers could tell each other what level of protocols they supported. The initial packets of a message exchange are sent with the full *User Datagram* and *Interenet* protocol headers since all machines, be they UNIX or Shoshin, support these protocols. In a pure Shoshin dialogue the IP and UDP headers will be dropped as soon as the sender of a packet knows that the receiver can process packets without the headers.

A possible objection to this scheme concerns the placing of knowledge about what lower level protocols are used directly in the Shoshin header which implies that the remote IPC protocol knows about the capabilities of the lower levels. This violates the concept that protocols at one level should only make use of the services of lower level protocols and should not be instructing them on how to prepare a packet. In an ideal situation the software that adds the IP and UDP headers would maintain a table which would provide information on the protocol requirements of the various machines on the the network. However, the problem here is that the IP and UDP headers do not contain any fields that would allow one site to determine this directly so some sort of communication manger to communication manager protocol similar to the Address Resolution Protocol (ARP) [PLUM82] would have to be adopted. In the implementing of an ARP type protocol, a protocol would have to be designed and internally a table of machines and protocol capabilities would have to be maintained. This type of arrangement would add greatly to the communications manger's processing load. Currently, to manage a remote IPC the communications manager keeps a list of outstanding communications. Some of the information in this list pertains to the remote host so this provided an existing data structure that could be used to store additional data about a remote host. A further advantage is that this data structure has to be looked up for each incoming or outgoing packet anyway so no new processing overhead for table lookups is incurred to manage this information. This is an important consideration especially if there are a large number of hosts on the network using the Shoshin remote IPC protocol. In the final analysis it was decided that although the implementation of an ARP type protocol would be conceptually cleaner it would also be the more difficult and time consuming to implement. As a result conceptual pureness was sacrificed for ease and speed of implementation.

Through the use of these various protocols it was possible for packets to be exchanged between communications managers on UNIX and Shoshin systems. Since the UNIX communications manager was to be a process's sole access to the remote machines some method had to be found to link the UNIX processes with their communications manager. However this was to be accomplished, the interface in the user program would have to have the same syntax and semantics as the message primitives in an actual Shoshin environment. By adopting this approach it is then possible to develop programs in a UNIX environment that would be easily transportable to a *real* Shoshin machine and vice-versa.

In earlier versions of UNIX the obvious way to communicate information between processes would have been to use pipes [RITC74]. This places a very serious constraint on flexibility in that all the processes must be "relatives" and the pipes must be set up by a common ancestor prior to their use. Restrictions like that hardly make for a flexible and dynamic working environment. To overcome this problem **AF_UNIX** [LEFF83] sockets which allow unrelated processes, in the sense of parent child relationships, to dynamically establish a reliable stream oriented communications channel between each other were selected. With this construct the purpose of the IPC primitives like **bsend** then became that of imposing a "message" type abstraction on top of the actual stream based connection.

To make use of these functions the only requirement is that a UNIX process establish a connection with the communications manager on its machine through the use of the special **LinkComm** function. This function takes no arguments and returns a one if the request succeeded and a negative value indicating the nature of the problem if it failed. With a few programming tricks it would be possible to eliminate the call to **LinkComm** but this would obscure the fact that this is a UNIX based process. During the development phase such distinctions are important as they provide a clue as to which system the program is running under and, unfortunately, such knowledge could be essential for debugging.

When the UNIX communications manager starts up it must go through an initialization phase just like its Shoshin counterpart. It must open an Internet datagram socket on a well known port so that data can be exchanged with other real or simulated

Shoshin machines on the Ethernet. The final initialization action is to create a general purpose AF_UNIX socket on which to accept connection requests from user processes.

Once a user program has connected to the communications manager a well defined set of messages are exchanged to communicate the information needed to conduct the remote IPC. Upon completion of a remote message exchange the status of the transmission and, if required, any received data are returned to the process through the connection. The UNIX communications manager can handle several connections (the actual number is system dependent) from user processes simultaneously and can manipulate multiple outstanding remote IPCs at once just like its Shoshin based counterpart.

4.3 The Hardware Interface

As was mentioned earlier the communications manager is responsible for the system's interaction with the network hardware. The implication of this is that the actual servicing of interrupts is handled by the communications manager and not the kernel. Typically these interrupts indicate that the transmission of a packet has completed or that a new packet has arrived. Thus the major interactions with the network consist of manipulating the controller hardware to allow for the sending and receiving of packets.

In the original Shoshin the sending and receiving of packets were handled by separate *busout* and *busin* worker processes respectively [Fig 4.5]. Since it is essential that

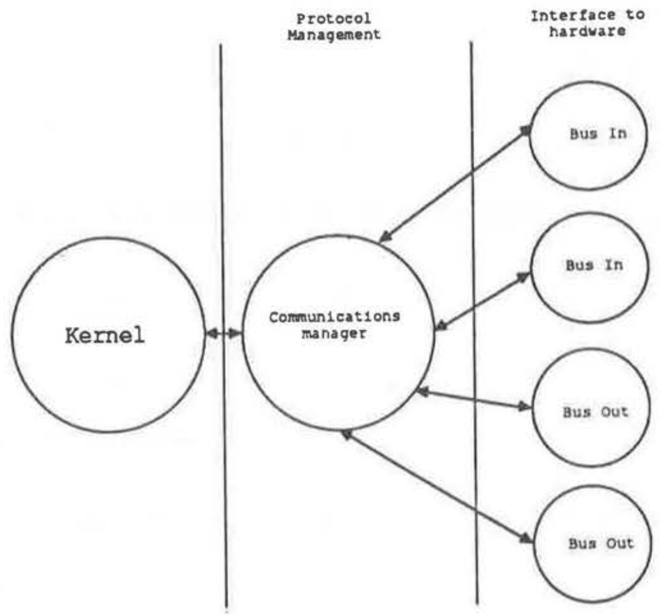

Figure 4.5: Communications manager - bus handler relationships

the communications manager not block, the processes are organized in a manager-worker relationship [GENT81]. The establishment of this type of relationship requires that the workers send a message to the communications manager using the **reply** primitive. The contents of the message indicate the status of the task the worker just completed and the arrival of the message implies that the worker is now free. When the communications manager has a task for a worker it unblocks one of the free workers by composing a message that indicates the type of task to work on and then issues the the non-blocking primitive, **reply**, to the selected worker. In using this design structure the communications manager is left free to process requests while the workers look after the actual network I/O activities. To complete its assigned task the worker does some processing and then blocks waiting for the interrupt which signals that an I/O request, either a transmit or receive depending upon the type of worker, has finished. When the I/O completes the worker is unblocked, performs any necessary computations, and then notifies the communications manager of the result using a **request**.

