INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International

300 N. Zeeb Road
Ann Arbor, MI 48106

8226591

Rao, Ram

A KERNEL FOR DISTRIBUTED AND SHARED MEMORY COMMUNICATION

University of Washington PH.D. 1982

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

A KERNEL FOR DISTRIBUTED AND SHARED MEMORY COMMUNICATION

by

Ram Rao

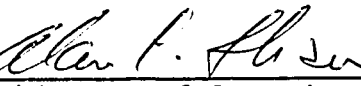
A thesis submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Washington

1982

Approved By 

(Chairman of Supervisory Committee)

Program Authorized
to Offer Degree Department of Computer Science

Date 3 June 1982

Doctoral Dissertation

In presenting this dissertation in partial fulfillment of the requirements for the Doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U. S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to University Microfilms, 300 North Zeeb Road, Ann Arbor, Michigan 48106, to whom the author has granted "the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform."

Signature Ram Rao

Date June 3, 1982

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

INDEX TO TERMINOLOGY

ACKNOWLEDGEMENTS

I am extremely grateful to Professor Alan Shaw for his encouragement, patience, expert guidance and friendship during the course of this work. I also wish to thank Professor Jean-Loup Baer for his constructive reading of this dissertation and for the loan of a computer terminal. I am indebted to Professor Edward Lazowska for many helpful discussions on performance. Financial support for this research was provided by the National Science Foundation under grant MCS-7826285.

Special thanks are due to my wife, Asha, for participating in the struggles and victories of my student life, for faithfully supporting me in prayer and for her labors in proofreading.

My soul exalts the Lord,
And my spirit has rejoiced in God my Savior.
For He has had regard for the humble state of His bondslave.

Luke 1:46-48

This dissertation is dedicated to my father, Mirle, and to the memory of my mother, Lakshmi. Their love, sacrifices, and keen interest in all aspects of my growth have formed the foundations on which my graduate studies have been built.

CHAPTER 1. INTRODUCTION

The first efforts in programming were limited to a single process executing on a uniprocessor. With the appearance of multiprocess programs in the early 1960's, the possibility of interaction between processes arose. Memory shared between processes was used as the medium for interaction. More recently, interaction between processes through another paradigm has been considered: passing messages between them. For each paradigm, communication primitives have been sought to help the programmer construct properly functioning multiprocess programs. Today, a wide variety of predefined primitives may be found in programming languages that support process interaction, with little agreement on what appropriate primitives are.

In this dissertation, it is argued that each paradigm is especially suited to a specific physical architecture. In particular, there are applications where both need to be used side-by-side. Hence, primitives for both should be available for programming. The DASH (Distributed And SHared) kernel was developed in response to these needs. It provides flexible and powerful tools for interprocess communication using message passing and shared memory separately, as well as the two together. Also, the programming task is simplified because programmers may use the kernel to construct higher-level mechanisms well-suited for specific applications.

1.1 Communication Architecture Definitions

In order to discuss distributed and shared memory systems, we

need to understand what the terms "distributed" and "shared memory" mean. These terms are difficult to define precisely. The brief definitions that follow are intended to give an intuitive feel for these terms. (The definitions appearing in Appendix A attempt to provide greater intuition.)

Memory is said to be shared between two (or more) processes if the memory is part of each process's address space and if each process can access it with the same ease and cost as it can its own local memory. Programming languages which reflect a shared view of communication include Modula [Wirth 77] and Mesa [Mesa 79].

Distributed communication between two processes involves the transfer of information between the processes through means other than shared memory. In comparison to memory access, this transfer usually involves greater delay and occurs over smaller bandwidth channels. Also, active cooperation between the two processes is normally required to complete the transfer. Distributed communication is often termed "message passing", since messages are a convenient means of transferring information. Abstractions providing a distributed view of communication may be found in CSP [Hoare 78] and PLITS [Feldman 79].

A system of processes, some of whose interactions are distributed and some shared, is termed a hybrid system. Hybrid communication and its applications form the subject of the next section.

1.2 Hybrid Communication

Interaction between processes residing on remote processors (i.e., processors that do not share memory) is, of necessity, distributed. This normally requires both participants to play an

active role in the information interchange. It is sometimes beneficial for a process to relegate its remote communication responsibilities to another communication process, so it can give more of its attention to its computational task. If the communication process, sometimes called an agent, interacts with its local client through shared memory, some benefits result: it may be possible to avoid copying of data, by using shared memory; and queries from remote processes about the client's state can be answered by the communication process on behalf of the client, if the relevant state information is in the shared memory. As far as the client is concerned, its remote interactions are asynchronous. The communication process is involved in hybrid communication: shared memory interactions with the clients and distributed interactions with remote processes.

Asynchronous remote interactions, as described above, are well-suited for hybrid communication. Such interactions arise in a number of situations, examples of which follow. A file server process could pass the address of a disk block (in shared memory) to a communication process, to be transmitted to a remote client. Similarly, asynchronous communication primitives in a programming language can be implemented using a buffer process to receive the address of the variable to be sent to a remote destination; this frees the sender from having to interact with the destination directly. Also, in implementing distributed mutual exclusion, it is necessary for a process to query the synchronizing status of other processes [Lamport 74, Ricart 81]. This is conveniently achieved by requiring the communication process to receive queries from remote clients, and to answer them based on the synchronizing state in shared memory. (The algorithm published in [Ricart 81] uses a hybrid approach.)

The subject of hybrid communication, and especially the programmer's view of it, is explored further in Chapter 5.

1.3 Summary of Contributions

The main contributions of this work are as follows:

- A unified approach to distributed, shared and hybrid communication is developed. The DASH kernel provides a set of simple tools that bring together the common features in communication. While permitting a programmer to take advantage of the benefits offered by a particular communication architecture, it provides flexibility both in dealing with a variety of process interactions and for constructing higher-level mechanisms suited to specific applications.

- The usefulness of hybrid programming is demonstrated. Though hybrid architectures arise naturally, so far, they have not been visible to the programmer. The utility of hybrid programming is demonstrated through a series of examples programmed in DASH.

- The DASH kernel developed in this dissertation is practical. The kernel has been implemented on a uniprocessor, and has been used to program a number of distributed, shared and hybrid problems. A strategy for implementing DASH on a distributed system is outlined.

Other contributions are: a study of distributed communication focussing on types of interaction and the issues relevant to the design of communication mechanisms; and an estimate of performance degradation due to transmission delay in distributed programs as compared with equivalent shared memory programs.

1.4 Organization of Dissertation

The next chapter contains a study of distributed communication. It investigates distributed interactions and the design of mechanisms for distributed programming. This leads into

Chapter 3, where the DASH kernel is introduced and its features for distributed communication are illustrated in a variety of applications. Shared memory and DASH features supporting it are discussed in Chapter 4. The techniques developed in Chapters 3 and 4 are brought together in Chapter 5, to facilitate hybrid programming. Approaches to hybrid programming are examined and the kernel is used in solving a few hybrid problems. Experience with a uniprocessor implementation of the kernel is described in Chapter 6. Also, issues pertaining to a network implementation are examined. Chapter 7 contains a study of the performance degradation of distributed programs caused by transmission delay. A simple problem is simulated and its results are used to develop some heuristics. In the final chapter, this work is evaluated and some interesting remaining problems are identified. Two appendices follow: one containing architectural definitions and the other a specification of the kernel in CSP.

CHAPTER 2. PROGRAMMING DISTRIBUTED COMMUNICATION

Concurrent programming has been studied for over a decade now, and much progress has been made in the issues pertaining to the use of shared memory. With the emergence of hardware making distributed programming feasible, the attention of researchers has turned in this direction only in the recent past. Because of the newness of the field, a detailed survey and evaluation of methods, problems and issues pertaining to communication in distributed systems seems appropriate. The discussion in this chapter is limited to communication aspects, particularly the design and application of programming language primitives. A perspective on other aspects of distributed computing (e.g., reliability, process creation/deletion, termination) may be found in [Mohan 80].

The contents of this chapter are drawn from [Rao 80]. The first section examines programming languages with features for distributed communication. Then in Section 2.2, the interactions between processes in a number of applications are studied. Finally, in Section 2.3 issues involved in the design of primitives are identified and their impact on programming applications is determined.

2.1 Extant Language Proposals

Several languages with communication mechanisms have been designed and documented in the literature. These languages are similar in that they all encourage communication by copying between address spaces and provide limited (if any) facilities to access globally shared variables. They differ in the approach they take

towards issues such as structured view of communication, control over message reception and synchronization semantics. In this section we examine seven of these languages, limiting our discussion to their communication features.

In the PLITS system [Feldman 79], modules (which are essentially processes) communicate through buffered messages. Messages consist of name-value pairs. Each module knows certain names and can access only the name-value pairs (in messages) with which it is familiar. Primitives are provided for the sending and receiving of messages. The 'send' primitive is invoked with a destination module name and an optional message type. The process executing the send does not normally block--the message is placed on a queue determined by the destination and message type (blocking occurs if the queue is full). The receive primitive may optionally state a source and message type, thereby limiting the set of messages it is willing to accept. It blocks the receiver until an appropriate message arrives or a timeout occurs. A 'pending' primitive is also provided to test for the availability of messages from a specific source and of a specific type. This primitive can be used to simulate more complicated control over receiving. (For example, a module can decide to wait for a message about one of two specific message types by executing a loop in which it uses the pending primitive to successively check for the arrival of each message type.) Modules may be created dynamically and module names may be passed in messages. Thus it is possible for a sending module to include in a message, the name of the module to which a response message is to be sent. Since the number of slots in a message is not known a priori, storage has to be allocated dynamically within the receive primitive. The ability to send variable length messages can simplify the user's programming task.

Communicating Sequential Processes (CSP) [Hoare 78] chooses to communicate via unbuffered messages. Messages consist of simple or structured types which must be known beforehand to both communicating partners. The send and the receive are fully synchronized, i.e., both sender and receiver block until the message transfer is completed. Sending takes place with the execution of an output command specifying the destination process as well as the type of information to be sent. The receiver executes an input command in which he specifies the source and the type of information desired. Nondeterministic receiving of a set of message types is accomplished by imbedding the input commands for each possible type in 'guards'[1]. Guards consist of boolean expressions and input statements[2] and are followed by a statement list. A guard is ready if the boolean expression is true and if the matching output statement has been executed. A process may list several alternative guards and choose to nondeterministically wait on them. This means that if more than one guard is ready, a nondeterministic selection between the ready guards is made. If none are ready, the process waits until one is ready and it is selected. Once a guard is selected, its statement

---

[1]The concept of nondeterministic guarded commands first appeared in [Dijkstra 75]. The meaning of nondeterminism as it appears here and throughout this document, differs from that used in automata theory. Here nondeterminism means that if a choice is to be made between several alternatives in the implementation of an operation, the selection made by the implementation is arbitrary, at least in the user's perception.

[2]In extensions to CSP, the inclusion of output commands in guards has been proposed [Hoare 78, Silberschatz 79, Bernstein 80]. This provides symmetry and would simplify some programs (e.g., the bounded buffer).

list is executed. The implementation of guards containing input commands requires the ability to ascertain if any messages of a particular type are present, without committing oneself to accepting the message. CSP does not allow dynamic process creation or the passing of process names as parameters in output commands. Another problem with CSP arises in the programming of server processes--the 'receive' has to specify the names of source processes. To get around these naming problems, subscripted processes are introduced and a range of subscripts is permitted in input commands. This is an ad hoc solution requiring all possible source processes to be known at compile time[3].

Distributed Processes (DP) [Brinch Hansen 78] is yet another proposal for programming in a distributed environment. DP uses the procedure call model for communication, by supporting the abstraction of a remote procedure call. When a process A calls a remote procedure P (in a process B), A is blocked until the call completes. Process B has a number of representative subprocesses, one of which fields A's request and attempts to execute the procedure P on behalf of A. The remote procedures resemble monitor procedures in that only one of them (or the body of the process) may be active at a time. DP also provides a guarded command based mechanism (guarded region) for processes to block until a certain condition arises. By use of this mechanism selective receiving can be implemented. On completion of the procedure P, process A is passed return arguments and is allowed

---

[3]The introduction of named I/O ports [Silberschatz 81] has been suggested as a solution to the explicit naming problem in CSP. Server processes input from ports, without a priori knowledge of the clients that may output to that port.

to resume. System buffering is left up to the implementor. Like CSP, DP is not geared for dynamic process creation. Also, at compile time, the calling process has to know the names of remote procedures it wishes to invoke and the processes they reside in.

Combining some of the ideas presented in CSP and DP, is another proposal--Synchronizing Resources (SR) [Andrews 81]. Processes are static and communicate via remote calls as in DP. There are some differences from the receiver's point of view, however. The receiving process executes an input statement which may contain several input commands in guards (as in CSP). An arithmetic expression may also be provided to determine the order in which requests are serviced. The boolean expressions in the guards and the arithmetic expressions can both contain parameters of the call, thereby providing selective receiving not only by type but also by content. (This forces the implementation to buffer messages at the receiving end.) On termination of the input statement, return parameters are passed to the caller, who is then resumed. An additional command 'send' is provided which functions similar to 'call' except that the caller is resumed the moment the message is transferred or delivered (a decision left to the implementor). Return parameters intended for the caller are ignored. Since no resource or process names are specified in the communication primitives, entry names must be unique system-wide.

In extensions made to CLU (ECLU) [Liskov 79], some primitives for distributed computing are proposed. A buffered message passing

approach is used[4] with no blocking of the sender. Messages consist of a command identifier and zero or more arguments. The send command takes a message and a port name and returns control once the message is dispatched. Messages are buffered at ports, so that the receiver can pick them up at his convenience. The receiver can specify a list of ports on which he is willing to receive. He may also state the commands he is prepared to receive and the action to be taken on the receipt of each command. He can also specify the action to be taken if the receive times out or fails for some reason. This is a convenient form of selective receive since nondeterminism is built into the receive primitive. Port names and command identifiers have to be known prior to any communication. Process and ports are dynamic. Port names may be passed as parameters in messages, permitting the sender to inform the receiver as to where any response should be sent.

The programming language Ada [Ada 79a] includes some constructs which may be used for distributed communication. The procedure call model is once again advocated and the primitives provided closely resemble those in SR. The calling process invokes a remote 'entry' and then blocks until the call completes. The called process issues an accept statement with an entry name. When an entry call is available, the "rendezvous" is completed and the statement body of the accept is executed. On completion of the accept statement, the calling and called processes are both free to

---

[4]The approach to communication in ECLU has been revised: the procedure call model has been adopted [Liskov 80, Section 7.4]. Also, considerable emphasis is being placed on designing primitives which are robust against process failure.

continue. Selective receiving is achieved by allowing accept statements to appear within guards of a guarded command. No scheduling of entries (as in SR) is permitted. The designers of Ada rejected the idea of scheduling entry invocations based on parameters of the call because of implementation difficulties [Ada 79b].

The distributed communication mechanisms in Communication Port (CP) [Mao 80] are based on the procedure call model and resemble those in SR and ADA. The main difference is the disconnect feature which allows the called process to specify when the caller should be resumed. Resuming the caller at the start of the called entry makes the communication resemble CSP, while resumption at the end makes it similar to SR and ADA. Schedulers are convenient to program--the caller is resumed only when the resource is available. Ports may be included in guards and can list processes that may access them. Thus, selective receiving by type and source is possible.

2.2 Interaction Classes

To help understand communication needs in distributed computing, it is useful to examine applications and define classes of interactions based on communication characteristics. Once this has been done, it is often sufficient while discussing communication to study a specific problem containing an interaction belonging to a class and to consider it representative of the class as a whole. The examples in the papers introducing the language proposals form the basis for the classification scheme suggested here.

Basic to all interactions is the simple logical transfer of information between processes. This is a rather informal notion, often found in the English language descriptions of programs (such as in the examples that appear later in this section). Statements such

as "process A sends process B the value x" are deliberately vague about the precise semantics of the information transfer (e.g., buffering and synchronization semantics). All interactions between processes may be described in terms of such information transfers. However, certain patterns of interaction are observed to occur often, and these suggest some structured abstractions.

2.2.1 Pipelines

One common interaction pattern is that found in a pipeline: each member process, called a pipeline process, receives input through a simple information transfer from a source process, computes on the input, and produces some output which it transfers to a sink process. The type of computation may vary: several input items may be required to produce an output item or one input item could result in several output items being produced. The important criterion is that there be exactly one source and one sink, and that the source, sink and pipeline processes be distinct. Examples of problems with pipeline interactions are listed below. Our definition can be extended to two dimensional pipeline interaction: the pipeline process has two distinct sources and two distinct sink processes. Such interaction is exhibited in the iterative array matrix multiplication problem.

Examples

1. Squash Asterisks [Hoare 78]: A stream of characters is input, every pair of consecutive asterisks "**" read is replaced by an upward arrow "^", and the characters are written on an output stream. The final character is assumed to be other than an asterisk.

2. Sorting Array [Brinch Hansen 78]: A sorting array is an array of processes which is used to sort a set of numbers.

In equilibrium each process contains one number. The processes each execute the following algorithm. Upon initialization, a number is input from the predecessor process and stored locally. Numbers are then successively input from the predecessor. If a number is greater than or equal to the local number it is passed on to the successor process, otherwise the local number is passed to the successor and the input number is stored locally. When the set of numbers has been input, the successive processes will contain the numbers sorted in nondecreasing order. Ignore the practical problem of reading out the sorted array.

3. Eratosthenes' Sieve [Hoare 78]: An array of processes is used to compute in ascending order all primes less than 10000. Each process inputs a prime from its predecessor and stores it. The process then inputs an ascending stream of numbers from its predecessor and passes them on to its successor, suppressing any that are multiples of the original prime.

4. Iterative Array Matrix Multiplication [Hoare 78]: A square matrix A of order 3 is given. Three streams are to be input, each representing a column of an array IN. Three streams are to be output, each representing a column of the product matrix IN x A. After an initial delay, the results are to be produced at the same rate as the input is consumed. The solution should take the form shown in Figure 2-1. Each of the nine nonborder nodes inputs a vector component from the west and a partial sum from the north. Each node outputs the vector component to its east, and an updated partial sum to the south. The input consists of row vectors of IN (with elements x,y,z) and is produced by the west border nodes. The desired results are consumed by south border nodes. The north border is a constant source of zeros and the east border is just a sink. No provision need be made for termination nor for changing the values of the array A.

2.2.2 Servers

Another interaction pattern frequently observed, is between a server and its clients. A server accepts information from a client, performs the requested service and sometimes transfers a response back to the requesting client. In general, servers handle several types of requests. The important characteristics of servers are a)

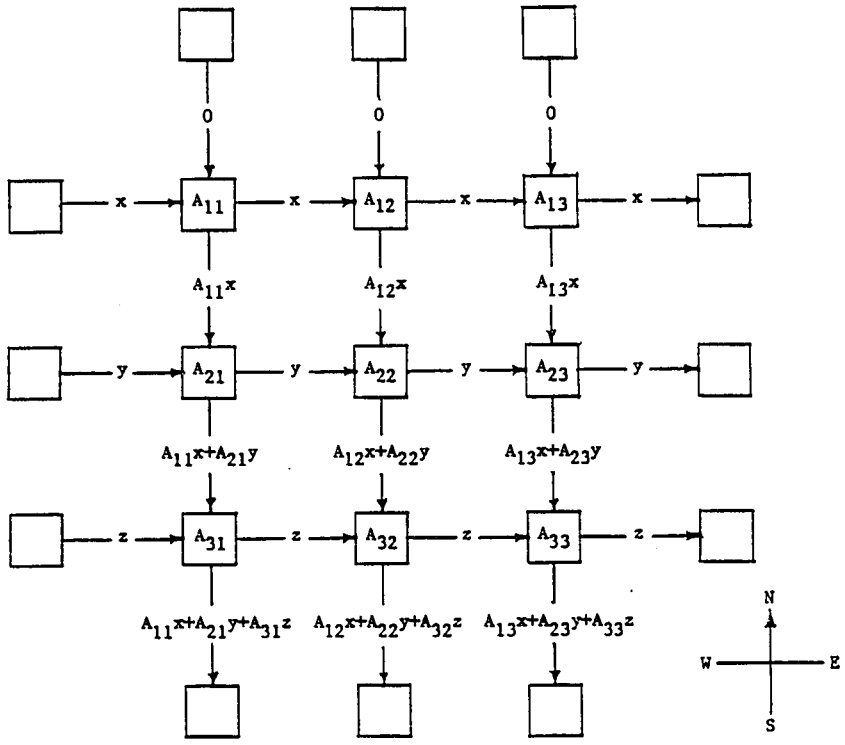


Figure 2-1: Iterative Array Matrix Multiplication

at least one type of request merits a response and b) they more often than not cater to more than one client. The response mentioned in a) may be explicit or implicit: the response information transfer may contain some data, or may simply be an indication of service completion (as in the response to the P request in the semaphore example below).

Some of the considerations involved in programming servers are: being attentive to many clients; perhaps accepting certain requests only when the server is in a specific state; sometimes deferring a partially processed request until other requests are handled.

Examples

1. Factorial Routine [Hoare 78]: The routine inputs an integer, computes its factorial and returns it to the client. (A simple server, requiring no synchronization between clients.)

2. Semaphore [Dijkstra 68]: This server implements a general semaphore process which provides indivisible P and V operations. A client invoking a P must wait until the semaphore is greater than zero and then decrement the semaphore. The V operation increments the semaphore by one. (The semaphore process may decide to accept P requests only when the semaphore is positive.)

3. Bounded Buffer [Habermann 72]: Here we have a buffer containing a finite number of slots used to store information in transit between producers and consumers. Each producer may make requests to the buffer server, passing it a portion of information to be buffered. The buffer server accepts the request only if space is available, in which case it stores the information. Each consumer process requests information from the buffer, and awaits this information. If there is information available in the buffer, the buffer process dispatches a portion to the consumer; otherwise it delays servicing the consumer request.

4. Priority Scheduler [Brinch Hansen 78]: The scheduler allocates a resource among requesting clients in priority order. Clients make requests on the server, furnishing their priority. When the resource is granted, the client uses the resource and finally releases it by informing the server. When a resource is freed by a client, the server grants it to the waiting client with the highest priority. (The server may have to queue the request and defer processing it until it has the highest priority.)

5. Alarm Clock [Brinch Hansen 78]: An alarm clock server allows clients to sleep for different time intervals, giving a wake-up call at the end of the interval. The server receives a signal from a timer process after each time unit.

6. Readers and Writers [Courtois 71]: Two groups of processes, readers and writers, share a database. To protect the integrity of the database, at most one writer at a time may access it, and no reader may examine the database while a writer is altering it; readers, however,

may access the database concurrently. (Some variations of this basic problem with differing priority and fairness requirements appear in [Courtois 71, Hoare 74].) A server controls access to the database.

2.2.3 Complex Interactions

Complex interaction patterns between a large number of processes can often be viewed as consisting of parts which behave as pipelines or servers. An information transfer may be required at times to "glue" the parts together. Such an approach is demonstrated in the example that follows.

Example

- Two Step Commit Protocol [Lampson 81, Baer 81]: This is an algorithm to ensure transaction update atomicity[5] in a distributed data base system. The transaction coordinating site (termed the master), having sent updates to participating sites (termed slaves) , asks each slave if the update can be performed. If all slaves respond positively, the master decides to commit (proceed with the transaction). Otherwise, (because one or more slaves has lost data due to a crash) it decides to abort the transaction. In either case, it sends messages informing all the slaves of its decision. Through appropriate use of stable (i.e., crash-resistant) storage, this algorithm ensures consistency of the distributed database, in the presence of crashes (master or slave). Recovery procedures are prescribed, to be used after crashes. In this example, we focus on the master's decision making, based on the interaction with slaves. The crash recovery aspects of the protocol are ignored.

In the example above each slave process may be considered as a server interacting with a single client: the master. The master sends a commit query request to each slave server and awaits a

---

[5]The concept of atomicity in the presence of crashes is discussed in Section 6.2.2

response. Based on the response, it either decides to commit or abort the transaction. Finally, the master sends a request to each slave informing it of its decision.

2.3 Design Issues in Primitives for Distributed Problems

While designing a set of communication mechanisms for distributed computing, several decisions need to be made (e.g., whether to have system buffering, and the type of synchronization to be implemented). This section examines some of these decisions in detail: it focuses on the design decisions and presents and evaluates alternatives. Next, the mechanisms found in the languages discussed in Section 2.1 are examined with respect to how they address the issues presented here and their ease of use.

2.3.1 Issues

Communication model

One of the major decisions that has to be made in the design of a set of communication primitives is the abstraction of the communication to be presented to the user. Two major ways of viewing the communication are a message transfer approach and a procedure call approach.

The message transfer model views communication between processes as information flowing between them on channels. A process wanting to transfer information to another process places this information on a channel to the destination process. The information is transferred in entities termed messages. When the receiving process needs information, it picks the message off this channel. The languages CSP and PLITS adhere to the message transfer model. In

the procedure call model (e.g., DP, ADA, SR) each process treats its communication partner as a set of remote procedures and issues procedure calls with some parameters. The 'called' process receives the request, services it and returns information back to the caller.

The message transfer model is appropriate when there is unidirectional flow of information between processes as in pipelines. For, example in UNIX [Ritchie 74] processes can be connected by pipes, and pipeline processes can be written to process the information on the input pipe and produce information for the output pipe. The message transfer model may result in greater (potential) parallelism than the procedure call model because once the message is transferred the sending process is free to continue execution, whereas a procedure caller normally waits for the called procedure to complete. (Languages such as Mesa [Lampson 80] get around this by allowing a procedure call to be forked as a separate process.) If the communication warrants a reply message, then the sender may choose to look for it at his convenience. The message transfer model can be used to simulate the procedure call model by establishing a convention that after sending a message, the sender awaits a reply (at the expense of forcing the user to adhere to the convention).

The procedure call model is convenient when processes interact in a server-client fashion. A common example is of a process providing access to a (possibly shared) resource. Processes wishing to use the resource issue appropriate remote calls, and control is returned when the called process has serviced the request. When processes communicate in this manner, procedure call is convenient from the user's point of view and can also be implemented efficiently in terms of the exchange of control information between the communicating processes.

Synchronization

Let us first consider the case where the sender is blocked on initiating communication. There are two choices: we may block until either the message is received at the destination (synchronized send) or until the destination process receives the message, processes it and sends back a reply message (send-reply). In the synchronized send case (e.g., CSP), the sender is unblocked as soon as the receiver has input the message, so the sender has assurance of message delivery. Buffering of messages is not necessary. In the absence of buffering, one must ensure that the receiver is ready when the sender transmits the message. In this scheme, there is some loss of parallelism, because the sender blocks if the receiver is not ready to accept message delivery. Send-reply primitives (e.g., DP, ADA, SR, CP) closely resemble procedure invocation. Since the sender is blocked for a longer time, there is an even greater loss of parallelism. However, implementation is even simpler than in the case above, because during the reply the sender is guaranteed to be waiting for it. While convenient for use in problems amenable to the procedure call model, these primitives are less suited to pipelined problems (see Section 2.3.2).

In the no-wait send case (e.g., PLITS), the sender is not blocked on initiating communication and greater parallelism is hence possible. The sender regains control the moment the message is buffered somewhere in the system and may reuse his message buffer. Should the message later prove undeliverable, an asynchronous exception must be raised. If buffer space is exhausted, the sender must be informed. The no-wait send may reduce the danger of deadlock. Situations exist such that if the sender blocked, deadlock

would occur, but would be safe from deadlock if the sender did not block [Kieburtz 79]. The no-wait send primitives are suitable for problems with a message transfer orientation, though the procedure call model can easily be simulated (with some overhead). However, if the sender needs to synchronize with the receiver, a protocol must be adopted for this purpose.

Another less important decision concerns the blocking of the receiving process. Causing the receiver to block until the message is ready for it leads to a simple implementation and easily usable primitives. The alternative is to permit the receiver to regain control if the message requested is not ready; greater parallelism results in that case. Languages such as PLITS and ADA provide means to check message availability.

Selective receiving

By selective receiving, we mean the ability of the user to restrict the types of messages that he is willing to receive to a subset of the universe of message types. With this facility, the user need only be concerned with handling those messages that are appropriate for his current state. Without this facility, the user is forced to explicitly queue those messages he is not ready to act on, perhaps duplicating some of the queueing structures in the communication mechanism.

For certain applications, one may want to select the messages to be received by the source: for example, a process may want to lock itself into communicating with one partner. CSP requires the source to be named in message reception. In the case of a general server which manages a resource, one may need to selectively receive by message type: a process implementing a bounded buffer would only be

able to process 'remove' requests when the buffer is full. This requires that messages have a type field associated with them to allow the receiver to select. If there is no system buffering, the type field has to be sent ahead to the receiver to allow it to choose between types. An even more selective form of receiving found in SR accepts messages only when values of certain fields in the message satisfy a specific constraint. This permits the programming of schedulers (e.g., selecting the request with the shortest service time). While being very flexible, this alternative forces the system to provide buffering at the receiver, in order to be able to select between messages on the basis of their content.

Selective receiving can be provided with varying degrees of convenience to the user. At one extreme is an easily implementable facility with no selective receiving, forcing the user to buffer the messages and select among them himself. At the other end is a facility which allows the user to list the set of messages that he is willing to nondeterministically accept, the conditions under which each message is acceptable and the actions to be taken on receiving each message, i.e., guarded commands with boolean expressions and receive statements in guards. In between the extremes is a facility (e.g., PLITS) where the user can test for the availability of a particular kind of message and choose whether or not to wait for it.

Message length and structure

Since in a distributed system all interprocess communication takes place via messages, a design decision has to be made as to the degree of information communicating processes have about the messages they will be handling. One possibility is to have a predetermined set of message types, each type having a fixed structure and length.

Both parties in communication agree on the types of messages they are dealing with. The receiver can use the message type information to determine the buffer size to be provided on a receive. There are, however, applications where variable length types, such as strings, trees, or file system writes, need to be transmitted. This can be simulated with a set of fixed length message types. We need to introduce a descriptor type which indicates the length (in number of physical messages) and structure of the logical message that follows. ECLU allows the user to furnish routines which "flatten" types into transmissible representations and "unflatten" them on reception. When variable length types are supported, transmission of such types becomes easier for the user. The implementation, however, is more complex because of the need for dynamic memory allocation to support variable length messages.

Naming and global information

The issue here is the kinds of names and global information that a user of a communication mechanism has to know in order to use it. A trade-off between functionality and location exists. One alternative is to let the user be aware of the entity (e.g., process, port) with which he is communicating. He would also need to know the names of the various functions performed at this entity. So in this case, there would be system-wide unique entity names, together with distinct function names within entities. Another approach is to have system-wide unique function names only. Here the user is not aware of how functions are grouped into entities, providing him location transparency (a higher level abstraction) but leaving the implementor a more difficult task. A more complete discussion of naming issues may be found in [Mohan 80].

System Buffering

The term system buffering refers to memory internal to the communication mechanism, which may be used for the temporary storage of communication data. This storage is not directly accessible by the user of the communication facility. The designer of the communication mechanism has to decide whether or not to include system buffering.

The presence of system buffering has two main advantages. First it allows greater parallelism, because the sender of the information need not wait until the receiver takes the information in--he can continue his processing asynchronously with the receiver (up to a limit imposed by the finite buffer space). Also, buffering at the receiving end allows the receiver to select from various message types and even select on the basis of contents of messages.

If there is no system buffering, a simpler implementation is possible because buffer management is no longer a concern. The sender is now forced to block until the receiver has input the message. When control returns to the sender, he is implicitly receiving information about the receiver's state; this synchronizing information may simplify the programming task.

The design issues discussed above are not completely independent of each other. Some of the interdependencies are now described. The choice of communication model can influence the synchronization semantics of the primitives. Synchronization issues have a direct bearing on whether system buffering is required. Selective receiving by content also forces buffering at the receiver. Certain strategies for large messages require dynamic buffer

	PLITS	CSP	ECLU	DP	SR	ADA	CP
model	message transfer	message transfer	message transfer	procedure call	procedure call	procedure call	procedure call
synch	no-wait send	full-sync send	no-wait send	send-reply	all three possible	send-reply	send-reply
selective receiving	by typ(o) by src(o)	by typ(m) by src(m) bool.cond nondeter	by typ(m) nondeter	by typ(m) bool.cond nondeter	by typ(m) by cnt(o) nondeter schd.cond bool.cond	by typ(m) bool.cond nondeter	by typ(m) by src(o) bool.cond nondeter
buffering	yes	no	yes	no	yes	no	no
global info	modules slots	processes types	ports commands	processes procedure	resources entries	tasks entries	processes ports
long messages	special provision	<---------- no special provision ---------->					

abbreviations:

o - optional	m - mandatory
src - source	bool.cond - boolean condition
cnt - content	schd.cond - scheduling condition
typ - type	nondeter - nondeterminism

Table 2-1: Summary of Language Characteristics

allocation at the receiver. In designing a set of primitives to meet certain functional objectives, these interdependencies must be considered.

The approaches taken by the languages presented in Section 2.1 are summarized in Table 2-1.

2.3.2 Applications

Our goal in this section is to observe how the languages presented in Section 2.1 function in different application areas and to attempt to relate this to the design issues. This exercise leads to better understanding of the issues and provides insight into the communication needs of applications. Each of the languages has been used to program problems typifying the two simple interaction types discussed in Section 2.2: pipelines and servers. This effort is described in full, together with the programs written, in [Rao 80]; the results are presented in this section.

Representing pipeline interactions is the sorting array. This is a simple concurrent problem which tests the ability of the communication mechanism to conveniently handle unidirectional information flow. The problem chosen to represent server interaction is the bounded buffer. Two characteristics of the bounded buffer problem that make it interesting are: the buffer server has to respond to two different types of requests (insert and remove), only one of which may be honored when the buffers are either all full or all empty; and, if the buffers are neither all full nor all empty, the server should be able to respond to either type of request immediately.