This type of structure is acceptable provided that the kernel can determine the exact nature of the interrupt. With the SUN Workstations this is not possible since interrupts are broken up into levels and there may be several different devices capable of causing an interrupt at any one level. This makes it very difficult for the kernel to determine exactly what device caused the interrupt, let alone the exact nature of the interrupt unless the kernel has a detailed knowledge of each device and how it is controlled. To incorporate this information in the kernel would compromise the multi-process structuring philosophy of removing this type of knowledge from the kernel and placing it in a separate process. In doing this the kernel can be made simpler and the knowledge about a device is then concentrated in one area. This makes it easier to think about the device, how it functions and how it is integrated into the rest of the system. Integrating the device into the system is also easier since the server process can create a virtual device and then provide a well defined interface for this abstraction to the other processes.

In Shoshin knowledge of the the Ethernet controller is concentrated in the communications manager. Since the SUN Workstation does not provide for a detailed specification of interrupts the concept of waiting on an I/O activity in the original

Shoshin has been eliminated from the user process domain and moved into the kernel. (Although the communications manager has access to some special functions it is still basically a user process.) In place of the I/O wait the communications manager receives an ordinary message from a special kernel team process that has been associated with it. This kernel process blocks waiting for an indication of an interrupt and then sends a message to the communications manager.

By noting who the message is from the communications manager determines that it is an interrupt signal and can then enter a processing phase to ascertain the exact nature of the interrupt. This processing phase is required since the message just indicates that an interrupt has occurred and does not provide details as to the type. It is at this point, after the exact nature of the interrupt is known, that an important design decision had to be made. Should the communications manager have worker processes to handle this processing or should it do it immediately?

The decision was made to do it immediately instead of sending messages. The justification is that the overhead incurred by having to send and receive these messages would contribute significantly to the elapsed time of a remote IPC. In a message transfer at least four packets are exchanged and if workers were used then eight messages would have to be sent. From figures in the next chapter this represents about eight milliseconds of elapsed time. It would be expected that some concurrency with packet transmission would be possible so only part of this extra processing would show up as added elapsed time. Furthermore, the structure of the hardware controller is such that sharing it does not make logical sense. The control of the transmitting and receiving functions requires access to the same hardware control registers so there is not a well defined logical distinction between these two operations that would facilitate assigning them to separate processes.

Another consideration for the communications manager is to minimize the movement of data. For example when a data packet is received the data must be moved from the reception buffer to the destination process's address space and likewise the reverse is true for the sending of a data packet. The copying of these data packets can be a relatively expensive operation and should be minimized. By having the communications manager service the interrupts directly some extra copying can be avoid since the data can be moved immediately from the receiving buffer to the destination process. When a packet is received it is the software's goal to process the interrupt and have the buffer ready to receive a new packet as soon as possible. If a separate process is assigned the task then extra message sends or data copying is required to free the receiving buffer as quickly as possible. However, both of these operations involve extra processing that can be eliminated if the communications manager does the task directly and immediately.

It is possible that while a transmission is in progress that a user request or another packet could arrive that would generate, as a response, a new packet to be sent. To avoid blocking while waiting for the transmission to complete a queue of messages to send is maintained. When a transmission completes this queue is checked and if it is non-empty the first packet is sent. This arrangement allows for concurrency between the transmit operation and other communications manager functions.

The result of all these considerations is a communications manager which accepts messages and then takes the appropriate action. In some situations these messages are requests to handle interrupts and in others they are requests to send messages. The implementation stresses the minimization of messages and data copying.

Chapter 5

The Results and Analysis

With the complete Team Shoshin tested and running a performance evaluation of the various IPCs was undertaken. Such an evaluation provides insight into how the various parts of Team Shoshin interact and function. Through a careful analysis of these results it is possible to more accurately determine the weaknesses and strengths of the system while at the same time proposing ways to enhance system performance. A performance evaluation also provides a yardstick that can be used to compare different systems provided a common measurement method is used. Carefully contrived laboratory expirements may provide some insight into a system but they do not demonstrate that a system is functional or practical. To display this a simple file server is developed that allows Shoshin processes to read and write files on a UNIX system. This successfully demonstrates the operation of both the UNIX and Shoshin based communication managers and provides a small example of how to write distributed programs.

5.1 Performance Measurements

The performance measurements can be broken down into four distinct categories depending upon whether message exchanges are between:

- processes on the same team,
- processes not on the same team but on the same physical processor,
- processes on separate machines that are both running the Team Shoshin operating system, or
- processes that are on separate machines one of which is running UNIX while the other is running Team Shoshin.

Each category can be further subdivided into the typical message exchange patterns of <**bsend,brec**>, <**bsend,brecany**> and <**request,brecany,reply**>.

5.1.1 Local IPC Measurements

In a multi-process structured operating system actions are coordinated through the use of messages. Consequently the efficient operation of the system can at least be partly attributed to the efficient implementation of the local IPC primitives. To try and ascertain the effects of IPCs on the operation of the communications manager the performance of local IPCs is measured. The timing for all local IPCs is done in the same manner and follows the general form of the code outline given in [Fig. 5.1]. The IPC command issued is usually something like **bsend** which can be called a producer type process since it continually produces messages. To avoid having this process block the destination process is in an infinite loop doing nothing but consuming the

```
main()
{
  ....
  gettime(start_time);
  for( i = 0; i < 10000; i++){
    /* Issue IPC command */
  }
  gettime(end_time)
  ...
}
```

Figure 5.1: Local IPC timing method

messages being sent by the timing process. One ramification of this set up is that the reported times include some overhead associated with the scheduling and executing of the consumer process which is not related to the actual function of sending or receiving messages. The resolution of the system clock is too coarse to allow for the timing of individual local IPC exchanges.

The performance figures reported are exactly those that are obtained from the example code fragment. The overhead involved in making the **gettime** system call and for loop control have not been deducted from the reported figures. The average overhead for the **gettime** call is 1.07 milliseconds which is insignificant when it is amortized over the 10,000 IPC calls.

All the measurements were made using a typical Team Shoshin system configuration which includes a fully operational communications manager. Since there is a certain amount of broadcast traffic from other machines on the network, some of the resulting non-trivial processing overhead is inevitably charged to the local IPCs. This approach is reasonable since it provides an indication of what type of response user programs can achieve with the normal system configuration which would be subject to this overhead.

Table 5.1 reports the message exchange times for the **bsend brec** pair. In all

Message size (bytes)	Same team elapsed time (ms)	Separate processes elapsed time (ms)
0	1.03	1.83
16	1.09	1.90
64	1.16	1.97
128	1.26	2.05
256	1.43	2.22
512	1.60	2.40
1024	1.92	2.72
2048	2.59	3.38
8192	6.53	7.34
10240	7.83	8.64

Table 5.1: bsend $\Longrightarrow$ brec 10,000 iterations

cases two sets of measurements are provided, one for processes on the same team, and one for processes on different teams. The significance of this is that messages passed between members of the same team require fewer context switches since they are sharing code and data spaces. As a result of this shared environment the passing of messages between team members is faster than between separate processes. However, this advantage is independent of the message size and has a constant value of about 0.80 milliseconds.

Message size (bytes)	Same Teams elapsed time (ms)	Separate Processes elapsed time (ms)
0	1.08	1.87
16	1.15	1.95
64	1.21	2.03
256	1.47	2.26
512	1.65	2.44
1024	1.99	2.77
2048	2.64	3.44
8192	6.58	7.37
10240	7.90	8.68

Table 5.2: bsend $\Longrightarrow$ brecany 10,000 iterations

Table 5.2 reports the values for the **bsend brecany** pair. Notice that these times are slightly longer than when the receiving processes is doing just a brec. The penalty for doing a brecany over a brec is independent of the size of the message and is about 0.06 milliseconds. This penalty is the same whether or not processes are on the same team and is due to the the extra code that has to be executed to deliver a message to a process doing a **brecany**.

The statistics in Table 5.3 refer to the **bsend**, **brecany** and **reply** triplet. In these measurements the size of the message sent in the request determines the size of the reply message. For example a request message of 1024 bytes results in a reply message

Message size (bytes)	Same teams elapsed time (ms)	Separate processes elapsed time (ms)
0	1.74	3.38
16	1.87	3.52
32	1.91	3.56
64	2.00	3.64
256	2.52	4.16
512	2.87	4.51
1024	3.54	5.17
2048	4.84	6.49
8192	12.74	14.38
10240	15.38	16.99

Table 5.3: request $\Longrightarrow$ brecany $\Longrightarrow$ reply 10,000 iterations

of the same size for a total data transfer, in this example, of 2048 bytes.

The <**request,brecany,reply**> message exchanges can be modeled by substituting the message exchange sequence of <**bsend,brecany**>, and <**brec,bsend**>. A summing of these constituent parts would yield, for a 1024 byte message, a total elapsed time of 3.91ms versus 3.54ms for request. The time difference between these two methods remains constant at about 0.37ms over all message sizes. When a process does a request it remains blocked until the corresponding reply is issued whereas that is not the case for the alternate method provided above. The <**bsend,brecany**> approach results in the requester being unblocked which leads to extra context switches, additional scheduling overhead and an extra system call not present when **request** is used.

Similar figures for elapsed times in the original Shoshin [TOKU84] are presented in Table 5.4. With the newer hardware it is to be expected that Team Shoshin would outperform its LSI 11 counterpart. Team Shoshin is typically 4.5 to 5.5 times faster than Shoshin for all classes of local IPCs and this can be attributed to the newer hardware and to the restructuring of the kernel. It is estimated that without restructuring the kernel the message exchange times would be about double those reported. The rest of the improvement in message exchanges is due to the hardware.

Message size (bytes)	Elapsed time (ms)
16	5.6
64	5.8
256	6.7
512	7.7
1024	9.9
2048	14.2

Table 5.4: bsend $\Longrightarrow$ brec in the original Shoshin

It is a slightly unfair to compare the results of those on a SUN Workstation to those on an LSI 11. A more reasonable comparison would be against the V Kernel [CHER83] running on a SUN Workstation with a 10 MHz Motorola 68000. In performing this comparison, it is essential to ensure that the selected primitives provide the same functionality to the user programs. The basic V Kernel primitives of Send, Receive and Reply correspond to Shoshin's **request**, **brecany** and **reply**. However, in the V Kernel the message sizes are fixed at 32 bytes and if one wants to move more than that then the MoveTo and MoveFrom primitives must also be used. For a message of 32 bytes the V Kernel requires 0.77ms to Shoshin's 1.91ms. By summing the various components required to exchange 1024 bytes of data (in both directions) it is estimated that the V Kernel would require 2.67ms to Shoshin's 3.54ms. The figures reported for the V Kernel have the loop control overhead and other "artifacts" removed from the reported times. This accounts in part for Team Shoshin's IPCs taking longer than the V Kernel's. Additionally the above estimate does not take into account the extra programming required by the user to manipulate and check the extra IPC calls. Furthermore, if the processor time required to schedule the processes is considered an "artifact" along with the overhead associated with getting the time then the difference in performance is reduced even more.

5.1.2 Remote IPC Measurements

With the Ethernet being a shared resource there will naturally be a certain amount of contention for it. In an attempt to minimize this problem all remote IPC measurements were made at anti-social times when the Ethernet is lightly loaded. There will always be some extraneous traffic on the net from other machines and this introduces a certain amount of variance into the measurements. To try and reduce this variance a number of runs of N iterations ($N > 1000$) were performed for each message size and the lowest average time for a run is reported.