As expected, the primitives designed for the message transfer model were better suited for programming the sorting array, and the

procedure call model primitives did well in the bounded buffer. Among the message transfer model based primitives, CSP (though it has cryptic notation) was convenient to use in programming the bounded buffer, while ECLU and PLITS were more difficult to use. This difference arises from the fact that in the general server environment, it is convenient to have boolean conditions describing the set of states in which a message may be accepted. Boolean conditions together with nondeterminism facilitate programming. In ECLU we find nondeterminism built into the receive primitive. However, the lack of boolean expressions forces the programmer to do a case statement on the state and then execute separate receives for each case. Since a server system is request driven, it seems more natural for the programmer to think in terms of the set of states that permit the reception of a message rather than the set of messages acceptable in each state. Further, it may be observed that primitives with boolean expressions but without nondeterminism would still be cumbersome. The absence of nondeterminism forces the boolean expressions to be disjoint. Therefore, in situations where the same action is to be taken on receiving a message in a number of disjoint sets of states, the action has to be explicitly specified for each separate set of states. For example, in programming a bounded buffer process, a three-way test must be made to see if the buffer is full, empty or neither full nor empty; insert requests may be serviced in the last two cases and the code for servicing the request must be repeated for each of these two cases.

Among the procedure call model primitives, SR, ADA and CP were found suitable in the sorting array problem, while DP was extremely difficult to use. The difference here could be the fact that control in SR, ADA and CP programs flows sequentially unlike control in DP.

Control in DP transfers between various procedures (as in a monitor [Hoare 74]) and boolean flags have to be set up to coordinate the activity of these procedures.

On writing solutions for the sorting array, PLITS and ECLU were observed to have a greater degree of parallelism than the rest because output to the successor process occurs in parallel with input from the predecessor. Solutions with greater parallelism were then obtained for CSP (by using the parallel command to perform the input and output concurrently), DP, SR, ADA, and CP[6] (by introducing an additional process). Greater parallelism in the PLITS and ECLU solutions may be attributed to no-wait send semantics.

Selective receiving by message type is of importance in the bounded buffer problem. As has been discussed earlier in this section, the two ingredients that lead to easy selective receiving are controlling boolean expressions and nondeterminism. It is interesting to note that DP, SR, ECLU, and ADA have no facility for selective receiving by source, whereas CSP insists that the source be specified on each receive. Both extremes present problems. Programming of server processes in CSP is problematical because the identity of clients has to be known beforehand. In problems such as the sorting array, there is no check in DP, SR, ECLU, ADA against a spurious message from a random process arriving and throwing one process out of synchronization. CP allows optional selection by source, resulting in considerable flexibility.

In summary, we list some features of a communication system

---

[6] A proposal to permit nondeterministic choice between connect and port statements [Yeh 80] would allow easy programming of the greater parallelism solution.

that provide user convenience for a wide range of applications and can be implemented efficiently. Neither choice of communication model stands out over the other. For user convenience (and perhaps also in proving correctness), textually sequential control flow is desirable. The receive primitive should be controlled by a boolean expression, and nondeterminism in receiving should be permitted. The receive should have optional parameters specifying the source and message type. Nondeterministic choice between sending and receiving can lead to improved parallelism. While none of the systems considered meets these requirements exactly, several come close, missing just one of the suggested requirements.

In an independent comparative study of CSP and DP [Welsh 79], a software science metric is applied to a number of example programs written in both languages, so as to obtain a quantitative estimate of the ease of program construction. Their results show DP prevailing in resource synchronization applications (e.g., bounded buffer, alarm clock) and CSP having the edge in applications such as squash asterisks and sorting array. (This is in agreement with our observations.) They note that DP provides a higher level of abstraction than CSP and hence facilitates solution of certain problems, but hinders solution of others. Both this study and ours are based on simple "toy" problems. As experience with distributed computing increases, more realistic applications will emerge, enabling us to better determine desirable features of communication mechanisms.

The DASH kernel, introduced in the following chapter, is designed specifically to meet the needs identified in this survey.

CHAPTER 3. DISTRIBUTED COMMUNICATION FEATURES OF KERNEL

3.1 Concepts and Background

We now introduce some features of the DASH kernel that were designed to facilitate distributed communication. The kernel may be viewed as encapsulating the physical communication architecture and providing operations suitable for building logical communication abstractions. The simple semantics of the operations permits both construction of solutions to problems such as those in Section 2.2, and modelling of the language mechanisms discussed in Section 2.1. An implementor may choose from a variety of communication technologies to construct the kernel, e.g., uniprocessor or local area network.

In order to communicate in a logically distributed manner, it is necessary to transfer control as well as data between communicants. Control information conveys the state of one correspondent to the other. It differs from data in that it is usually brief and requires more prompt attention. Early in the design of DASH, it was decided to keep control and data as separate as possible, for reasons of conceptual clarity. This led to the definition of one set of data structures and operations for handling control and another set for handling data transfer.

While interchanging control information (and data), it is often necessary to await arrival of information from one or more processes. In Section 2.3.2 it was observed that nondeterminism, controlling boolean expressions and the ability to use type and source information all greatly facilitate solution to a variety of problems. The DASH kernel incorporates these ideas, providing the

option to wait, and allowing a user to wait for fairly complex sets of circumstances involving the local process state, control and data information. This same wait is also used in shared memory communication (Chapter 4).

Background

Prior to introducing DASH some related kernels are described and their suitability for our needs is considered.

Thoth, a uniprocessor message-based operating system [Cheriton 79a, Cheriton 79b], has interesting communication features. Interprocess communication is limited to the passing of two types of messages: short (8-word) and long (arbitrary length). Separate primitives are used for each type of message. The operations on short messages have send-reply semantics and the ones on long messages have synchronized send semantics. It is suggested that short messages be used for control information transfer and long messages for data. The Thoth mechanisms, while well-suited for uniprocessor architectures, are less adequate for physically distributed architectures. Since the implementation of send-reply semantics for short messages between remote processes requires at least two transmissions, it does not seem feasible to use such messages for fast transfer of control information. Further, though the receive operation can optionally specify a source, it has none of the other desired characteristics.

In papers suggesting modifications to CSP [Silberschatz 79, Silberschatz 81], an "abstract implementation" has been used to describe the semantics of the revised CSP input/output statements. Operations, which hide physical architecture characteristics, are described and used to implement the modified CSP. Since DASH has

been influenced by these operations, their descriptions are given in detail. Associated with each process are some data structures: an array of booleans (one array element per communication partner), a buffer for data and a boolean flag associated with the buffer. The operations provided are a) Signal(P,F) which sets the boolean variable F in process P to true; b) Put(Q,M) which transfers message M to the buffer of process Q; c) Get(Q,M) which copies the contents of Q's buffer into M and d) wait(F) which waits until the value of the boolean variable F is true (if F is the array of booleans, then a nondeterministic wait occurs for any of the array elements to turn true). The use of these operations in implementing CSP-like input-output is illustrated below. The array of booleans is named Read. The notation used in CSP to denote input of a variable "a" from process Q is Q?a. Similarly, P!b denotes output of b to process P.

```
Process P: Q?a                  Process Q: P!b

Wait(Read[Q])                   Buffer := b
Get(Q,a)                        Signal(P,Read[Q])
Read[Q] := false                Wait[Flag]
Signal(Q,Flag)                  Flag := false
```

Here again is an attempt to separate operations for control (signal, wait) from those for data (get, put). These primitives have the following shortcomings. First, the implementation of the get operation requires some control information to be sent to the sender before data can be transferred (it is suspected that Silberschatz had a uniprocessor in mind when he suggested these primitives). Next, there is no provision for the typing of messages, making it difficult to check whether sender and receiver types match prior to CSP communication. Finally, the wait operation is not as general as we desire (in particular, it lacks controlling boolean expressions).

The Accent kernel [Rashid 81] uses ports for its interprocess communication. Ports are named buffers maintained by the kernel, into which messages may be written and from which they may be read. The send primitive has no-wait send semantics; the receive primitive allows nondeterministic message reception from a set of ports. Messages to remote ports go to a local port read by a network server process, which relays the message over the network; identical intranode and internode communication results. Users of the communication system can specify the type of network service they desire, such as reliability, sequentiality, encryption, flow control. Access to ports is controlled by capabilities, which may be passed around in special messages. The separation of control and data is not maintained in Accent. Accent is designed to provide reliable, transparent communication in general. We need facilities at a higher level, which are flexible and simple to use in programming applications of the type discussed in Chapter 2.

3.2 Kernel Features

3.2.1 Data Structures and Operations

The DASH kernel associates some data structures with each process and provides operations that allow processes to manipulate them. Signals are meant for transfer of control information, the data buffer is used in data transfer, and locks are used for mutual exclusion in shared memory programming (discussed in Chapter 4). DASH also provides a primitive to enable a process to wait for a desired state.

Signals

Processes may communicate state information to other processes using 'signals'. A signal has two attributes: a source process name and a type. The type field of a signal is small and of fixed length (8 or 16 bits), and its interpretation is left up to the kernel user. Each process has a Signalbox, which stores received signals. The Signalbox resembles a set in mathematics in that it can only store distinct signals (i.e. signals whose attributes are distinct). The operations on Signalbox are:

1. signal(p,t): When this operation is invoked by process q, the signal with the attributes <q,t> is added to the Signalbox of process p.

2. test(p,t,p',t'): This boolean function indicates whether the specified signal <p,t> is present in the Signalbox of the calling process. The output parameters p' and t' are not used if p and t are both specified. A "wild card" facility exists allowing p or t (or both) to be unspecified. If a match is found, the matched signal is returned as <p',t'>. Should more than one signal match a wild card specification, the selection (of the matched signal) is nondeterministic. (An implementation should be fair in selecting the match.)

3. discard(p,t): The specified signal is removed from the local Signalbox, if it is present.

These operations access the Signalbox indivisibly. This means that the insertion of a signal in the destination process's Signalbox (as part of the signal operation), cannot occur concurrently with other signal insertions, or with test and discard operations on the same Signalbox.

Buffer

The Buffer is primarily used for data transfer. It receives incoming messages of size not exceeding Buf_size. The buffer size,

Buf_size, is the same for all processes and is determined at implementation time. The Buffer has an indicator which is set each time a data message arrives. The operations on the Buffer are:

1. transfer(p,b,c): c units[1] of data starting at address b in the invoking process's address space are transferred to the Buffer in process p, setting its data indicator.

2. datatest(): This boolean function returns 'true' if the invoking process's buffer indicator is set and 'false' otherwise.

3. datasize(): The size of the data message in the invoker's Buffer is returned.

4. datacopy(b,i,n): n units of data starting at unit i are copied from the Buffer to the invoker's address space starting at location b.

5. datareset(): The Buffer data indicator is reset.

A process can only perform operations 2 through 5 on its local Buffer. Operations 3 and 4 are meaningful only when the Buffer is non-empty. There is no safeguard against a transfer occurring while another transfer to the same destination is taking place or while the destination process is copying the Buffer out. Also, two successive transfers will overwrite the Buffer. These situations may be averted by suitable use of signals prior to a transfer. (Proper usage of the kernel requires that processes cooperate.)

The Wait Primitive

An additional primitive is available to enable a process to wait until its local state, the state of the signalbox, and the state of the data buffer together satisfy a user specified condition. The

---

[1]The unit size is defined at implementation time.

user indicates the condition he desires by writing a procedure, termed a condition procedure, which computes a boolean value based on predicates involving process local variables, and the DASH test and datatest operations. Side effects in this boolean procedure are not forbidden. The wait primitive is of the form:

wait(<condition_procedure_name>, <argument_list>)

When wait is invoked, the supplied boolean procedure is applied to the argument list and control is returned to the invoker if a 'true' value is computed. Otherwise the caller is blocked until the procedure returns a 'true' value. (The implementation should re-execute the condition procedure each time a signal or data message arrives.)

Example 3-1

```
procedure signalwait(source:process_name;type:signaltype)
        returns(boolean);
begin
    if test(source,type) then
        begin discard(source,type); return(true) end
    else return(false)
end
```

This condition procedure[2] when used in a wait:

wait(signalwait, <P,T>)

causes the invoker to wait until a signal from the process P of type

---

[2]The programming language Pascal [Jensen 75] is used here and in most of this thesis for algorithm descriptions. The syntax of functions is modified: they are called procedures and they include a return statement that contains the value of the function (as in C [Kernighan 78] and Mesa [Mesa 79]). The strict typing in Pascal has been relaxed somewhat.

T is received. (Notation: the angle brackets surround the argument list.) Since both source and type are specified, the test operation is written in its abbreviated form with only two input arguments. The condition procedure signalwait has a side effect: it discards the received signal, since note has been taken of it.

Example 3-2

This example illustrates how a process P may use DASH to transfer data to process Q. P first sends a signal to Q requesting allocation of Q's buffer. On receiving this signal, Q sets aside its buffer for P and indicates this through a confirming signal to P. Then P transfers the data and Q waits for the data and copies it out.

```
Process P:                          Process Q:

signal(Q,req);                      wait(signalwait,<P,req>);
wait(signalwait,<Q,ok>);            signal(P,ok);
transfer(Q,send_addr,size);         wait(datatest, <>);
                                    size:=datasize();
                                    datacopy(rcv_addr,0,size);
                                    datareset();
```

The signalwait condition procedure introduced in Example 3-1 is reused here. The data is to be transferred from send_addr in P's space to rcv_addr in Q's address space. The receiver awaits data arrival by issuing a wait call, using the DASH datatest operation as a condition procedure. The datacopy operation copies the entire data message out (the first unit is numbered 0). The sequence of operations in process Q from wait(datatest,<>) up to datareset() occurs often enough that we introduce a special procedure datawait:

```
procedure datawait(receive_addr:address)
var size:integer;
begin
    wait(datatest, <>);
    size:=datasize();
    datacopy(receive_addr,0,size);
    datareset()
end;
```

This procedure will be used in subsequent descriptions.

The description of DASH features above is informal. In Appendix B, the semantics of the distributed features of DASH is specified more precisely through an implementation in CSP.

3.2.2 Discussion of Features

Signals

Signals are designed to convey control information. This information is presumed to be small and to require prompt delivery. Since signals are to be used as building blocks for a variety of communication abstractions, a message passing model was adopted. To allow a receiver to choose between requests, the set of alternatives must be easily (locally) accessible. This resulted in the use of the Signalbox to buffer signals. The small size of signal messages makes buffering many of them feasible. Having decided on buffering, no-wait semantics for sending signals can easily be implemented.

What should the attributes of a signal be? Since signals convey state information, it is necessary to know whose state has been transmitted, i.e., the name of the source process. A process could send signals in different states conveying different information; hence, it is convenient to know what the signal is

about, i.e., the type. One also has to decide how much information the type field should convey. At one extreme, it could convey no information: all the information must be communicated via the data transfer mechanism. In addition to being inconvenient, this would cause both control and data to be transmitted using the same facility. At the other extreme are arbitrary length type fields. This would result in large signals, greatly increasing the cost of buffer space at the receiver. An intermediate size has to be found based on the amount of control information one expects typical applications to pass. For most of the applications considered, an 8 or 16 bit type field is sufficient. As we shall see in Section 3.3, there are a few applications where arbitrary length type fields would be more convenient.

The Signalbox can store only distinct signals. This decision was made in the interest of limiting the size of the Signalbox while guaranteeing buffer space for distinct signals. Experience shows that this does not restrict applications significantly. (If signals are used to count occurrences of an event, then handshaking may be necessary to avoid miscounting due to multiple identical signals arriving between successive tests on the Signalbox.) If there are a fixed number of processes, say n, in the system, and the type field can encode m types, then the number of distinct signals is m x n. If this is large, space for a smaller number of signals may initially be allocated, and more space may be allocated dynamically as the need arises. In the applications that we have studied, the number of outstanding signals at any instant is much smaller than the number of distinct signals.

Access to the Signalbox was defined to be indivisible. If the access is implemented as single indivisible machine instructions,

this requirement is easily met. Otherwise, the implementing environment must provide a means to ensure indivisible access.

The dispatching of signals with a signal operation and the waiting for signals with a wait on a test operation resemble the V and P operations on a binary semaphore. The presence or absence of the signal is akin to the value of the binary semaphore. An invoker of a wait(signalwait,...) operation is blocked if the signal is not present, and is allowed to proceed when it is present. There are differences, however. The DASH test operation allows the invoker to determine the status of the signal. It may then decide to wait for it. Also, a signal has only one sender and only one process may be waiting for it, unlike the semaphore which may have several invokers of the P and V operations.

We have side-stepped the question of what happens when a signal is sent to a nonexistent process. This is because we have assumed a static process configuration; hence, the question does not arise. Some thoughts concerning dynamic processes may be found in Section 3.4.

Buffer

The presence of a data buffer permits no-wait semantics for the transfer operation. There are, however, two issues concerning the buffer that effect the way the senders of data use data buffers: buffer size and buffer allocation. By fixing the size of data buffers, space is assured for data messages up to that size. Larger messages have to be sent in pieces (discussed in Section 3.4). If the buffer size is chosen based on the typical data transfer size in the intended application, fragmenting of large data messages should occur infrequently. Data messages being larger than signal messages,

the buffering of all possible data messages (as is done for signals) is not feasible. With limited buffering, control information regarding the allocation of the single buffer needs to be passed to the sender before it transmits the data. In keeping with our desire to separate control from data, it was decided not to build such buffer allocation semantics into a data transfer primitive. Instead, signals must be used to ensure that the buffer is allocated to the sender before he transfers data, i.e., processes must cooperate.