Message size (bytes)	bsend - brecany (ms)	request - reply (ms)
0	30.2	62.8
16	32.4	64.3
32	32.3	64.2
512	36.2	69.3
1024	38.8	75.6
2048	55.4	108.7
8192	134.1	266.2

Table 5.5: Remote IPC elapsed times, Shoshin to Shoshin

Table 5.5 reports the figures for pure Shoshin to Shoshin communication without the Internet and UDP headers. For the purposes of measurement the options field in the Shoshin header was not used and all communications were assumed to be between machines running Shoshin. This eliminated the extra computation associated with building the IP and UDP headers for the initial packets. The measurement technique is the same as that for the local IPCs except that the time is measured for an individual IPC operation and summed over the N iterations instead of just recording the total elapsed time. In this situation the overhead for the call to **gettime** (1.07ms) becomes significant since it is an artifact in the measurement of the elapsed time for each message send. The figures for the <**bsend,brec**> are so similar to <**bsend,brecany**> that they are not reported. This should not be unexpected because the extra time associated with the code for a **brecany** (0.06ms) is unnoticeable given the magnitude and the amount of variance associated with the elapsed times for a remote IPC.

One observation about this data is that 16 byte messages take longer to send than 32 byte ones. This relationship existed in all the sample runs not just the one reported. It has not been determined why this is the case, but one possibility is that it has something to do with attempting to send packets on the Ethernet that are near the minimum packet size of 60 bytes. Another possibility is that due to the size of the message and the amount of time needed to transmit it that more concurrency (as realized through interrupts) is possible for the longer messages and this results in a reduced elapsed time.

The numbers reported here represent a 2.9 to 3.5 times increase in performance over the original Shoshin. The performance increase is greatest for large messages since fewer data packets need to be exchanged in Team Shoshin due to the larger packet size. This improvement is not as great as that for the local IPCs probably due in part to the lack of hardware ACKs and the resulting changes required in the protocol to generate and accommodate software acknowledgements.

A comparison of the remote IPC times with the V Kernel's indicates that the protocol for exchanging messages could use some tuning. To send 32 bytes on a 3Mbs

Ethernet the V Kernel requires 2.54ms (3Mb Ethernet) compared to Shoshin's 32.3 ms and for 1024 bytes similar figures are 18.54ms (estimated) and 75.6ms respectively. These numbers support Cheriton's claim that remote IPCs implemented through a server process introduce a four fold increase in elapsed times. However, a closer examination of the remote IPC costs in Team Shoshin suggests that it should be possible to reduce elapsed times considerably without moving the communications manager into the the kernel.

As a basis for discussing the large difference in elapsed times consider that to send a zero byte message, using the minimum number of four packets, requires 30.2ms or roughly 7.55ms per packet. If the initial CTS and RTS packets could be eliminated then the message exchange times should be reduced by about 15ms each. These packets could be dispensed with if the RTS and first data packet were combined. To accomplish this receiving communications manager would, if possible, store the data until the receiving process was ready. Once the remote communications manager has accepted the data then the sender would only have to enter a process checking phase until an acknowledgement indicating the data had been delivered to the destination process was received. The acknowledgement would then serve as the signal to begin transferring any remaining data.

An examination of the execution costs of sending a packet reveals that the transmission of a packet consists of the following areas of interest:

1. The costs of sending messages to coordinate the various aspects of the transmission.
2. Processing overhead to operate the controller.
3. The cost of moving data from the user address space to the communications manager and then, when ready, to the transmission buffer.
4. The actual transmission time on the network.
5. The processing overhead required to run the protocol and build any necessary headers.

Since the communications manager is implemented as a process, information on the local machine is conveyed between processes using messages. To transmit one packet a minimum of three messages must be sent to other system processes with a resulting overhead of about 5.6ms. (One message to notify the communications manager to transmit a packet, one to notify a process to wait for the transmission interrupt and one to indicate that an interrupt (transmission) has taken place.) The processing overhead of operating the controller and acquiring the ether was computed to be a constant value of about 0.52ms. This was computed by measuring the time required to send an N byte packet and then subtracting how long it would take to transmit that packet on the raw Ethernet. The time remaining is then the time needed to manipulate the controller and acquire the ether. This value was confirmed as a constant when measurements over a variety of different sized packets resulted in a constant factor being computed for the overhead.

A similar technique was used to determine the costs associated with the movement of the data and for a 60 byte packet this is 0.15ms. When the user has a non-zero length message to send then the data must be first copied to the communications manager's address space in case the full IP and UDP headers and associated data checksums are required.

The actual transmission time on the network for the minimum sized 60 byte packets is 0.05ms. However, the actual transmission proceeds concurrently with other communications manager functions and does not contribute to any added delay. The remaining 1.28ms of the 7.55ms can be attributed to the costs associated with manipulating the protocol and maintaining message states. This actual value may be a little low since there is the possibility of concurrency in the remote IPC processing between the sending and receiving sites.

By far the largest component in the cost associated with the sending of a packet is that of the local messages. Thus if one could eliminate two of the messages then the message transfer time might drop by as much as fifty percent assuming that other costs currently hidden by the simultaneous execution of the two communications manager do not appear.

At least part of the poor performance of Team Shoshin with respect to the V Kernel results from having the communications manager being a process separate from the kernel. This results in extra context switches and message passes that contribute significantly to the overhead of transmitting a message to a remote machine.

For large data transmissions the stop and wait protocol does not make good use of the high channel bandwidth.(Maximum measured data transfer rate between processes on separate machines was 492Kbs with 8192 byte messages .) One solution would be to provide an optimized protocol for *large* data transfers. However, such a protocol should not be implemented by adding new IPC primitives. The parsimony and flexibility of the current set of primitives should not be compromised by introducing new primitives that would require the user to be aware of the size of the message to be sent. Before modifying the protocol extensively, consideration must also be given to how important a fast remote IPC need be. For example if most applications spend several seconds waiting for their counterpart to become ready to receive a message then the reduction by even 1.0ms in message elapsed time is insignificant. However, if processes are tightly coupled and spend most of their time sending messages then it would be important to reduce message elapsed time. Highly utilized system servers would fall into this last category since it would allow them to complete more requests in a given period of time.

Measurements for Shoshin to UNIX have also been made and predictably they are worse than Shoshin to Shoshin by from thirty to sixty percent over the chosen message sizes. The performance of the Shoshin to UNIX communications appears to be a function of the message size. As the number of packets that must be exchanged to transmit a message increases the performance deteriorates with respect to the pure

Shoshin to Shoshin communication although the data transfer rate is increasing. (That is, its transfer rate is not increasing as fast as pure Shoshin's is.) Most of this extra overhead can be attributed to having to build the IP/UDP headers and to the inefficient method, compared to Shoshin, of moving data between the communications manager on UNIX and the processes it is serving. In a pure Shoshin environment timings indicate that the adding of the IP and UDP headers extracts a performance penalty in excess of 20%, primarily due to the extra burden of computing checksums.

A brief comment on the size of the communications manager is also in order. Both the UNIX and Shoshin communications manager consist of slightly more than 16,000 lines of commented C code. Some of this is conditionally compiled depending upon where the communications manager is expected to run. In both implementations about 11,000 lines of the code are devoted to handling the protocol. In Shoshin there are then a further 5,000 lines required to handle the controller and the implementation of UDP and IP. On a UNIX system the 5,000 lines are used to simulate the Shoshin IPC primitives with the 4.2BSD UNIX IPC facilities and to provide access to the network. This code results in about 26Kbytes and 47Kbytes of optimized code for the Shoshin and UNIX communications managers respectively.

There are several components of the communications manager that contribute significantly to the code. One is the routines required for I/O operations to the console which is where any major system problems that are detected are reported. Another major component concerns the routines that manage the information about a communications status. In many respects this is a duplication of information that is already maintained in the kernel. Since the communications manager is a separate process it does not have access to this information and must maintain and duplicate it. Naturally a large component of the code deals with error handling and is seldom executed.

5.2 A Sample Distributed Program

As a demonstration of the viability of Team Shoshin and the various IPC primitives a simple file service program was developed and implemented. The implementation consists of two parts. One part resides on a UNIX machine and provides access to the UNIX file system. The remaining part of the system is a set of functions that user processes can invoke that allow file access requests to be issued to this server.

Even though the file system was intended primarily as a demonstration of Shoshin's IPC capabilities an attempt was made to try and present to the user programs an interface that could be made to be uniform over a wide variety of I/O resources. The byte stream I/O model was selected to form the basis of how the I/O system would appear to the user program. Such a system simply presents bytes as a stream to the applications program and does not interpret them. One of the nice things about this approach is that it can cater to a wide variety of I/O objects from terminals to files and network connections.

In such an environment where a number of different "things" are all suppose to appear logically the same the concept of a file really isn't that applicable. An attempt to provide this logical consistency is achieved through the development of the notion of object oriented I/O [CHER81]. With this approach "things" are objects and a set of functions are provided to manipulate and work on the objects. As in UNIX, it is the naming scheme that ultimately determines what the object is and how to go about opening it.

Functions are provided to open objects, close objects, read objects, write objects, seek on objects, and get information on objects. Since there are a number of different type of objects some functions, like seek, only make sense to use on certain objects.

Insight into how the I/O system operates is best provided by examples of what happens when the various I/O functions are invoked. When the user requests that an object be opened the routine first identifies the server for that class of object. In the current implementation that is accomplished by parsing the name of the object the user wishes to open. In the future it is hoped that this would be accomplished by querying a name server to map the symbolic object to the appropriate server. Once the location of the server has been determined a message, using **request**, is dispatched to the server with the information required to open the object. The server responds with a status indicating how the operation went along with some object specific details. The function then builds an object descriptor to store the object information that is needed to perform actions on the object.

Once the object is opened there are numerous types of requests that can be issued to manipulate it; as an example consider what happens when a read takes place. When the "read object" function is called the function checks to see if the local buffers, which are part of the user's address space, can supply the data. If they cannot then a request is made of the server to send more data. The key concept in the operation of this system is that the user program is a client which through supplied routines issues requests of a server to perform specific operations on a particular object. In essence this is a level three protocol based on the IPC service. The use of the standard IPCs as the communication mechanism between client and server means that it is irrelevant what machines the processes are on since the difference between a local and remote communication is transparent to the processes involved. Part of the power with such a configuration is that as long as a server can be located, perhaps through a name server, then requests can be processed and servers can move around without affecting client programs providing, of course, that the servers keep the name server database up-to-date. Similar actions take place for writes, only in the reverse order.

As its "title" implies the server process for a particular object just services requests. On reception of a message it examines the message to see what is being requested and if possible completes the request. In certain circumstances the server may institute authentication checks to insure that the requester is allowed to have access to the object. The final action of the server before returning to its wait state is to reply with a completion code and if needed the results of the request.

The current implementation recognizes two types of objects, terminals and files. The name of the object determines if it is a terminal or file and for a file the name also specifies what machine the file is residing on. In this implementation the file servers run on UNIX machines and provide access to their local files by utilizing the standard UNIX I/O routines. To accomplish this the server must keep a list of file descriptors and associated processes. Under UNIX there is a maximum number of files that any one process may have open so this places an upper limit on the total number of files that clients may be working on.

Distributed programs of this nature present some unique problems not found in classical systems. In this implementation no attempt has been made to solve the problems, only to become aware of them. One major problem develops when a client process dies before it has had an opportunity to close its files. Since the server is unaware of the death it never releases the resources acquired by that client. This could be overcome by having a "vulture" process watch over each client with an open file. The vulture would issue a **brec** to the client and since the client knows nothing about this process it will never send to it. However, as soon as the client dies the **brec** would terminate with a status indicating the process was not alive and the server could be notified so that the appropriate cleanup could take place. One problem with this approach is that it places an extra load on the communications manager and network as Are You There packets and the responses must be generated. If there are lots of open files this would result in a large amount of overhead that could seriously hamper performance. If a function like `pexist`, which checks for the existence of a process, were extended to execute across machines then it could be used to check for dead processes.

The reverse problem of a server dying on a client also exists but the ramifications are not quite as severe. In this situation the assumption can be made that the files have been closed and appropriate steps can be made to reclaim local resources. Some sort of error code would also have to be returned to the user program but dealing with it is no different then handling file access problems in conventional systems.

Even though the file access portion of the I/O system has some potentially serious problems it has proven to be a very useful tool. For example it was used to log the data that was collected by the performance measurement routines. This proved to be invaluable as it allowed long numerous performance runs to be conducted in an unattended mode. To demonstrate the capabilities of the system other trivial programs such as ones to list and copy files have been produced.