The arrival of a data message is indicated by the setting of the buffer data indicator. Instead of having a special indicator which is set by a data message, the possibility of using a signal to achieve this effect was considered. The sender would then use a simpler version of transfer with no data indicator, followed by a signal of a special type indicating data arrival. This alternative was not pursued because signal messages and data messages could be treated differently by an implementation, such that order of reception of signals and data may differ from the order of transmission. It may be noted that the kernel description leaves unspecified the ordering of signals and data by the transmission medium. User protocols may be needed to ensure correct operation.

Wait

The flexibility of the wait primitive is made possible by allowing the user to supply a condition procedure. By writing suitable condition procedures, a user can, for example, wait for a signal or for data to arrive, or wait for one of a possible set of signals, each of which is acceptable in only some states of the local process. The user may implement nondeterministic selection between a set of alternatives, or may assign priorities to alternatives; this

is in addition to a limited nondeterministic selection between signals provided by the DASH test primitive. Examples of each of these appear in Section 3.3. The kernel requires the user to state these conditions, since it cannot anticipate all possible sets of conditions on which applications may wait. The price paid by the user to obtain this flexibility is the additional effort required to write condition procedures. When DASH is used to implement language mechanisms in which the conditions may be determined during compilation, the compiler can generate the required condition procedures.

The implementation of wait requires the condition procedure to be executed each time a signal or a data message arrives. Since this procedure can execute for an arbitrary length of time (in fact, it may not terminate), it is inappropriate for this to be executed by a system scheduling routine to determine if a process should be unblocked. Instead, it is preferable to implement wait by having the invoking process block until there is input, then have it execute the supplied condition procedure to decide whether or not to block again. The scheduler now simply schedules the blocked process after each input message. This strategy may cause some processes to be scheduled unnecessarily (because they block again after executing the condition procedure). Both the implementational specification of DASH (Appendix B) and the DASH UNIX implementation cause waiting processes to be unblocked when messages arrive.

The semantics of wait resembles those of the await statement in conditional critical regions [Brinch Hansen 73]. The statement

await B

when executed inside a critical region causes the boolean expression

B to be evaluated. If B is true, the invoker continues; otherwise, the invoking process releases the critical region, and is placed on a queue associated with the critical region. When the critical region has been successfully completed by another process, all processes waiting on the queue are removed from the queue; they may enter the critical region again to reevaluate B. If B is false, they reenter the queue as before. The similarity between the wait operation in DASH and the await operation in conditional critical regions is the concept of blocking a process on a condition, allowing it to retest the condition each time another process may have made the condition possible. The differences are that each process in DASH waits on a queue of size 1, whereas in conditional critical regions, several processes await conditions and all are unblocked when there is a chance that their conditions are true. Implementation of conditional critical regions in DASH is considered in Section 4.3.

3.3 Examples of Kernel Usage

The distributed features of the DASH kernel are illustrated through examples of problem solutions and modelling of language features.

3.3.1 Problem Solutions

Example 3-3

A distributed implementation of the bounded buffer problem (Section 2.2.2), consists of a buffer server process responding to producer and consumer requests. To program it with DASH, we first need to describe the pattern of interactions (in terms of signals and data) between producer, buffer and consumer processes. The producers

attempt to send portions to the buffer server process via data transfers. Before the transfer they need to ensure they have exclusive permission to write the server's data buffer (as in Example 3-2). Each producer requests permission with a signal of type insert and then waits for the server to send an ok signal. Then data is transferred. Consumers attempt to obtain portions from the server, by sending signals of type remove. They then wait until their data buffer is written into. The remove signal implicitly gives the server permission to write the requester's data buffer. These interactions are shown in Figure 3-1 for a single producer and a single consumer. In the figure, single lines with arrowheads denote signals, and double lines denote data.

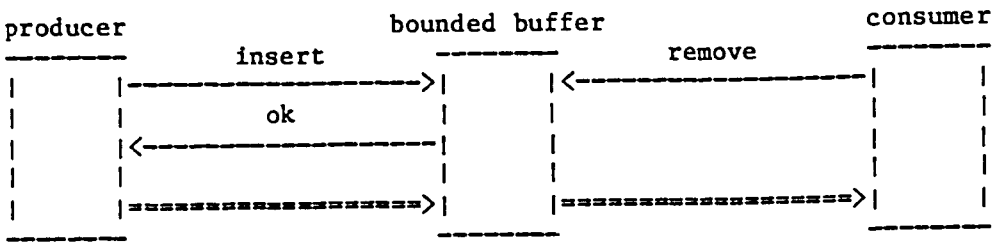


Figure 3-1: Interactions with Bounded Buffer

The code for the producer and consumer processes outlined in Figure 3-2 follows directly from the interactions illustrated in Figure 3-1. BBUF denotes the bounded buffer server process. The code for the buffer process is interesting. The process services two types of requests: inserts and removes. It can choose to service an insert request, only if space is available in the buffer and an insert signal is pending. Similarly, it can choose to service a remove request only if there are portions in the buffer and a remove signal is pending. The buffer process invokes a wait with a

```
producer:                          consumer:

...                                ...
signal(BBUF,insert);               signal(BBUF,remove);
wait(signalwait,<BBUF,ok>);        datawait(data);
transfer(BBUF,data,n);             ...
...                                ...
```

```
bounded buffer:

var inp,outp:integer;
    source:process_name; insert,remove,type:signaltype;
    buffer:array [0..N-1] of portion;

procedure reqselect() returns(boolean);
begin
    if (inp-outp<N) and test(*,insert,source,type) then
        begin discard(source,type); return(true) end;
    if (inp>outp) and test(*,remove,source,type) then
        begin discard(source,type); return(true) end;
    return(false)
end;

begin
    inp:=0; outp:=0;
    loop
        wait(reqselect,<>);
        case type of
        insert: begin
                   signal(source,ok);
                   datawait(buffer[inp mod N]);
                   inp:=inp+1
                end;
        remove: begin
                   transfer(source,buffer[outp mod N],n);
                   outp:=outp+1
                end;
        end
    end loop
end bounded buffer;
```

Figure 3-2: DASH solution for Bounded Buffer

condition procedure reqselect to decide on (and perhaps wait for) the next serviceable request. The first if statement in reqselect, checks if space is available and an insert signal is pending. The source parameter in the test operation is left unspecified (indicated by an asterisk). If an insert signal is found, its source is returned through the third parameter of the test. The body of the if statement discards the signal and returns the value true. If insert is infeasible then the feasibility of a remove is similarly checked in the second if statement. If remove is also not feasible, the value false is returned. On completion of the wait, the buffer process uses the global variable type to determine which request was chosen and completes the interaction processing and the processing needed to service the request.

Note that the boolean expressions indicating availability of space and of portions are included in the reqselect procedure. This makes the wait convenient to use. Reqselect is not fair in that it checks for inserts first and then for removes. If this is a problem, reqselect should be rewritten (perhaps randomly selecting one or the other condition to check first). Wait may call reqselect several times before it returns a true value. In this case, the boolean expressions in the local process variables (which do not change during a wait) are unnecessarily reevaluated. This may be avoided by assigning these expressions to a boolean variable outside reqselect and using these variables instead of the expressions inside. In this example, global variables rather than argument lists are used to pass information to and from the condition procedure.

The bounded buffer example illustrates the convenience offered by condition procedures and the DASH wait operation. Also, the general characteristics of interaction with a server are illustrated:

the producer-server interaction is typical for server requests which have input arguments only; the consumer-server interaction is typical when requests have output arguments only.

Example 3-4

A pipelined algorithm for sorting a set of numbers using an array of processes was presented in Section 2.2.1. Numbers are successively fed into the first process in the pipeline. On completion of input, the pipeline processes contain the array sorted in nondecreasing order. In this example, a process in this pipeline is programmed. The DASH interaction necessary to fetch a number from the predecessor process consists of sending it a request signal and then waiting for the data, containing the number, to arrive (Figure 3-3). Once the data has arrived, and has been copied out of the data buffer, another request signal may be sent to the predecessor before the input number is processed. This allows greater parallelism. After the input is processed, the result may be sent to the successor (after a signal is received from it) and then fresh input data from the predecessor is awaited. The solution in Figure 3-4 strives for increased parallelism, by handling the request from the successor or input data from the predecessor, whichever comes first.

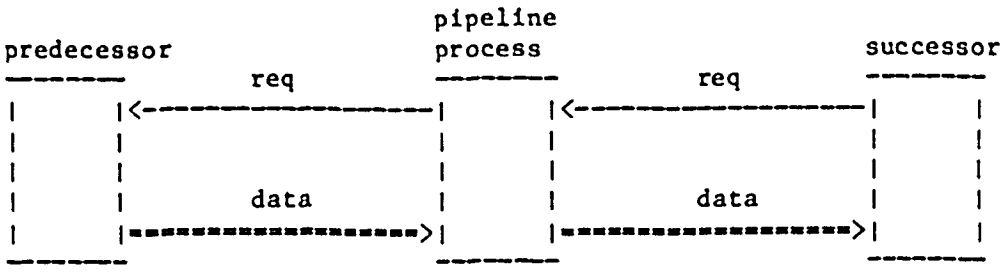


Figure 3-3: Interactions in a sorting array

```
sorting array process:

var m,n,big:integer;
    pred,succ:process_name; req:signaltype;
    event:(req,data);

procedure req_or_data() returns(boolean);
begin
    if test(succ,req) then
    begin
        discard(succ,req);
        event:=req;
        return(true)
    end;
    if datatest() then
        begin event:=data; return(true) end;
    return(false)
end;

begin
    signal(pred,req);
    datawait(n);
    signal(pred,req);
    datawait(m);
    signal(pred,req);
    loop
        if m<n then begin big:=n; n:=m end else big:=m;
        wait(req_or_data,<>);
        case event of
        req:   begin
                   transfer(succ,big,1);
                   datawait(m);
                   signal(pred,req)
               end;
        data:  begin
                   datawait(m);
                   signal(pred,req);
                   signalwait(succ,req);
                   transfer(succ,big,1)
               end;
        end
    end loop
end sorting array process;
```

Figure 3-4: DASH solution for sorting array

The condition procedure req_or_data allows the process to wait for either request or data. When the wait terminates, the variable event indicates the condition that occurred. If the request signal came, then the result is transferred to the successor (the number is assumed to require one unit of data) and input data is awaited from the predecessor. If data from the predecessor came, it is copied and a signal from the successor is awaited. In this example we see the use of more than one condition procedure (signalwait and req_or_data). Also, the ability in DASH to handle interactions with either the predecessor or successor processes in FCFS order enhances the parallelism of the solution to the problem.

Example 3-5

Some of the interactions between master and slave processes in the two step commit protocol (Section 2.2.3) are now described with DASH primitives[3]. The focus is on the interaction necessary between master and slaves to decide whether to commit or to abort a transaction. The crash recovery aspects of the protocol are ignored. In this implementation of the protocol, slaves respond to the master's query only when they have stayed healthy during update transmission. If they have recovered from a recent crash, they do not reply (since updates may have been lost or damaged). The master uses a timer process, which it starts when it makes the query. If one or more replies are missing when a timeout occurs, then the transaction must be aborted. Updates are recorded on a stable medium

---

[3]The protocol is used in this example solely for demonstrating DASH usage. An explanation of how it achieves its objectives may be found in Section 6.2.2

through use of a Stable_write procedure. The procedure Save_state saves the process state on stable storage. A hardware clock is assumed, which sends a signal to the timer process each time it "ticks".

Most of the master's decision making regarding the fate of a transaction happens in its condition procedure decisionwait (Figure 3-5). If a timeout signal is received, an abort is decided upon. Otherwise, a check is made to see if all the slaves' affirmative votes have been received; if so, commit is decided upon. If none of these conditions have arisen yet, the wait continues.

If a slave is of unhealthy disposition, we are not concerned with its interactions. Healthy slaves respond affirmatively to the master's query and then obey the master's command regarding the transaction. The timer process, for simplicity, serves only one client, signalling it after a fixed interval. (A more general timer appears in Section 5.3.) It receives signals from the master (requesting it to start and stop) and tick signals from the clock. It is possible that the master's stop signal to the timer and the timer's timeout signal to the master cross in transit. This could leave the signalboxes of the master and the timer with obsolete signals which may confuse further interactions. This eventuality is handled by requiring the timer to respond to a stop command by sending a timeout signal; if the timer runs out, then on sending the timeout signal it waits for a stop signal. Similarly, the master responds to a timeout with a stop signal, and if it initiates the stop it waits for the timeout.

After having run through this protocol once, each of the processes needs to clean up after itself before reexecuting this protocol. In certain states, the processes should be able to absorb

```
master:

var decision:(commit,abort); i: [1..N]; timer: process_name;
    slave: array [1..N] of process_name;
    query,ok,timeout,com,abt: signaltype;

procedure decisionwait() returns(boolean);
var all:boolean; i:integer;
begin
    if test(timer,timeout) then begin
        signal(timer,stop);
        discard(timer,timeout);
        decision := abort;
        return(true)
    end;
    all := true;
    for i := 1 to N do all := all and test(slave[i],ok);
    if all then begin
        signal(timer,stop);
        decision := commit;
        for i := 1 to N do discard(slave[i],ok);
        return(true)
    end;
    return(false)
end;

begin
    ...
    ...
    <send updates to slaves>
    for i := 1 to N do signal(slave[i],query);
    signal(timer,start); /* start timer */
    wait(decisionwait,<>);
    case decision of
    commit: begin wait(signalwait,<timer,timeout>);
                  Save_state("commit decided");
                  for i := 1 to N do signal(slave[i],com);
                  Save_state("transaction complete")
            end;
    abort:  begin Save_state("abort decided");
                  for i := 1 to N do signal(slave[i],abt);
                  Save_state("transaction aborted")
            end;
    end;
    ...
    ...
end master;
```

Figure 3-5: Two Step Commit Protocol: Processes

```
slave:

var master: process_name;
    query,ok,com,abt,command: signaltype;

procedure commandwait() returns(boolean);
var source: process_name;
begin
    if test(master,*,source,command) then
        if (command = com) or (command = abt) then
        begin discard(source,command); return(true) end;
    return(false)
end;

begin
    ...
    ...
    <interact with master to get updates>
    wait(signalwait, <master,query>);
     /* write updates on stable storage */
    Save_state("query received");
    signal(master, ok);
    wait(commandwait, <>);
    case command of
    com: begin
            Stable_write(<update>);
            Save_state("update complete")
        end;
    abt: Save_state("update abort");
    end;
    ...
    ...
end slave;
```

Figure 3-5, continued

```
timer:

const interval: integer;
var time: integer;
    master,clock: process_name;
    start,stop,tick,timeout,input: signaltype;

procedure inputwait() returns(boolean);
begin
    if test(master,stop) then
        begin
            discard(master,stop);
            input := stop;
            return(true)
        end;
    if test(clock,tick) then
        begin
            discard(clock,tick);
            input := tick;
            return(true)
        end;
    return(false)
end;

loop
    wait(signalwait, <master,start>);
    discard(clock,tick); /* initialize */
    time := interval;
    while time > 0 do
    begin
        wait(inputwait, <>);
        case input of
        stop: begin signal(master,timeout); goto done end;
        tick: time := time-1;
        end;
    end;
    signal(master,timeout);
    wait(signalwait,<master,stop>);
    done:
end timer;
```

Figure 3-5, concluded

duplicate signals or answer queries from recovering processes [Baer 81]. These actions, being associated with recovery, do not appear in the code.

Using CSP to program the two step commit protocol is inconvenient for a couple of reasons. The master's requirement to wait until either all affirmative responses are in or a timeout (or negative response) occurs, is clumsy to program in CSP since there is no facility to wait for a conjunction of inputs. The mechanisms in CP, DP, PLITS, ECLU would be inconvenient for the same reason. The ability to interrogate the number of pending invocations in ADA and SR allows less cumbersome solutions. The other (and perhaps more serious) objection to CSP arises because of its synchronized semantics. If a slave has crashed, the master will block in its attempt to query. The remedy for this is to introduce an intermediate process for each slave. The mechanisms in ADA, SR, CP, and DP, having synchronous semantics, would not fare any better. If the master's query contains some data describing the transaction, then in DASH a signal will not suffice and a more complicated solution is necessary: the master must detect crashed slaves which do not cooperate in data transfer.

In summary, this example once again illustrates the flexibility of DASH in programming the master. Also, a method of generating timeouts is suggested. The entire example uses signals alone and no data. When data is also necessary, the solution will be more complex.

3.3.2 Modelling Features of Extant Languages

The use of the DASH kernel is further demonstrated in examples where mechanisms representative of a number of programming languages

are modelled. A fully synchronized message transfer mechanism (CSP), a procedure call approach (ADA) and some variations are presented. Discussion of asynchronous (no-wait send) mechanisms is deferred to Chapter 5, since these are conveniently implemented with hybrid techniques.