This simple file server demonstrates that the communications managers of UNIX and Shoshin are capable of cooperating and that the protocol they implemented is reliable and workable. This rudimentary I/O system greatly increases the scope of the problems that can be tackled using the Shoshin system. As an example, it is now possible to consider the problem of dynamically loading programs. Currently when Shoshin is booted it must be loaded with all the user programs that are to be run. Dynamic loading would permit a more versatile and useful operating environment since arbitrary programs could be executed without including them at system boot time.

Chapter 6

Concluding Remarks

6.1 Summary

This thesis has presented a description of the communications model used in the Team Shoshin system with particular emphasis on remote interprocess communication and its extension to machines running 4.2BSD UNIX. As a demonstration of the usability of the system a simple object oriented I/O system is developed that allows access to files on a UNIX systems. Performance measurements of the various IPC primitives were also made to gain insight into how the system functions and to propose methods of improving performance.

The local IPC performance measurements for Team Shoshin have shown that it is possible to efficiently implement a set of general purpose IPC primitives with variable message sizes. By going to improved hardware and restructuring the IPC code Team Shoshin's local IPCs have become 4.5 to 5.5 times faster than the original Shoshin's. When the IPC's general purpose nature is considered they are comparable in performance to the similar V Kernel primitives.

The remote IPCs only run about 3.5 to 4.5 times faster than the original Shoshin and are considerably slower than their V Kernel counterparts. This has led to an analysis of the protocol which strongly suggests the initial setup phase should be combined with the transmission of the first data packet. Although the multi-process structuring of the communications manager provides a convenient implementation framework it extracts a hefty performance penalty through extra message exchanges and context switches. Currently the problem of restructuring the communications manager to eliminate some of the messages is being pursued. If this can be accomplished then a significant reduction in elapsed time should be achievable. A modification of the protocol to eliminate some of the initial overhead and to utilize the higher bandwidth of the Ethernet would also improve remote IPC performance.

The implementation of a simple file server has demonstrated both the local and remote message capabilities of the Team Shoshin IPC facilities. At the same time the file server provides a mechanism that can be used to develop new applications and features for Shoshin like dynamic process creation.

6.2 Future Work

The original purpose for developing Shoshin at the University of Waterloo was to provide a flexible and modular testbed for the investigation of distributed operating systems and programs. The porting and development work at UBC has established such a working environment and provided a body of information that can be used to guide future research efforts.

Team Shoshin can now serve as the basis for exploring other topics in the area of distributed systems. One class of problems of interest concerns the extension of existing services so that they become more functional and efficient. A prime candidate for work in this area concerns the simple file server. The interface to the user and the mechanism for issuing I/O requests is clean and flexible. However, the server portion of the system extracts a large performance penalty by first analysing the request and then issuing the appropriate user level system call. An alternative approach would be to integrate Shoshin into either SUN's network disk or file server protocols. The network disk approach would provide Shoshin with a virtual disk that it could read and write thus allowing Shoshin to implement its own file structure. The network file server would allow Shoshin to access directly the SUN's UNIX based file system using SUN's remote procedure call protocol. With both of these proposals serious consideration will have to be given to file protection and consistency.

To make many of the system services work reasonably in a highly distributed environment there needs to be some method of locating processes, especially ones which provide general services like printing or file access. To this end some form of a name-server to map well known names to processes is required. Issues such as how to deal with name conflicts and how to resolve the names of processes which are on other machines must be considered. Naturally the ramifications of having centralized versus distributed control of such a system must also be addressed.

Currently the only form of network access is provided indirectly through the basic IPC primitives and it is transparent to the user. It would be nice if the communication manager could provide a mechanism so that processes could access the network directly thus allowing them to implement their own protocols. (For example this facility could then be used to implement TCP or X.25) Further extensions to the network could include the addition of broadcast and multi-cast services. Incorporation of the broadcast and multi-cast facilities into the remote IPC protocol would provide new ways of initiating dialogues between processes.

In porting Shoshin to the SUN Workstation/Ethernet environment a flexible distributed software testbed has been developed and extended for a commonly available hardware environment. This can then serve as stepping stone to further research in the area of distributed systems.

Bibliography

[ANDR83] G.R. Andrews, F.B. Schneider, "Concepts and Notations for Concurrent Programming", *Computing Surveys,* vol.15 #1 pp. 3-43, March 1983

[CHER79a] D.R. Cheriton, *Multi-process Structuring and the Thoth Operating System*, Ph.D Thesis, University of Waterloo, 1979.

[CHER79b] D.R. Cheriton, *Interactive Verex*, Technical Report 79-1, University of British Columbia, September 1979 revised January 1983.

[CHER79c] D.R. Cheriton, *Zed Programming Manual*, Technical Report 79-2, University of British Columbia, September 1979 revised January 1983.

[CHER81] D.R. Cheriton, *Distributed I/O using an Object-based Protocol*, Technical Report 81-1, University of British Columbia, January 1981.

[CHER83] D.R. Cheriton, T.P. Mann, "The Distributed V Kernel and its Performance for Diskless Workstation", *Proceedings of the Ninth Symposium on Operating System Principles,* pp. 128-140 ACM, October 1983.

[DIJK65] E.W. Dijkstra "Solution of a Problem in Concurrent Programming Control", *Communications of the ACM,* vol.8 #9 pp. 569, Septemper 1965.

[DIJK68] E.W. Dijkstra "The Structure of the *THE*-Multiprogramming System", *Communications of the ACM,* vol.11 #5 pp. 341-346, May 1968.

[DEC80] Digital Equipment Corporation, Intel Corporation, Xerox Corporation, *The Ethernet: A Local Area Network - Data Link Layer and Physical Layer Specification, Version 1.0*, September 1980.

[GENT81] W. Morven Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", *Software Practice and Experience,* vol.11, pp. 435-466, 1981.

[HOAR74] C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM,* vol.17 #10 pp. 549-557, October 1974.

[KERN78] B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., 1978.

[LEFF83] S.J. Leffler, *A 4.2BSD Interprocess Communication Primer DRAFT of January 22, 1983,* Department of Electrical Engineering and Computer Science, University of California, Berkley, January 1983.