Example 3-6

For communication to take place in CSP, the sending process must execute an output statement furnishing a destination process name and a typed expression, and the receiver likewise must execute an input statement naming a source process and a typed variable. Communication occurs only when both parties have executed their respective input/output statements and the type of the output expression matches that of the input variable. An exchange of DASH signals, as shown in the code below, can implement these semantics. We assume the existence of two arrays: sigtype, used to map types in the language into signal types; and length, used to determine the number of data units required for the representation of each language type. Without loss of generality, the quantity sent can be assumed to be a variable instead of an expression. It is possible to save one signal transmission, by having the receiver initiate the dialogue and the sender perform the transfer on receipt of the receiver's signal. This strategy does not generalize to the case when the receiver has several input commands in guards and is nondeterministically waiting on them. The handling of input commands in guards is discussed in the following paragraphs.

```
Process P: Q!(a:T1)          signal(Q,sigtype[T1]);
                             wait(signalwait,<Q,ok>);
                             transfer(Q,a,length[T1]);
```

```
Process Q: P?(b:T2)          wait(signalwait,<P,T2>);
                             signal(P,ok);
                             datawait(b);
```

We assume that all guards have input commands in them. (The general case can be accommodated with a few simple modifications.) Associated with each guard in a guarded command, is a record containing three fields: a boolean containing the value of the boolean expression preceding the input command in the guard, the sender process name, the signaltype associated with the information being sent and the address of the input variable. Since several guards may exist in a guarded command, a record is needed to keep a list of the guards in the command. The guardlist record has fields for the number of guards, an array for the guards and an integer for the index of the selected guard. In implementing a CSP guarded command, a guardlist has to be constructed for the command (presumably by a compiler). Prior to executing the command, all the boolean expressions in guards must be evaluated and their results stored in the enabled fields of the guards. Then the routine cspreceive (in Figure 3-6) is called. On return, the selected field in the guardlist indicates the guard selected. The statement following the guard is then executed.

The procedure cspreceive is a generalization of the code for the simple receive presented earlier. It uses a condition procedure guardwait. This procedure scans the guardlist for enabled guards, and tests the signals associated with them. If one is found, the index of the guard is noted in the guardlist. For completeness, the sending code is also included in Figure 3-6 as a procedure.

The routines in Figure 3-6 have some deficiencies, and these are now addressed. A simple change to guardwait is needed to improve

```
guard = record
            enabled: boolean;
            source: process_name;
            stype: signaltype;
            addr: address of input variable
        end;

guardlist = record
                nguards: integer;
                list: array [1..nguards] of guard;
                selected: [1..nguards]
            end;

procedure capreceive(var glist:guardlist);

    procedure guardwait(var glist:guardlist) returns(boolean);
    var i: integer;
    begin
        for i := 1 to glist.nguards do
            with glist.list[i] do
                if enabled then
                    if test(source,stype) then begin
                        discard(source,stype);
                        glist.selected := i;
                        return(true)
                    end;
        return(false)
    end guardwait;

begin
    wait(guardwait,<glist>);
    with glist.list[glist.selected] do begin
        signal(source,ok);
        datawait(addr)
    end
end capreceive;

procedure capsend(dest:process_name;t:type;snd_addr:address);
begin
    signal(dest,sigtype[t]);
    wait(signalwait,<dest,ok>);
    transfer(dest,snd_addr,length[t])
end capsend;
```

Figure 3-6: Modelling of CSP input/output

the fairness of the condition procedure: initiate scanning of guards where it left off the last time. The CSP guard is defined to have failed if the process named in the input command in the guard has terminated. This aspect is not dealt with in Figure 3-6, but can be easily incorporated by having a terminating process send all its recipients an appropriate signal[4]. CSP also allows an input command in a guard to list an array of processes as possible sources. This too can be easily accommodated by modifying the data structures and the guardwait procedure. The cspsend and cspreceive routines are written assuming that a single simple variable is transferred. Since there are a small number of elementary types in the language, the arrays sigtype and length may be predefined. If lists of variables or complex structures of variables are to be transmitted, then a signaltype must be associated with them and the structures must be flattened prior to transfer. The DASH datacopy operation, which allows access to parts of the data buffer, should prove useful in unflattening argument lists on reception. The number of possible structures grows exponentially with the number of variables in the structure. This causes problems in mapping structured types into a small signaltype field in a signal even with a small number of elementary types. This problem is discussed at length in Section 3.4.

A compiler has been implemented which translates CSP into C processes that use DASH for communication [Yenbut 81, Odano 82]. This demonstrates that the issues above can be resolved in practice.

---

[4]In some CSP programs, processes do not terminate individually, but collectively in sets. Algorithms have been developed for determining termination in such cases [Francez 80].

Example 3-7

The interaction between calling and called processes in Ada is not dissimilar to CSP input/output. Associated with each Ada entry is a signal type. The sending of parameters to a called process in an entry call proceeds exactly as a send in CSP. The behavior of the called process differs from that of a CSP receiver in that it does not name the source of an invocation. The called process, on completing execution of the entry statement, transfers return arguments back to the caller. Since the caller is blocked awaiting the transfer, there is no need for any further synchronization with signals before transferring the data to the caller's data buffer.

In Figure 3-7, CSP's guard and guardlist records have been renamed entry and entrylist. The condition procedure, entrywait, uses the wild card option of the test operation to select an entry invocation. The receiver is expected to initialize the entrylist record, use the procedure adareceive, service the entry call and then use adareply to return arguments. The caller uses the procedure adacall to transmit parameters to and from the entry.

The routines in Figure 3-7 are geared towards the most commonly used features of Ada entry calls. The Ada language provides some variations of this basic scheme: timeouts for the calling and called processes and families of entries. A timeout feature can be incorporated using a timer, similar to the one in the two step commit protocol example (Section 3.3.1). Timeouts in Ada communication are further examined in Section 5.3. Ada's families of entries consist of an array of entries having the same name, but differing in index. The calling process specifies which member entry is to be invoked by providing an index in the entry call. Entry families provide a

```
entry = record
            enabled: boolean;
            caller: process_name;
            stype: signaltype;
            addr: address of input variable
        end;

entrylist = record
                nentries: integer;
                list: array [1..nentries] of entry;
                selected: [1..nentries]
            end;

procedure adareceive(var elist:entrylist);

    procedure entrywait(var elist:entrylist) returns(boolean);
    var i: integer; t: signaltype;
    begin
        for i := 1 to elist.nentries do
        with elist.list[i] do if enabled then
            if test(*,stype,caller,t) then begin
                discard(caller,stype);
                elist.selected := i;
                return(true)
            end;
        return(false)
    end entrywait;

begin
    wait(entrywait,<elist>);
    with elist.list[elist.selected] do begin
        signal(caller,ok);
        datawait(addr)
    end
end adareceive;

procedure adareply(caller:process_name;t:type;addr:address);
begin transfer(caller,addr,length[t]) end;

procedure adacall(dest:process_name;t:type;
        snd_addr,rep_addr:address);
begin
    signal(dest,sigtype[t]);
    wait(signalwait,<dest,ok>);
    transfer(dest,snd_addr,length[t]);
    datawait(rep_addr)
end adacall;
```

Figure 3-7: Modelling of ADA entry call

limited capability to selectively receive messages based on their content. In Section 2.3.1 it was observed that this capability requires buffering of messages at the receiver. Though DASH does not have a data buffering facility that can conveniently support receiving by content in general, it is able to cope with Ada's entry families. The strategy involves using different signaltypes to represent different entry indices. However, if the range of indices is large, the number of different signaltypes can soon be exhausted. This problem is similar to the problem of sending arbitrary structured types in CSP and is also discussed further in Section 3.4.

The examples in this section have focussed on the communication features of Ada and CSP in some detail. The strategy, for modelling the other languages with synchronous primitives mentioned in Section 2.1, should be apparent. SR differs from Ada in that it does not have built in timeouts and entry families; it has selective receiving by content, which is difficult to model with DASH[5]. The primitives in CP also resemble the basic Ada primitives, but allow entries to list clients they will service. This additional feature may be modelled by constructing an appropriate condition procedure. The DP procedure call may be modelled by passing the procedure name and parameters to a surrogate (remote) process using Ada-like communication. The surrogate process interacts with the server through a variant of a monitor which allows nondeterministic

---

[5]We do not consider selective receiving by content to be particularly desirable. Should it be necessary to implement it in DASH, it can be accomplished by buffering messages at the receiving end.

waiting on several conditions, and implicitly signals waiting processes on monitor exit.

In Section 2.3.2, some desirable features of language primitives were listed: sequential control flow, boolean expressions, nondeterminism and optional source and type parameters in receiving, and nondeterminism between sending and receiving. The facilities in DASH are adequate to build all these features.

If a language's primitives are implemented using DASH disregarding the interaction context, then use of the language in solving problems may result in unnecessary signals. For example, if the bounded buffer problem was written in CSP, which in turn was implemented using the routines in Figure 3-6, the consumer-buffer interaction would use 4 signals and 2 data transfers. The solution programmed directly in DASH requires only one signal and one data transfer (Figure 3-1). The latter case makes use of knowledge about the interaction in determining what signals are needed. An interesting area for study is the design of a high-level language that allows a user to provide interaction information (perhaps through flow expressions [Shaw 78]) to a compiler which translates the language mechanisms into DASH kernel calls, allowing it to optimize transmissions. The attempt to classify interactions in Section 2.2, is a step in this direction.

3.4 Some Limitations of DASH

The previous section demonstrated what the DASH kernel could do. This section points out what it can not do, and what it has difficulty in doing.

Restrictions placed by finite type field

The problem of sending arbitrary structured types in CSP, and the problem of implementing large families of Ada entries, result from limitations on the amount of information that can be conveyed in the type field of a signal. The existence of considerable buffer space at the receiver for data would alleviate this problem somewhat. In such a scheme, the type field could be sent with the data, and the cost to buffer the type information would probably be small relative to the cost to buffer the data. For the CSP case the following alternatives exist.

1. Limit the size of structured types so that the number of signal types required is a small manageable number. This effects the definition of the CSP language.

2. Pass information about the structured type through data transfers (with prior signal exchange). If the receiver is not willing to accept the structured type, it has two choices: it can buffer the information, greatly increasing the buffer space requirements at the receiver, or it can discard the information, forcing the sender to retransmit it later. In either case, interactions become more complex.

3. Let the compiler assign signal types to structured types that are encountered in communication statements. This requires that the entire CSP program be compiled at one site.

4. Require the programmer to name all structured types (that appear in communication statements) in a preamble which is used in the (perhaps remote) compilation of each process. This, too, affects the CSP language.

If the CSP language may not be altered, then options 2 and 3 are open, with option 3 being preferred if separate (remote) compilation of processes is not required. In my opinion, option 4 is preferable in general, because it forces the programmer to explicitly identify the communication interfaces between processes.

For the Ada entry family case, only the first two options are meaningful, with the first being restrictive, but efficient. It is felt that the applicability of entry families is sufficiently limited to justify not having them in the language. The user can achieve the effect of entry families by buffering entry invocations himself. The language would have to provide him the name of the client process and the ability to send a reply to it.

Buffer size

Should there be a need to send more data than can fit in the data buffer, the data would need to be broken up and sent in pieces. The sender would first assure itself of exclusive access to the receiver's data buffer (as before). It would then dispatch a piece at a time, waiting for a confirmation signal from the receiver between successive data transfers.

Dynamic processes

Since DASH is not geared for dynamic processes, some considerations on how it may be extended to allow for them are outlined. The two major issues are process naming, and communication with nonexistent processes. Some naming concerns are examined in [Pouzin 78]. One simple technique suggested is a hierarchical scheme: the process name is a concatenation of the node name and the local name. Each node can accommodate up to a certain maximum number of processes. The local name space could, however, be much larger, so the reuse of names is less likely to lead to confusion because a large interval of time elapses between consecutive uses of a name. At each node a facility for creating processes should exist. Process creation involves checking that the number of processes does not

exceed the limit, allocating a name, creating the process lated data structures (including the DASH signalbox and data b fer), starting execution of the process image and returning the new ocess name to the creating process. It should also be possible t pass process names through data messages.

The problem of sending a signal or data message to a nonexistent remote process is made harder because of the -wait semantics chosen for signal and transfer. (If the sende were waiting, then it could be notified of the nonexistent destin ion.) Plits and ECLU, with no-wait semantics, generate exceptions w n the process next uses the communication facility. Similarly, DAS could generate an exception the next time any of its operatio are invoked. It is generally true that programming an applicat n is considerably harder when one has to protect against none stent processes.

3.5 Summary

The DASH kernel has been introduced in this chapter The kernel operations are simple, yet powerful. Through example they have been shown to be useful in programming a number of commun ation problems, as well as in implementing higher-level mechanisms. Some deficiencies in the kernel have been identified, and reasonab ways of handling them have been suggested.

CHAPTER 4. KERNEL FEATURES FOR SHARED MEMORY

4.1 Shared Memory Programming

Processes can use shared memory for communication. However, interference between processes is possible and can lead to undesirable time-dependent behavior. In order to protect against this undesirable interference, it is often required that certain types of access to shared data be performed atomically, i.e., the invoking process completes the entire operation without interference from other processes. Also, when processes cooperate, there is often a need for them to synchronize, i.e., be assured that other processes are in certain states. In awaiting synchronization, it is prudent when the processor is shared between several processes, to give up the processor until the desired state occurs.

Recognizing these needs, several techniques have been suggested in the literature:

1. Mutual exclusion through the clever use of shared variables [Shaw 74, Section 3.3] with conventional programming constructs: Atomicity of "critical regions" is ensured by testing and setting of shared variables prior to and following the execution of the region. This approach is impractical in that it is non-intuitive and inefficient.

2. Semaphores were introduced in Section 2.2.2. Semaphores have been widely used in the literature for synchronizing processes and for implementing critical regions. They have also been used as a completeness measure for the expressive power of new primitives.

3. Locks [Dennis 66]: Atomic lock and unlock operations are defined on a shared boolean lock variable. A process invoking the lock operation must wait until the variable is false and then set it to true. The unlock operation sets the lock variable to false. Locks are useful for implementing mutual exclusion.

4. Conditional critical regions were mentioned in Section 3.2.2. They group shared variables into resources and provide exclusive access to each resource within regions of code that access the resource. As described earlier, a process may await a synchronizing boolean condition in the region. Conditional critical regions encourage users to maintain invariants on the shared variables used in the region.

5. Monitors [Hoare 74] facilitate proper access to shared resources by grouping not only the shared variables, but also the statements that access them. These statements are grouped into procedures that are mutually exclusive. A condition variable is associated with each synchronizing condition in the monitor. Processes may wait on these condition variables (similar to the await in conditional critical regions). When a condition holds, it may be explicitly "signalled", thereby awakening a process waiting for that condition (if one exists) and causing it to run immediately. In Mesa [Lampson 80], the only known commercially-used language that supports monitors, a variation of the Hoare monitor appears: signals are only hints that the condition occurred and do not cause the signalled process to run immediately. As a result, on awakening from a wait on a condition variable, the process must retest its condition to be sure it has occurred. Mesa semantics results in simpler and more efficient monitor implementation. Several other variations of Hoare monitors exist, and are discussed in [Howard 76a].

6. Regular expression based specifications [Shaw 79]: By extending regular expressions with features to handle parallelism, some specification methodologies have been developed. Path expressions [Campbell 74] are designed to express synchronization and scheduling constraints. Paths control the ordering of operations and consist of operators for sequencing, selection, repetition and simultaneous execution. No explicit synchronization appears in paths. Flow expressions [Shaw 78] consist of regular expressions enhanced with operators for infinite repetition, shuffle, locks, semaphore-like waits and signals. They have been used to describe a variety of operating system features.

4.2 Kernel Features

In the previous section, synchronization and mutual exclusion were identified as needs in shared memory communication. In Section

3.2.1, signals and waits were suggested as a synchronizing tool for distributed programs. These same signals are also suited for synchronization in shared memory programs: the type of the signal can be used to encode the synchronizing condition that has occurred. As before, each process has its own Signalbox, with its indivisible operations, which is used to store synchronizing information.

The DASH kernel provides locks to facilitate sequential access to shared resources in a shared memory environment. The lock semantics used here are the same as those proposed by Dennis and Van Horn (1966). A lock can be in one of two states: acquired or released. Each lock has associated with it a name, assigned to it when the lock is allocated. The indivisible operations on locks are:

1. lock(L): the invoking process is delayed until the lock L is released, at which point the lock is acquired.
2. unlock(L): releases the lock L.

Since dynamic processes are not supported in DASH, it was decided to allow locks to be static too. This means that locks are allocated at compile time and exist for the entire duration of the concurrent program. (The issues relating to dynamic locks are similar to those for dynamic processes: name allocation, deallocation and transmission; handling erroneous operations on deallocated locks.)

Normal usage involves alternating invocations of lock and unlock operations on each lock. However, successive invocations of unlock are tolerated (though not encouraged) in DASH. This decision makes locks semantically equivalent to binary semaphores, with unlock equivalent to a V operation and lock equivalent to a P.

It is interesting to compare the expressive power of locks and

signals. Locks can be simulated with signals as follows: associate a process with each lock, and a signaltype with each of the lock and unlock operations; let the lock process maintain the state of the lock, and respond to lock and unlock request signals when the state is appropriate; the users of the lock and unlock operations signal their request and await approval from the lock process. However, we are not aware of a straightforward way to precisely model signals between shared memory processes with locks, because of the difficulty in modelling wait on a test operation. Since locks may be modelled with signals, the reason for introducing locks is convenience: they meet a need for which signals were not specifically designed. In deciding to include both locks and (semaphore) waits and signals in flow expressions, Shaw states

> While locks may thus be replaced by signals, we prefer to keep them as separate mechanisms because locking and signalling appear to be fundamental and different requirements in concurrent systems.