[LIU82] M.T. Liu, W. Hilal, B.H. Groomes, "Performance evaluation of channel access protocols for local computer networks", *Proceedings of the COMPCON FALL 82*, IEEE Computer Society, pp. 417-426, September 1982.

[METC76] R. Metcalfe, D. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM,* vol.19 #7 pp. 395-404, July 1976.

[MOTO84] Motorola, *M68000 16/32-bit Microprocessor Programmer's Reference Manual,* fourth edition, Englewood Cliffs, New Jersey, Prentice-Hall, 1984.

[NABE82] W.L. Nabert, *Indirect Interprocess Communication: Virtual Calls as a Message-Passing Discipline for Message-Switched Operating Systems*, CCNG Report T-107, University of Waterloo, June 1982.

[PARN72] D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM,* vol.15 #12 pp. 1053-1058, December 1972.

[PLUM82] David Plummer, *An Ethernet Address Resolution Protocol,* RFC 826, USC/Information Sciences Institute, November 1982.

[POST80] Jon Postel, ed., *User Datagram Protocol,* RFC 768, USC/Information Sciences Institute, August 1980.

[POST81] Jon Postel, ed., *Internet Protocol - DARPA Internet Program Protocol Specification,* RFC 791, USC/Information Sciences Institute, September 1981.

[RAVI85] K. Ravindran, S. Chanson, *State inconsistency issues in local area network-based distributed kernels*, TR 85-7 Department of Computer Science, The University of British Columbia, Vancouver B.C., August 1985.

[RITC74] D. Ritchie, K. Thompson, "The UNIX Time-Sharing System", *Comunications of the ACM*, vol.17 #7 pp. 365-367, July 1974.

[STALL84] W. Stallings, "Local Networks", *ACM Computing Surveys*, vol.16 #1 pp. 3-41, March 1984.

[TANE81] Andrew Tanenbaum, *Computer Networks*, Prentice-Hall, Englewood Cliffs, N.J., U.S.A., 1981.

[TOKU83a] Hideyuki Tokuda, Sanjay Radia, Eric Manning, "Shoshin OS: a Message-based Operating System for a Distributed Software Testbed", *Proceedings of the Sixteenth Annual Conference on System Sciences, 1983,* pp. 329-338, 1983.

[TOKU83b] Hideyuki Tokuda, Eric Manning, "An Interprocess Communications Model for a Distributed Software Testbed", *Proceedings of SIGCOMM '83 Symposium on Communications Architectures and Protocols,* pp. 205-212, March 1983.

[TOKU84] Hideyuki Tokuda, *Shoshin: A Distributed Software Testbed,* Ph.D. Thesis, University of Waterloo, 1984.

[WOOD82] B.J. Wood, D.R. Thompson, L.D. Rogers, D.M Bryant, M.A. Malcom, "A Local-Area Network Architecture Based on Message-Passing Operating System Concepts", *Proceedings of the 7th Conference on Local Computer Networks,* pp. 59-69, October 1982.

[3COM] *Model 3C400 MULTIBUS Ethernet (ME) Controller Reference Manual,* May 1982.

Appendix A

Object Manipulation Routines

This appendix provides a brief description and introduction to the various routines available for the manipulation of objects (files). As a general rule these routines return a positive value when the operation they are asked to perform is accomplished. If the operation fails then a negative value indicating what went wrong is returned.

A number of routines return or require a pointer to the mysterious type *Object*. This is really a pointer to an object descriptor kept in the user's data area. The user does not have to know anything about this and should not try to access the information that is there. For the sake of compatibility the user should declare things to be pointers to the *Object* if required, however, a pointer to anything is sufficient. To get the the definition of the *Object type* and the meanings of the various codes used by the Team Shoshin I/O subsystem the file "io.h" should be included in programs.

Object ***OpenObject**(object_name, mode)
char *object_name;
unsigned int mode;

This routine opens the object specified by *object_name* with the access mode given by *mode*. Currently it is possible to open an object with access for reading, writing, both reading and writing or with a request to create the object if it does not exist. If the object is successfully opened then a pointer to the the object descriptor is returned. A NULL value is returned if the object cannot be opened.

CloseObject(object)
Object *object;

Close the object specified by *object*. A status is returned indicating how the close proceeded. A negative value indicates that the object could not be closed and serves as an indication of the reason for the error.

RemoveObject(object_name)
char *object_name;

This function removes the instance of the object identified by the string pointed to by *object_name*. A positive value is returned if the object was removed. If the object could not be removed then a negative value indicating why it could not be removed is returned.

ReadObject(object, addr, size)
Object *object;
char *addr;
int size;

The purpose of this function is to read a maximum of *size* bytes of data from the object identified by *object* and place them at location *addr*. This function returns the number of data bytes that were actually read. An error situation or, for lack of a better term, "end of file" will result in a negative error code being returned. The exact characteristics of an object will determine just how many bytes are read. For example when reading a "file" *size* bytes will be read whereas if the object is a terminal just the number of bytes currently ready will be read.

WriteObject(object, addr, size)
Object *object;
char *addr;
int size;

This function attempts to write *size* bytes of data from the location specified by *addr* to the object identified by *object*. The number of bytes actually written is returned by the function with a negative value indicating the type of error. If less then *size* bytes are written then it should also be regarded as an error.

```
seek(object, where)
Object *object;
int where;
```

Logically this operation can only be supported on objects like "files" which support the concept of moving around in a file. The purpose of this function is to move the object's read/write pointer to *where* bytes from the beginning of the object. If the operation succeeds then it returns the value of *where* otherwise it returns a negative error code.

```
tell(object)
Object *object;
```

This function is the reverse of *seek* and returns the current location, in bytes from the beginning of the specified *object*. Once again a negative value is an error code indicating what type of error was detected.

```
osize(object)
Object *object;
```

This function returns the current size, in bytes, of the specified *object*. A negative value indicates the type of error that was detected when trying to obtain this information. This function is meaningless for "terminal" type objects.

```
uosize(object_name)
char *object_name;
```

The purpose of this function is to obtain the size of an unopened object as identified by the string pointed to by *object_name*. In typical Shoshin fashion a negative value identifies why the desired information could not be obtained about the object. If the the operation succeeds then the size of the object in bytes is returned.