We agree with this assessment of the different roles of locking and signalling.

When a lock operation is executed with the lock in the acquired state, the invoking process waits. DASH does not prescribe whether the wait is busy or not. (A busy wait implementation could lead to starvation of the invoking process under some scheduling algorithms.) It may be noted that DASH has two types of waits: an explicit wait primitive, which waits for signals or data, and an implicit wait in the lock operation. The former is intended for longer term scheduling (and hence does not use busy waiting) and the latter for short term scheduling.

4.3 Some DASH Shared Memory Programs

Example 4-1

The general semaphore may be programmed in DASH using locks for mutual exclusion and signal/wait for synchronization. Before discussing this example, some convenient programming structures to be used here and in the following examples, are described. Modules are used as encapsulation tools: they contain data and procedures to manipulate the data. Modules may be shared--DASH mechanisms are used for synchronization. A predefined generic module, named queue, maintains a FIFO list and provides procedures insert (to insert an element in the queue), remove (to remove a specified element from the queue), first (to return the element at the head of the queue), and size (to return the number of elements in the queue). Finally, the following condition procedure (a variant of signalwait) can be used to wait for a signal of a given type (regardless of source).

```
procedure syncwait(type:signaltype) returns(boolean);
var source:process_name; T:signaltype;
begin
    if test(*,type,source,T) then
        begin discard(source,T); return(true) end
    else return(false)
end
```

The semaphore module in Figure 4-1 uses the integers na, np and nv to record the usage history of the P and V operations, a lock semlock for mutual exclusion, and a signal of type unblock for synchronization. Access to the shared data structures only occurs after the lock has been acquired. If procedure P is unable to complete the operation immediately, the invoking process (self) is placed on a queue associated with the semaphore and the lock is released. The process then waits for an unblock signal. If

```
semaphore: module
var na: integer; /* number of P operations attempted */
    np: integer; /* number of P operations completed */
    nv: integer; /* number of V operations completed */
    semlock: lock; unblock: signaltype;
    semq: queue of process_name;

{I_s = np = min(na,nv)}
procedure P;
begin {I_s}
    lock(semlock);
    na := na+1;
    {np = min(na-1,nv)}
    if na>nv then begin
        {np = min(na-1,nv) & na>nv} => {I_s}
        semq.insert(self);
        unlock(semlock); {I_s}
        wait(syncwait,<unblock>)
    end else begin {np = min(na-1,nv) & na≤nv}
        np := np+1;
        {np-1 = min(na-1,nv) & na≤nv} => {I_s}
        unlock(semlock) {I_s}
    end
end P;

procedure V;
var p: process_name;
begin {I_s}
    lock(semlock);
    nv := nv+1;
    {np = min(na,nv-1)}
    if np<na then begin
        {np = min(na,nv-1) & np<na}
        signal(semq.first,unblock);
        semq.remove(semq.first);
        np := np+1
        {np-1 = min(na,nv-1) & np-1<na} => {I_s}
    end else {np = min(na,nv-1) & np≥na} => {I_s};
    unlock(semlock) {I_s}
end V;

/* initialization */
begin {true} na := np := nv := 0; {np = min(na,nv)} end;
```

Figure 4-1: Semaphore module

procedure V is invoked with some processes waiting, the process at the head of the queue is selected and signalled.

Operations on semaphores must maintain the semaphore invariant I_s: np=min(na,nv) (first defined in [Habermann 72]). This means the invariant can be assumed prior to accessing the shared variables in the module, and must be shown to hold following access. The code for the semaphore module has assertions (in braces) inserted between statements. Proof rules for the assignment and if statements only are used in deriving these assertions. No proof rules are used for the locking, queuing and signal operations, since they do not affect the variables appearing in the invariant. The assertions allow us to conclude that if I_s holds prior to (locked) access to the shared variables, it will hold following the access. Though the invariant is maintained, we cannot, however, conclude anything about the fairness of the queuing used; nor can we conclude that the signal sent in the code for V, will indeed wake up a process waiting in procedure P--proof rules for the queues and the DASH signal/wait mechanisms are needed for these.

The proof and invariant used in this example are modelled after those in [Habermann 72, Howard 76b]. Invariants and proofs are used here to illustrate that Howard-style proofs are feasible for DASH modules. Similar proofs are possible for the rest of the examples in this chapter, but are not presented in this document.

Example 4-2

In implementing conditional critical regions (CCR), locks provide the exclusive access to the shared data. For each CCR, a queue is required for processes awaiting synchronizing conditions. The await operation queues the invoking process, and waits for a

wake-up signal using the DASH wait operation outside the critical region. Each time a process completes the CCR, a wake up signal is sent to all processes on the queue of that CCR. Figure 4-2 contains some tools useful in implementing CCRs: procedures to be used on entry, exit and await in CCRs.

The use of these tools is illustrated in programming a reader/writer problem (Section 2.2.2) with CCRs. The solution in Figure 4-3, gives writers priority, i.e., readers may not start when a writer is waiting. The await is implemented with repeated calls to the routine ccr_await, each time the synchronizing condition fails. It may be noted that the solution in Figure 4-3 is as straightforward as the solution in [Brinch Hansen 73, page 113] and that language features for CCRs could easily be translated into equivalent DASH code by a preprocessor.

Example 4-3

The strategy for modelling Mesa monitors is considered before that for Hoare monitors, since Mesa signalling semantics is simpler. A lock is associated with each instance of a Mesa monitor. Prior to executing a monitor entry procedure, the lock is acquired. The lock is released prior to exit of such a procedure. Corresponding to each monitor condition is a record consisting of a queue and a signaltype. Condition waits queue the invoking process and cause it to wait, with the lock released, for a signal of the appropriate type. The Mesa notify operation selects a process from the condition queue and sends it a signal. Likewise, broadcast operations send signals to all processes on the condition queue. (Exit from conditional critical regions can be thought of as having an implicit broadcast on a single condition queue associated with the region.) Some aids to the implementation of Mesa monitors in DASH appear in Figure 4-4.

```
type ccr = record  lk: lock;
                   sig: signaltype;
                   q: queue of process_name
           end;

procedure ccr_enter(c:ccr);
begin lock(c.lk) end;

procedure ccr_exit(var c:ccr);
with c do begin
    while q.size > 0 do
        begin signal(q.first,sig); q.remove(q.first) end;
    unlock(lk)
end ccr_exit;

procedure ccr_await(var c:ccr);
with c do begin
    q.insert(self);
    unlock(lk);
    wait(syncwait,<sig>);
    lock(lk)
end ccr_await;
```

Figure 4-2: DASH tools for conditional critical regions

```
var rw: record /* shared data */
            rr: integer; /* number of running readers */
            aw: integer; /* number of active writers */
            rw: integer; /* number of running writers */
            rwccr: ccr;
        end;
rr := aw := rw := 0;

reader:                          writer:

...                              ...
ccr_enter(rw.rwccr);             ccr_enter(rw.rwccr);
    while (aw≠0) do                  aw := aw+1;
        ccr_await(rw.rwccr);         while (rr≠0) or (rw≠0) do
    rr := rr+1;                          ccr_await(rw.ccr);
ccr_exit(rw.rwccr);                  rw := 1;
<perform read access>            ccr_exit(rw.rwccr);
ccr_enter(rw.rwccr);             <perform write access>
    rr := rr-1;                  ccr_enter(rw.rwccr);
ccr_exit(rw.rwccr);                  rw := 0; aw := aw-1;
...                              ccr_exit(rw.rwccr);
...                              ...
```

Figure 4-3: Reader/writer conditional critical region

```
type cond = record
                q: queue of process_name;
                sig: signaltype;
            end;

procedure condwait(var c:cond;monlock:lock);
with c do begin
    q.insert(self);
    unlock(monlock);
    wait(syncwait,<sig>);
    lock(monlock);
    q.remove(self)
end condwait;

procedure notify(c:cond);
with c do begin
    if q.size > 0 do signal(q.first,sig)
end notify;

procedure broadcast(c:cond);
var p: process_name;
with c do begin
    for all processes p in q do signal(p,sig)
end broadcast;
```

Figure 4-4: Mesa monitor implementation tools

A Mesa monitor for a UNIX pipe [Ritchie 74] written using these aids may be found in Figure 4-5. The pipe resembles a bounded buffer, but has some differences: an arbitrary number of portions may be read or written from the pipe buffer in a single operation--the caller is blocked until the entire operation is completed; also, pipes were originally intended to have only one process access each end. The parameters of the read and write routines consist of the number of characters to be transmitted together with an array of characters for the transmission. The procedure read will, in general, perform many condition waits, depending on the amount written by the writer each time.

```
pipe: module
var pipebuf: array 0..N-1 of char;
    count: 0..N;
    last: 0..N-1;
    nonempty, nonfull: cond;
    plock: lock;

procedure write(n:integer;x:array of char);
var m: 0..N;
begin
    lock(plock);
    while n>0 do begin
        while count = N do condwait(nonfull,plock);
        m := min(N-count,n);
        copy the next m chars of array x into pipebuf
            starting at subscript last;
        count := count+m;
        last := (last+m) mod N;
        n := n-m;
        notify(nonempty)
    end;
    unlock(plock)
end write;

procedure read(n:integer;var x:array of char);
var m: 0..N;
begin
    lock(plock);
    while n>0 do begin
        while count = 0 do condwait(nonempty,plock);
        m := min(count,n);
        copy m chars from pipebuf starting at subscript
            (last-count) mod N, to array x;
        count := count-m;
        n := n-m;
        notify(nonfull)
    end;
    unlock(plock)
end read;

/* initialization */
begin count := 0; last := 0 end;
```

Figure 4-5: Mesa monitor for UNIX pipe

More than one reader and writer process may be accommodated by modifying the solution as follows. To prevent interleaving of data written by writers, a boolean variable (write_busy) and a condition variable (write_possible) are necessary. On obtaining the monitor lock in procedure write, a condition wait on write_possible must occur as long as write_busy is true. On successful completion of the condition wait, write_busy must be set to true. Prior to releasing the lock on exiting procedure write, write_busy should be set to false and a notify on write_possible must be performed. The read procedure should be similarly modified to permit several processes to read from the pipe.

Example 4-4

Hoare monitors differ from Mesa in the semantics of the synchronization operations. A process signalling the condition variable must immediately yield the monitor to a process waiting on that condition, and enter an "urgent" queue. Prior to releasing the monitor lock (on a condition wait or on monitor exit), the monitor must be handed over to a process on the urgent queue (if one exists). Also, Hoare allows priorities to be associated with condition waits.

The procedures in Figure 4-6 have been written to facilitate DASH implementation of Hoare monitors. A priority queue data structure is introduced as an enhancement of the simple queue (described in Example 4-1). The insert operation on a priority queue takes an element together with its (integer) priority and places it appropriately in the queue (low priority values are near the head of the queue). An additional operation (top_pr) exists to return the priority of the element at the head of the queue. Also, a simple queue is associated with each monitor to keep track of processes that

```
type monrec = record
                  urgentq: queue of process_name;
                  urgsig: signaltype;
                  lk: lock
              end;
type condition = record
                     cq: priority_queue of process_name;
                     csig: signaltype;
                 end;

procedure mon_entry(m:monrec);
begin lock(m.lk) end;

procedure mon_exit(m:monrec);
with m do begin
    if urgentq.size > 0 then signal(urgentq.first,urgsig)
        else unlock(lk)
end mon_exit;

procedure condsig(c:condition;var m:monrec);
with c,m do begin
    if cq.size > 0 then begin
        signal(cq.first,csig);
        urgentq.insert(self);
        wait(syncwait,<urgsig>);
        urgentq.remove(self)
    end
end condsig;

procedure condwait(var c:condition;var m:monrec;pr:integer);
with c,m do begin
    cq.insert(self,pr);
    if urgentq.size > 0 then signal(urgentq.first,urgsig)
        else unlock(lk);
    wait(syncwait,<csig>);
    cq.remove(self)
end condwait;
```

Figure 4-6: Hoare monitor implementation tools

```
alarmclock: module
var now: integer;
    alarm_mon: monrec;
    wakeup: condition;

procedure wakeme(duration:integer);
var alarmsetting: integer;
begin
    mon_entry(alarm_mon);
    alarmsetting := now+duration;
    if now < alarmsetting then
        condwait(wakeup,alarm_mon,alarmsetting);
    /* check if other processes need to be woken at same time */
    if now ≥ wakeup.cq.top_pr then condsig(wakeup,alarm_mon);
    mon_exit(alarm_mon)
end wakeme;

procedure tick();
begin
    mon_entry(alarm_mon);
    now := now+1;
    if now ≥ wakeup.cq.top_pr then condsig(wakeup,alarm_mon);
    mon_exit(alarm_mon)
end tick;

/* initialization */
begin now := 0 end;
```

Figure 4-7: Hoare monitor for alarm clock

have given up the monitor on a condition signal. Procedures are provided for monitor entry, exit, condition signals and waits. Intended for general use, these procedures can be simplified in a manner similar to the simplification possible with Hoare's implementation of monitors with semaphores [Hoare 74] (e.g., if there are no condition waits or signals in the monitor, the urgent queue is unnecessary).

The alarm clock problem (Section 2.2.2) is programmed with a Hoare monitor in Figure 4-7. Clients invoke the wakeme procedure, compute their alarmsetting and use it as the priority on the

condition wait. A hardware timer calls procedure tick to update the time and perhaps wake up some processes. In Hoare's solution to this problem [Hoare 74], the process at the head of the condition queue is awakened at every tick of the clock. Our solution only awakens this process when it is time for it to wake up. This is accomplished by permitting the priority of the process at the head of the condition queue to be read in procedure tick.

In the late 70's, the semantics of nested monitor calls was the subject of considerable debate [Lister 77, Parnas 78, Wettstein 78]. The problem arises when a procedure in monitor M_1 invokes a procedure P_2 in monitor M_2. If P_2 executes a condition wait, should exclusion on M_1 be released? If it is released, the monitor invariant in M_1 must be established prior to calling P_2. If the lock on M_1 is not released, besides reducing parallelism, deadlock could occur: the wait in M_2 may depend on a process entering M_2 via M_1 to perform the signal. While staying clear of this controversy, it may be noted that each strategy (with its inadequacies) is implementable in DASH. It is up to the user to choose the semantics.

The examples in this section have demonstrated the use of DASH in shared memory communication. Having modelled semaphores and monitors, we have shown that the facilities in DASH are, in principle, no less powerful than them. Though not presented here, DASH can also be used to implement path expressions, by mimicking the semaphore implementation of path expressions in [Campbell 74]. Based on the applications described above, we may conclude that DASH is well-suited for programming in shared memory environments.

CHAPTER 5. HYBRID COMMUNICATION WITH THE KERNEL

In Chapter 1, hybrid communication was described as consisting of elements of both distributed and shared communication. Some examples were given showing when such communication is useful. The previous chapters have focussed on distributed and shared communication separately and have introduced DASH features suitable for programming them. Here, the two types of communication are brought together and a technique for programming hybrid problems is suggested.

5.1 Hybrid Programming

In programming hybrid applications, it is convenient to have mechanisms that are suitable to both types of communication. We concur with Liskov (1979) that a programming language should give the programmer a realistic model of the underlying architecture. This would enable him to handle problems related to the direct sharing of data and let the architectural expense be evident in the program structure. (One drawback with this philosophy is that once a program is written, it is predisposed towards a particular implementing architecture.) In addition, it is felt that the programmer should easily be able to blend the two types of communication in applications. Having described above some desirable characteristics for hybrid programming, some approaches found in extant languages are examined in this section.

At this stage it is important to clarify the ideas found in a controversial paper by Lauer and Needham (1978). They have suggested that programming language abstractions supporting shared and

distributed views of process interaction are closely related. A "typical" set of primitives for each abstraction is presented and used to program a resource manager. They assert that for the resource manager, the sets of primitives are duals. That is, a manager programmed with one abstraction may be transformed into the other abstraction using simple syntactic transformations. The duality transformation is claimed to preserve the logical structure and the performance of the resource manager. This leads them to conclude that the application does not determine the choice between abstractions, but the available machine architecture does. This conclusion is only valid for conventional strictly shared memory systems because of an underlying assumption in their paper: mention is made of passing large data structures by reference even in the distributed abstraction. Further, the generality of the conclusion may be questioned because the resource manager model restricts the types of communication that may occur. Hence, the duality claim is not relevant to the present discussion. However, their statement about allowing the underlying architecture to influence programming abstractions supports the approach taken in DASH for hybrid architectures--both shared memory and distributed abstractions are used.

A conceptually simple approach for programming hybrid applications is to provide a single logical view (either distributed or shared) and let the implementation map this view into the different (physical) architectures. Such an approach has the advantage of hiding architectural boundaries from the user, at the expense of restricting him from taking advantage of the physical communication architecture. Both CSP [Hoare 78] and Accent [Rashid 81] suggest a uniform distributed view of communication. There are

some difficulties though. Distributed abstractions call for parameter passing by value. When shared memory is present, the implementor would like to take advantage of it to avoid copying. However, copying is necessary because of the value semantics inherent in a distributed view. Further, there are some applications whose process interactions are hard to cast into a purely distributed mold. For example, in programming distributed mutual exclusion [Lamport 74, Ricart 81], each process participating in the mutual exclusion has to respond to queries about its synchronizing variables. Since queries arrive asynchronously, their handling makes the program very complex. If shared communication was permitted, the programmer could create a slave process which responds to such queries by examining the synchronizing state in shared memory, thereby simplifying the master's code.

A uniform shared view of process interaction is also possible. There are indications that Mesa may be extended to provide a somewhat shared view of physically distributed process interaction [Nelson 81]. It is infeasible to provide a view of unrestricted sharing between remote processes because of the expense in implementing single access to remote shared variables. A remote form of sharing is possible, where a number of remote accesses are grouped and are performed by a remote procedure. If such an approach is used for remote sharing, then uniformity requires that it also be used for local sharing--this can be inconvenient. Also, provision should be made for waiting on any of several conditions--a monitor procedure's ability to wait on just one condition is sometimes awkward.

Another approach is to offer the programmer both shared memory and distributed abstractions. This view is apparent in Synchronizing Resources (SR) [Andrews 81], ADA [Ada 79a] and ECLU [Liskov 79].

Each of these languages allows processes to be clustered into "nodes", such that processes within each node may share memory. The mechanisms for communication and synchronization are of a distributed nature. This limits the ease of controlling shared access. For example, if a lock on shared memory is needed, it can be implemented as a special lock process which accepts requests to lock and unlock. Not only is this inconvenient but it is inefficient (because of the additional context switches that take place). The Eden kernel [Eden 81] has an invocation facility for communication between remote processes, as well as semaphores for synchronizing access to shared structures. Though these do well in meeting the remote and local communication needs separately, they were not designed to be used together.

5.2 Kernel Usage in Hybrid Problems

The features of the DASH kernel presented in the last two chapters may be classified as follows. First, there are tools to achieve synchronization between processes: signalbox and the wait operation. Then there are facilities for the transfer of data between address spaces: databuffer and wait. Finally, locks are included as a mechanism for exclusive access to shared resources. In modelling distributed interprocess communication, the synchronization and data transfer features are used. The synchronization and locking features are convenient in implementing shared communication. In hybrid communication, all three features, synchronization, data transfer and locking, are used.

Since signals and waits are used for synchronizing in both distributed and shared interactions, this makes hybrid programming easier. A process can decide to wait for synchronizing information

from other processes, be they local or remote: program an appropriate condition procedure which waits (perhaps nondeterministically) for the arrival of signals from the other processes, and use this condition procedure in a DASH wait. For example, a process could wait for a monitor condition to be signalled or for a signal from a remote process. If the remote signal arrives first, it could be processed, and then the monitor condition wait resumed.

The use of kernel in hybrid applications is illustrated in the examples that appear in the next section.

5.3 Examples of Hybrid Programs

Example 5-1

Ada permits a process to use a delay statement acting as a timeout facility in place of an accept statement in a nondeterministic receive. If the duration specified in the delay statement expires before any of the other alternatives is selected, then the sequence of statements following the delay statement is executed. This feature is easily modelled with the timer module in Figure 5-1.

The timer process in Example 3-5 receives a startup request and sends a timeout signal to the (single) client after a fixed duration has expired. An alarm clock monitor was demonstrated in Example 4-4. This monitor serves several clients and blocks them until the requested duration elapses. The timer module described here, combines the features in those examples: several clients, variable duration, asynchronous start and stop requests. The start procedure in Figure 5-1, enters the request on the priority queue and the stop procedure removes the request from the queue. (It is

```
timer: module
var now: integer;
    tlock: lock;
    timerq: priority_queue of process_name;

procedure start(duration:integer);
begin
    lock(tlock);
    if duration > 0 then timerq.insert(self,now+duration)
        else signal(self,timeout);
    unlock(tlock)
end start;

procedure stop();
var P: process_name; T: signaltype;
begin
    lock(tlock);
    timerq.remove(self);
    unlock(tlock);
    if test(*,timeout,P,T) then discard(P,T)
end stop;

procedure tick();
begin
    lock(tlock);
    now := now+1;
    while now ≥ timerq.top_pr do begin
        signal(timerq.first,timeout);
        timerq.remove(timerq.first)
    end;
    unlock(tlock)
end tick;

/* initialization */
begin now := 0 end;
```

Figure 5-1: Timer module

possible for a timeout to occur after a decision to invoke stop is made, but before the stop invocation acquires the monitor lock. Therefore, a check is made for a timeout signal on completion of stop and, if found, it is discarded.) Each time the clock ticks, all processes whose durations have elapsed are signalled.

```
entrylist = record
                nentries: integer;
                list: array [1..nentries] of entry;
                selected: [1..nentries];
                duration: integer
            end;

procedure adareceive(var elist:entrylist);

    procedure entrywait(var elist:entrylist) returns(boolean);
    var i: integer; t: signaltype;
    begin
        <procedure entrywait same as in Figure 3-7 (page 60)>
    end entrywait;

begin
    timer.start(elist.duration);
    wait(entrywait,<elist>);
    with elist.list[elist.selected] do
        if stype ≠ timeout then begin
            signal(caller,ok);
            datawait(addr);
            timer.stop()
        end
end adareceive;
```

Figure 5-2: Modelling of Ada delay statement

The procedure adareceive in Figure 5-2 is a modification of adareceive in Figure 3-7 to permit delay statements. The entrylist contains an entry record for the delay statement (just as it does for accept statements) with the expected signaltype set to timeout. A duration field is added to the entrylist record. Prior to invoking adareceive, the entrylist must be initialized as before, and the value of the delay expression stored in the duration field. Adareceive starts the timer prior to performing a DASH wait. If the wait completes before the timer duration expires, the timer is stopped.

A similar strategy may be used to model the timeout on entry call. In this case, since the caller has already sent a signal to the called process indicating its readiness to invoke an entry, a recovery protocol is necessary when the caller times out.

In this example, a timer "monitor" has been used alongside logically distributed communication. The wait in procedure adareceive combines waiting for signals from a distributed process as well as from a shared memory timer process.

Example 5-2

The Ricart-Agrawala distributed mutual exclusion algorithm [Ricart 81] uses both shared memory and distributed communication. Semaphores are used to control access to (locally) shared variables. Operations for sending and receiving messages are used, whose semantics are not described in the paper. The algorithm written using DASH appears in Figure 5-3.

The following is a brief description of the algorithm. N user processes, each on a distributed "node", cooperate in sequentializing access to a resource. When a user wishes to access the resource, it sends a request message to each of the other users. On receiving replies from all other users, it is free to use the resource. Each user has a unique index and maintains a sequence number associated with resource accesses. Prior to vying for the resource, a sequence number is chosen, which is one greater than the largest sequence number observed thus far. A request to another user contains the ordered pair <chosen sequence number, user index> which is used to sequence access to the resource. Each node has a process dedicated to answering requests from other nodes. The answering process replies immediately if its user is currently not vying for the

```
/* variables shared by user and its answering process */
const N, /* number of nodes participating in mutual exclusion */
      up,ap: array [1..N] of process_name, /* of the user and
                          answering processes at each node */
      me, /* this node's index */
      request,reply,ok: signaltype;
var Our_Sequence_Number, Outstanding_Reply_Count,
    Highest_Sequence_Number: integer; /* initialized to 0 */
    Requesting_Critical_Section: boolean; /* initially false */
    Reply_Deferred: array [1..N] of boolean; /* initially false */
    Shared_vars: lock;

user process:

var j: integer;

procedure count_replies() returns(boolean);
var p: process_name; t: signaltype;
begin
    while test(*,reply,p,t) do begin
        discard(p,t);
        Outstanding_Reply_Count := Outstanding_Reply_Count-1;
        if Outstanding_Reply_Count = 0 then return(true)
    end;
    return(false)
end count_replies;

begin
    ...
    ...
    /* prepare for critical section entry */
    lock(Shared_vars); /* choose sequence number */
        Requesting_Critical_Section := true;
        Our_Sequence_Number := Highest_Sequence_Number + 1;
    unlock(Shared_vars);
    Outstanding_Reply_Count := N-1;
    for j := 1 to N do /* send requests to other users */
        if j ≠ me then begin
            signal(ap[j],request);
            wait(syncwait,<ok>);
            transfer(ap[j],<Our_Sequence_Number,me>,2)
        end;

    wait(count_replies,<>); /* await replies from other users */
```

Figure 5-3: Ricart-Agrawala distributed mutual exclusion algorithm

```
    <perform critical section processing>

    Requesting_Critical_Section := false;
    lock(Shared_vars);
    unlock(Shared_vars);
    for j := 1 to N do
        if Reply_Deferred[j] then begin
            Reply_Deferred[j] := false;
            signal(up[j],reply)
        end;
    ...
    ...
end user;

answering process:

var requestor: process_name;
    Req_Index,Req_Seq: integer;

procedure requestwait(var source:process_name) returns(boolean);
var T:signaltype;
begin
    if test(*,request,source,T) then
        begin discard(source,T); return(true) end
    else return(false)
end requestwait;

loop
    /* receive next request */
    wait(requestwait,<requestor>);
    signal(requestor,ok);
    datawait(<Req_Seq,Req_Index>);

    Highest_Sequence_Number :=
        max(Highest_Sequence_Number,Req_Seq);
    lock(Shared_vars);
        if Requesting_Critical_Section and
            ((Req_Seq>Our_Sequence_Number) or
                (Req_Seq=Our_Sequence_Number and Req_Index>me))
        then Reply_Deferred[j] := true
        else signal(requestor,reply);
    unlock(Shared_vars);
end answering process;
```

Figure 5-3, concluded

resource, or if the requesting sequencing ordered pair is less than its own sequencing ordered pair. Otherwise, it defers the request until its (local) user has completed access to the resource, at which time the user replies to all deferred requests. The original published version of the Ricart-Agrawala algorithm contains a race condition which could lead to deadlock. This problem has been detected and corrected by us, and by the authors independently. This correction is reflected in the DASH description in Figure 5-3.

The sending of a request message is modelled in DASH as a request signal, an ok response signal and transfer of the sequencing ordered pair to the answering process, whereas the reply message consists of a simple signal. The decision made in DASH to limit the information carried by a signal, forces the handshaking necessary to convey the request parameter. This, unfortunately, increases the number of transmissions per resource usage from 2*(N-1) to 4*(N-1). The 2*(N-1) complexity measure found in the paper does not include transmissions to ensure receiver readiness, which would be required in general (the 4*(N-1) measure includes such costs). Including such transmissions would worsen the complexity, though probably not by a factor of two.

The DASH wait operation, when used with the condition procedure count_replies, provides a convenient way to wait till all the replies are in. Each time count_replies is invoked, it checks for reply signals and decrements the count of outstanding replies by the number found. When this count reaches zero, it returns true, thereby terminating the wait. The processing on completing the critical section contains a lock statement followed immediately by an unlock. This serves to ensure that the answering process is not in the middle of recording a deferral, when the user begins acting on

deferrals. (This point was overlooked in the original published algorithm, resulting in a race condition.)

This example suggests the use of DASH in describing hybrid algorithms. Communication is often an important part of such algorithms and well defined communication primitives simplify and clarify descriptions.

Example 5-3

Asynchronous communication can be conveniently programmed using a hybrid approach. Here, the user process desires to communicate with a remote process but would prefer not to be concerned with details of remote access. So, it uses a local communication process (cp) for such matters and it transfers message buffers to/from the cp using shared memory. Such a scheme can be used to implement asynchronous communication primitives of the type found in PLITS. Also, a file server process could transfer portions of files from the storage medium to message buffers and hand these to its cp for transfer to (remote) clients.

An implementation strategy for asynchronous output is sketched in Figure 5-4. The memory shared by the user and cp processes contains a buffer pool managed by the buffer allocator monitor in Figure 5-5. This monitor is modelled after the one in [Hoare 74], and is programmed using the tools in Figure 4-4. The user requests an empty buffer from the buffer allocator monitor, fills it and passes the address of the buffer to cp using a single-buffer monitor (discussed below). Process cp accepts the message buffer address and queues it. It carries out interactions with remote processes to relay messages to them. It is desirable to program cp such that it is able to nondeterministically respond to stimuli from the user via

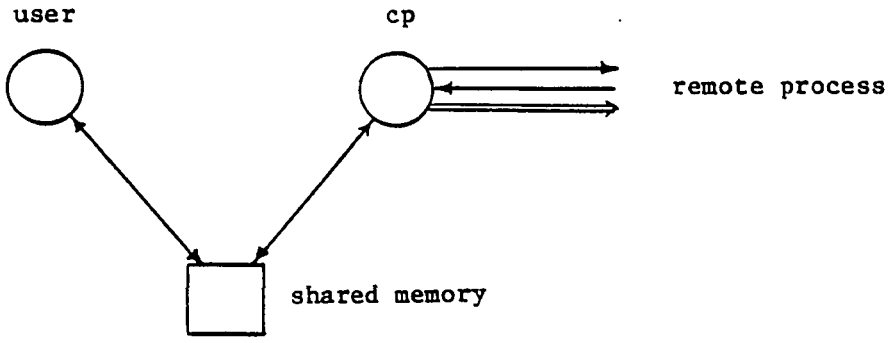


Figure 5-4: Interactions in Asynchronous Communication

```
type msg = record
               source,dest: process_name;
               type: signaltype;
               size: integer;
               body: <message contents>
           end;

allocator: module
var freepool: queue of ^msg;
    alock: lock; avail: condition;

procedure acquire(var m:^msg);
begin
    lock(alock);
    while freepool.size=0 do condwait(avail,alock);
    m := freepool.first;
    freepool.remove(m);
    unlock(alock)
end acquire;

procedure release(m:^msg);
begin
    lock(alock);
    freepool.insert(m);
    notify(avail);
    unlock(alock)
end release;

/* initialization */
begin <freepool := all buffer addresses> end;
```

Figure 5-5: Buffer allocator monitor

the single-buffer monitor, as well as from remote processes. This differentiates cp from the answering process in the previous example.

The single buffer module (Figure 5-6) behaves basically like a bounded buffer monitor with a single slot. The module differs from an equivalent monitor in that no condition waits are used. The requirement that cp be able to nondeterministically respond to the user and remote processes, makes waiting on conditions within a monitor procedure undesirable. Since condition waits in a bounded buffer monitor occur at the start of the monitor procedures, their effort can be simulated by causing these procedures to return with a value 'false' on failing the condition and with a value 'true' on successfully completing the procedure. In addition, when a process fails a condition, its name is stored in a queue associated with the condition; when the condition holds, all the processes on the queue are signalled. When a module procedure returns a value false, the invoking process can wait until it is thus signalled and retry the procedure. The user process in Figure 5-7 interfaces with the single buffer module in this manner. The process, while waiting for the condition signal, could be open to servicing remote signals. If only one process accesses each monitor procedure (as is true for this example), a simpler solution without queues is possible: introduce a module procedure to indicate whether the single buffer is empty or not and then decide to execute the fetch or store procedure. The solution presented has the advantage that it can be generalized more easily than the simple solution.

The communication process (Figure 5-8) maintains queues of message buffers. For simplicity of description, a separate queue is associated with each <destination,type> it services. A message received from the user is placed on the appropriate queue and the

```
singlebuffer: module
const notempty,notfull: signaltype;
var slot: ^msg;
    full: boolean;
    nonempty,nonfull: queue of process_name;
    slock: lock;

procedure store(m:^msg) returns(boolean);
begin
    lock(slock);
    if full then begin
        nonfull.insert(self);
        unlock(slock);
        return(false)
    end;
    nonfull.remove(self);
    slot := m;
    full := true;
    for all processes p in nonempty do signal(p,notempty);
    unlock(slock);
    return(true)
end store;

procedure fetch(var m:^msg) returns(boolean);
var p: process_name;
begin
    lock(slock);
    if not full then begin
        nonempty.insert(self);
        unlock(slock);
        return(false)
    end;
    nonempty.remove(self);
    m := slot;
    full := false;
    for all processes p in nonfull do signal(p,notfull);
    unlock(slock);
    return(true)
end fetch;

/* initialization */
begin full := false end;
```

Figure 5-6: Single-buffer module

```
user process:

const notfull: signaltype;
var m: ^msg;

begin
    ...
    ...
    allocator.acquire(m);
    <fill m>
    while not singlebuffer.store(m) do wait(syncwait,<notfull>);
    ...
    ...
end;
```

Figure 5-7: Asynchronous communication: user process

remote process is signalled, if this is the only message on the queue. Servicing signals from remote processes involves retrieving and dispatching a message buffer, and signalling the remote process if more messages remain on the queue. In the main loop, the fetch procedure is first tried. If it fails, cp is assured of being on the nonempty signal "mailing list" and condition procedure workwait may be used to await further work. Workwait on receiving the nonempty condition signal attempts to invoke fetch; if the invocation terminates successfully, the wait terminates.