Appendix B

State Transition Matrixes

The following state transition matrixes describe the remote IPC protocol used in Team Shoshin. The matrix is indexed by an event and the current state of a message. These indexes identify a state/action pair which indicate what action to take and what state to advance to.

The matrixes for driving the sending and receiving protocols share a number of common events:

START: This is the initial event that triggers the start of the protocol machine.

MSG_SENT: The packet that was requested to be sent has been transmitted.

TIMEOUT: The outstanding communications, while waiting for the next significant event, has timed out.

REC_REJ: The remote site has rejected the most recently sent packet.

INV_DEAD: The process on this host that initiated the protocol exchange has died.

The only state common to the sending and receiving sites is the *NUT* which is the initial state that the protocol starts out in. Additionally, there are a number of state/action pairs common to both sending and receiving protocols that are represented by a single term. These single terms usually indicate an error condition and describe the actions needed to terminate the communication. The common or abbreviated state/action pairs are:

ERR: Some sort of protocol related error has been detected. The opposite side is notified and the local process is unblocked with some sort of error indication.

REJ_S,REJ_R: The sender or receiver site has rejected the packet. This is usually a protocol error and the local process is unblocked with an indication of the appropriate error.

FRE,NT_FREE: The process on the local machine has died. The action to take is to notify the remote site of its death and reclaim any resources.

IGNOR: Ignore the packet and stay in the same state.

BLKED: Put the communications on the blocked queue.

B.1 The Sender's Protocol

The protocol state matrix used by the the sender site is given in [Fig. B.1]. When a **bsend**, the sending portion of a **request**, or a **reply** have been issued it is this matrix which drives the protocol.

For the sending site the unique protocol states are as follows:

S_SRTS: Sender, Send a Request To Send.

S_WCTS: Sender, Wait for a Clear To Send.

S_SDD: Sender, send the next data packet.

S_SRTSR: Sender, Send a Request to Send Repeat.

S_WACK: Sender, wait for an ACK.

The unique events are:

REC_CTS: Receive a Clear to Send.

REC_ALIVE: Receive a packet indicating the process of interest is still alive.

ACK: Receive an ACK for the last data packet sent.

The possible actions are:

X_RTS: Transmit a Request to Send.

X_RTSR: Transmit the Repeat of a request to send.

R_CTS: Received a Clear to Send.

SET_ALIVE: Record that the remote process is alive.

U_CHKD: Update data transfer information and see if there is more data to send.

bsend, request, reply (sender)

<----- STATES ----->

EVENTS ->	NUT	S_SRTS	S_WCTS	S_SDD	S_SRTSR	S_WACK
START	S_SRTS,X_RTS	ERR	ERR	ERR	ERR	ERR
MSG_SENT	ERR	S_WCTS,BLKED	ERR	S_WACK,BLKED	S_WCTS,BLKED	ERR
TIMEOUT	ERR	ERR	S_SRTSR,X_RTSR	ERR	ERR	S_SDD,U_CHKD
REC_REJ	REJ_S	REJ_S	REJ_S	REJ_S	REJ_S	REJ_S
INV_DEAD	NT_FREE	NT_FREE	NT_FREE	NT_FREE	NT_FREE	NT_FREE
REC_CTS	ERR	S_SDD,R_CTS	S_SDD,R_CTS	S_SDD,IGNOR	S_SDD,R_CTS	S_WACK,BLKED
REC_ALIVE	ERR	S_SRTS,SET_ALIVE	S_WCTS,BLKED	ERR	ERR	S_WACK,BLKED
ACK	ERR	ERR	ERR	S_SDD,R_ACK	ERR	S_SDD,R_ACK

Figure B.1: Sender's Protocol State Matrix

B.2 The Receiver's Protocol

The protocol state matrix used by the the receiving sites are given in [Fig. B.2]. When a **brec**, a **brecany**, or the receiving portion of a **request** have been issued it is this matrix which drives the protocol.

The unique events for the receiver's protocol are:

REPLY_START: The sending portion of the ***request*** primitive has been executed and the process is now waiting for the reply.

REC_RTS: A request to send packet has arrived.

REC_DD: A data packet has arrived.

REC_LD: The last data packet has arrived.

The uniques states for the receivers are:

R_SAYT: Receiver, send an Are You There packet.

R_WRTS: Receiver, waiting for the remote machine to issue a request to send.

R_SCTS: Receiver, send a Clear To Send.

R_WDD: Receiver, wait for a data packet.

R_SAYT2: Receiver, repeat the sending of an Are You There packet.

R_DONE: Receiver is done, the communications is finished.

The unique actions for the receiving protocol are:

X_AYT: Transmit an Are You There packet.

R_RTS: Received a Request To Send.

XCTS: Transmit a Clear to Send.

C_SUC: Communications was a success, take appropriate termination action.

C_D_O: A clear to send was sent, check to see if there is any data to receive.

U_BLKED: Received a data packet, update the receiver information and transmit an ACK.

brec, brecany, request (receiver)

EVENTS | V	<----- STATES ----->					
	NUT	R_SAYT	R_WRTS	R_SCTS	R_WDD	R_SAYT2
START	R_SAYT,X_AYT	ERR	ERR	ERR	ERR	ERR
MSG_SENT	ERR	R_WRTS,BLKED	ERR	R_WDD,C_D_O	ERR	R_WDD,BLKED
TIMEOUT	ERR	ERR	R_SAYT,X_AYT	ERR	R_SAYT2,X_AYT	ERR
REC_REJ	ERR	REJ_R	REJ_R	REJ_R	REJ_R	REJ_R
INV_DEAD	FRE	FRE	FRE	FRE	FRE	FRE
REPLY_START	R_WRTS,BLKED	ERR	ERR	ERR	ERR	ERR
REC_RTS	ERR	R_SCTS,R_RTS	R_SCTS,R_RTS	R_SCTS,IGNOR	R_SCTS,XCTS	R_SCTS,XCTS
REC_DD	ERR	ERR	ERR	R_WDD,U_BLKED	R_WDD,UBLKED	R_WDD,U_BLKED
REC_LD	ERR	ERR	R_DONE,C_SUC	R_DONE,C_SUC	R_DONE,C_SUC	R_DONE,C_SUC

Figure B.2: Receiver's Protocol State Matrix