The solution in this example generalizes easily for the case where many user processes rely on a bounded buffer with more than one slot to pass message buffers to several communication processes. The strategy to signal all cps on the nonempty queue when the buffer becomes nonempty, ensures that a cp on the queue that is not preoccupied with a remote signal will complete its fetch soon. The price for efficient monitor usage is paid by some cps that attempt fetch in vain (because of the broadcasting of signals).

```
communication process (cp):

const notempty: signaltype;
var m: ^msg;
    P: process_name;
    T: signaltype;
    msgq: array [process_name][signaltype] of queue of ^msg;

procedure workwait() returns(boolean);
begin
    if test(*,*,P,T) then begin
        discard(P,T);
        if T = notempty then /* condition signal */
            return(singlebuffer.fetch(m))
        else /* remote signal */
            return(true)
end workwait;

loop
    /* check if user has any work for me */
    if singlebuffer.fetch(m) then with m^ do begin
        if msgq[dest,type].size=0 then signal(dest,type);
        msgq[dest,type].insert(m)
    end
    else begin
        wait(workwait,<>);
        if T = notempty then /* condition signal */
            with m^ do begin
                if msgq[dest,type].size=0 then signal(dest,type);
                msgq[dest,type].insert(m)
            end
        else begin /*remote signal */
            m := msgq[P,T].first;
            msgq[P,T].remove(m);
            transfer(P,m^.body,m^.size);
            allocator.release(m);
            if msgq[P,T].size > 0 then signal(P,T)
        end
    end
end cp;
```

Figure 5-8: Asynchronous communication: communication process

The technique used to avoid condition waits only works if the condition is tested at the beginning of a monitor procedure. This restriction may be removed by introducing semicoroutines to save the state of a partially completed monitor procedure. (The state needing to be saved consists of the local variables of the monitor procedure). On failing a condition, a detach(false) should be executed in place of a return(false). The author is not aware of any naturally arising situations where this generality is needed--an artificial one can be contrived easily.

Should different conditions be tested within a module procedure, then the return on failure should be modified to let the caller know which condition failed. This will allow the calling process to wait for the signal associated with that condition.

One of the advantages of using shared memory communication between the user and cp processes is that copying may be avoided. However, there are other aspects of asynchronous output that may necessitate copying. If the user is constantly updating some variables and periodically sending them asynchronously to a remote process, then copying is required in passing them to cp.

In exploring DASH usage for asynchronous communication, an adaptation of a monitor has been introduced. This permits condition waits to be treated the same as remote signal waits, allowing them to be used together in a DASH wait.

CHAPTER 6. IMPLEMENTATION

The DASH kernel was implemented on a uniprocessor that provides logical distribution. The first section of this chapter presents this implementation. The following section addresses the major issues in implementing DASH on a network.

6.1 A Uniprocessor Implementation

The DASH implementation was performed on a Caldata 135 minicomputer running the UNIX [Ritchie 74] operating system. It was later moved to a DEC VAX 11/750 running Berkeley UNIX. UNIX was chosen primarily because it supports a distributed view of process communication. However, sharing of memory between processes is hard to accomplish in UNIX. We were willing to tolerate this inconvenience, in order to have a pleasant environment in which to gain experience with distributed programming.

6.1.1 Characteristics of Unix Implemented DASH

UNIX-DASH [Rao 81] consists of a number of data structures and C routines which must be linked to each user process at load time. These routines are implemented on top of the standard UNIX system calls. UNIX allows process interaction through pipes, which were introduced in Example 4-3. Both signals and data transfers are placed in messages and sent over pipes. Shared memory is simulated through shared files. UNIX-DASH does not attempt to conceal this from the user--he must use UNIX file i/o facilities explicitly, in order to read/write shared memory.

The signalbox at each process is implemented as a two

dimensional array of signals allocated at compile time, indexed by process names and signal types. Process names are assigned by the user and differ from those used by UNIX. While this implementation strategy becomes expensive when the number of processes and signal types is large, it has proven adequate for the size of the problems that can be handled on our minicomputer. When a signal arrives at a user process, it is assigned an integer "time stamp", and this is stored in the signalbox array. The test operation first updates the signalbox with all signals waiting on the input pipe. Then it checks for the presence of the argument signal. When a wild card is used, the oldest matching signal based on the time stamp is chosen and returned to the caller--this assures fairness. This strategy is also used in the CSP implementation of DASH found in Appendix B.

The semantics of the DASH wait operation requires the condition procedure to be evaluated each time an external stimulus (i.e., signal or data) is received. The UNIX-DASH implementation of wait reads messages on the input pipe (and appropriately stores incoming signals and data), as long as the condition procedure returns false. When it attempts a read on an empty pipe, the UNIX read system call blocks the invoking process until the read can be satisfied. When the read completes, the condition procedure is tried once again. The argument list parameter in the wait operation is assumed to be the address of a data structure which the user has designed to transmit parameters to the condition procedure. It is the responsibility of the user to develop structures suitable for each condition procedure.

UNIX provides no explicit mechanism for locking (since it does not allow users to share memory). However, the lock and unlock operations may be simulated by suitable use of creation and deletion

of a known file [Ritchie 78]. An attempt to create a specific protected file when it already exists, returns an error. The lock operation consists of busy waiting till this creation succeeds. Unlock simply deletes the file.

Before pipes may be used for communication in a multiprocess UNIX program, they need to be connected appropriately. This "plumbing" task is made difficult by the UNIX restriction that two processes may access a pipe only if the pipe was created by a common ancestor; pipe descriptors are acquired only through inheritance and not through passing between processes. Also, there is a limit (16) on the number of pipe descriptors a process may use at any one time. For other than the simplest communication graphs, the plumbing task is inordinately complex. A utility program named Tree has been developed to facilitate the set-up of programs consisting of processes with arbitrary communication graphs. It creates a full ternary[1] tree of processes of sufficient depth so that the number of leaves is at least as many as the number of process images. The leaf processes execute these process images, and the internal processes serve only to route messages to leaves. Each user (leaf) process is supplied a pipe descriptor with which it may read messages from its parent, and a pipe descriptor to direct messages to the root process (which forwards them down the tree). The Tree program allows any two leaf processes to communicate in time $O(\log_3 N)$. This is considerably more expensive than the constant time which should be possible if UNIX were more hospitable.

---

[1]"Bushier" trees would further decrease the height of the tree, but would exceed the process file descriptor limitations.

When a user process is started up by Tree, it must initialize the kernel using a routine which has been provided. Once the process is running, it may need to receive input from the terminal. The Tree program provides a rudimentary facility for commands entered at the terminal to be directed to user processes.

The DASH implementation was performed almost entirely on top of UNIX. However, two minor changes to UNIX were necessary: the ability to detect an empty pipe was added; and a "bug" which occasionally permitted interleaving of data written on a pipe, was corrected. Overall, UNIX and C together provided a good environment for this implementation. The weak typing in C was suitable for the low level aspects. Also, it allowed transmission of arbitrary argument lists to condition procedures via the wait operation. UNIX, however, could benefit from the following:

- facilities for shared memory
- a nondeterministic wait on several pipes
- a simpler means for arbitrary process interconnection
- tools for debugging multiprocess programs
- better facilities for the compile-time initialization of variables and structures in C

6.1.2 Uses of Implemented Kernel

UNIX-DASH has been used in implementing features of various languages as well as in the solution of distributed, shared and hybrid problems. Among the languages whose features have been modelled are CSP, CP and PLITS. In each case, a set of routines has been developed which implements the communication mechanisms in the language, and may be used in applications.

```
#include        "proctype.h"
#define BBNUM   4
#define BBSIZE  2
#define WILD    -1
int     inp,outp;
int     match[2];
reqselect()
{
        if((inp-outp<BBNUM) && test(WILD,INSERT,match)) {
                discard(match[0],match[1]);
                return(TRUE);
        }
        if((inp>outp) && test(WILD,REMOVE,match)) {
                discard(match[0],match[1]);
                return(TRUE);
        }
        return(FALSE);
}
main(argc,argv) /* bounded buffer process */
int     argc;
char    **argv;
{
        int     myid,inpipe,gcrw; /* implementation related */
        int     buffer[BBNUM][BBSIZE];

        /* initialize DASH kernel */
        gcrw = 5;
        inpipe = gcrw+1;
        myid = atoi(argv[1]);
        initdash(inpipe,gcrw,myid);

        inp = 0; outp = 0;
        for(;;) {
            wait(reqselect,0);
            switch(match[1]) {
            case INSERT: signal(match[0],OK);
                         datawait(buffer[inp%BBNUM]);
                         inp++;
                         break;
            case REMOVE: transfer(match[0],buffer[outp%BBNUM],
                                                    BBSIZE);
                         outp++;
            }
        }
}
```

Figure 6-1: UNIX-DASH bounded buffer process

```
#define NONFULL         0
#define NONEMPTY        1
#define BBNUM   4

struct  monblk  mondes;
int     inp,outp,databuf[BBNUM][BBSIZE]; /* monitor variables */
int     cndnvar[2];     /* condition variables */
int     fdes;
store(data)
int     *data;
{
        monentry(&mondes,&fdes);
        while(cndntest((inp-outp)==BBNUM,NONFULL,&mondes))
                {monexit(&mondes,fdes); wait(syncwait,NONFULL);
                 monentry(&mondes,&fdes);}
        copybuf(data,databuf[inp%BBNUM]);
        inp++;
        broadcast(NONEMPTY,&mondes);
        monexit(&mondes,fdes);
}
fetch(data)
int     *data;
{
        monentry(&mondes,&fdes);
        while(cndntest(inp==outp,NONEMPTY,&mondes))
                {monexit(&mondes,fdes); wait(syncwait,NONEMPTY);
                 monentry(&mondes,&fdes);}
        copybuf(databuf[outp%BBNUM],data);
        outp++;
        broadcast(NONFULL,&mondes);
        monexit(&mondes,fdes);
}
initbbuf()
/* initializes bounded buffer module */
{
        inp = 0; outp = 0;
        cndnvar[0] = 0; cndnvar[1] = 0;
        monwrite(&mondes);
}
initbdes(monname,condbase)
char *monname; int condbase;
/* initializes bounded buffer monitor descriptor */
{       ...
        ...
}
```

Figure 6-2: UNIX-DASH bounded buffer monitor

To illustrate the flavor of UNIX-DASH, parts of solutions to two problems are included in this section. Figure 6-1 contains a bounded buffer process (similar to the one in Figure 3-2). The UNIX-DASH solution differs from the one in Figure 3-2 in some minor respects: code for initializing the kernel is included; and the parameters of the test operation are organized somewhat differently.

A bounded buffer monitor appears in Figure 6-2. It has been written with tools similar to those found in Figure 4-4, and its logic is similar to the single buffer module in Figure 5-6. A monitor descriptor is associated with the monitor: it contains the location of the shared variable block, as well as the condition queues (represented as bitmaps). The procedure for entering the monitor acquires the monitor lock and reads in the shared variables from the shared file; the monitor exit procedure is similar. The procedure cndntest tests if the boolean expression is true in order to decide whether to enter or remove the invoker from the appropriate condition queue. (The tools here are somewhat simpler than those in Figure 4-4, since they are also intended for hybrid applications.) Routines also exist for initializing the monitor and setting up its descriptor.

UNIX-DASH has also been used for instruction in operating systems: at a graduate course at the University of Washington and at an intensive summer course at the University of California, Santa Cruz. One class project used UNIX-DASH for implementing a CSP compiler [Yenbut 81]: a preprocessor was written to translate CSP into C with a run-time package that uses UNIX-DASH for communication.

Finally, programs implemented in UNIX-DASH have been instrumented and the measurements have been used to arrive at performance estimates for certain architectures (Chapter 7).

6.2 Network Implementation Considerations

Network environments need to be treated differently from uniprocessor environments because of two main characteristics: unreliable communication and processor failure. The communication system typically provides a reasonable assurance of faithful message transmission, but makes no guarantees. It is the task of communication protocols to achieve communication that is sufficiently reliable for the user's needs. Also, in a distributed system, a processor crashing leaves part of the system operational. When a crash occurs in a uniprocessor system, the usual remedy is to restart the entire system. It is desirable in a distributed system to avoid a system-wide reset, each time a processor crashes. In this section, we examine how a network implementation of DASH can provide the highly reliable communication needed. This is followed by a discussion of coping with processor failure.

6.2.1 Communication Failure

A recent thesis [Nelson 81] considers how remote procedure calls may be implemented with semantics similar to those for local procedure calls. It contains details of an implementation strategy for reliable transmission of remote procedure calls and for the graceful handling of crashes. The reliable transport protocol for DASH signals (Figure 6-3) is adapted from Nelson's implementation.

The protocol in Figure 6-3 uses low-level network packet transport primitives to implement reliable communication: it can tolerate lost, mutilated and duplicated packets, but not prolonged communication outages or processor crashes. Synchronous semantics are chosen for signal transfer: the process invoking the DASH signal

Code for sending a signal:

```
seq_num := seq_num+1;
<determine destination node; store source, destination and
 type fields in packet pkt>
pkt.seq_num := seq_num;
for i := 1 to max_tries do begin
    Net_transfer(pkt);
    <wait (on waiting list) for ack;
      timeout after retransmit_interval>
    if <ack received> then goto done
end;
<raise failure exception>
done:
```

Code invoked on reception of a DASH packet pkt:

```
if <packet damaged> then <discard packet and skip rest of code>
if <packet contains signal> then begin
    if pkt.seq_num < largest_seq[pkt.dest] then
        <disregard (old) signal and skip rest of code>
    else if pkt.seq_num = largest_seq[pkt.dest] then
        Net_transfer(ack) /* resend ack */
    else begin /* new signal packet */
        <update signalbox of pkt.dest and awaken it if it is on
            waiting list>
        largest_seq[dest] := pkt.seq_num;
        <construct ack packet>
        ack.seq_num := pkt.seq_num;
        Net_transfer(ack)
    end
end
else if <packet contains ack> then
    if <pkt.dest on waiting list> and
        pkt.seq_num = pkt.dest.seq_num then
            <awaken pkt.dest>
    else <disregard ack>
```

Figure 6-3: Reliable Signal Protocol

operation waits until the transfer completes but not until the receiving process is willing to honor the signal. This simplifies the implementation and use of the signal operation, because synchronous exceptions are easier to report and handle than asynchronous ones. The signalling process must be delayed until the signal packet has been successfully acknowledged--a price which is small on local networks but could be significant on other networks; in which case the protocol can be modified.

Each process has a packet buffer permanently allocated for use in sending signals. In addition to source, destination and type fields, this packet also contains a sequence number. A monotonically increasing sequence number is maintained in each process, which is incremented each time a signal operation is invoked. Once a signal is sent, the signalling process enters a waiting list until an acknowledgement (ack) packet is received. A timeout occurs if the ack does not arrive within retransmit_interval and the packet is resent. After max_tries retries have failed, a communication breakdown is assumed to have occurred and a failure exception condition is raised. The degree of reliability achieved by the protocol is a function of the network physical transmission reliability and the parameter max_tries.

When a packet arrives at a node, an interrupt-level process is activated. This process has access to a data structure containing the largest sequence number received from each of the other processes, and to the waiting list of processes. When the received signal packet sequence number is less than the largest sequence number for the source, the packet is assumed to be a duplicate and is discarded. If the sequence numbers agree, then the packet is a repetition of the last signal because the ack was lost, and the ack

containing the sequence number is resent. If the packet contains a new signal, it is recorded in the signalbox, an ack is sent, and if the recipient process is on the waiting list it is awakened. When an ack with the correct sequence number is received, if the signalling process is waiting, it is resumed.

In the protocol described above, only signal transfers are mentioned. Since data transfers can be implemented just as if they were signals (using the same sequence number), the details have been omitted in Figure 6-3.

6.2.2 Processor Failure

When a processor crashes, it may leave data structures in an inconsistent state. Also, much of the processor state is lost, making it difficult to resume its processes at the point of the crash (in order to remedy inconsistencies). For example, consider a program which transfers x dollars from account A to account B. (A and B are assumed to reside on storage that can survive crashes.) It performs the statements

```
A := A-x;  B := B+x;
```

If the processor crashes after the first statement is executed but before the second, then an inconsistency in the account balances results. This problem is similar to the critical section problem (caused by uncontrolled parallel access to shared data). The critical section problem may be solved by ensuring that access to the critical section is atomic (or indivisible) in each process. This guarantees that an invariant on the shared variables holds when no process is in the critical section. The problem of data inconsistency due to crashes can be handled in a similar fashion:

extend the definition of atomicity, so that in the presence of crashes, atomic operations either complete or appear not to have started. This ensures that the data invariant holds even when crashes occur.

Tools for programming in the presence of crashes are developed and explained in [Lampson 81]. The concept of stable storage (i.e., storage that survives crashes) is introduced along with atomic operations for accessing it. An implementation of stable storage using normal (fault-prone) disk storage is given. The process state can be saved in stable storage with a Save operation; on recovering from a crash, the process is restarted in the last saved state. A Reset operation alters the saved state so that the process returns to its idle state after a crash. Often it is necessary to ensure that transactions consisting of many atomic steps are themselves atomic. If the sequence of atomic steps is restartable (i.e., repetition following crashes causes no ill effects) then the transaction as written below is atomic:

```
Save; <sequence of atomic steps>; Reset;
```

The account balance example may be made into an atomic transaction as follows:

```
a := Stable_read(A); b := Stable_read(B);
a := a-x; b := b+x;
Save; /* saved state includes a and b */
Stable_write(A,a); Stable_write(B,b);
Reset;
```

If a crash occurs before the Save, the transaction aborts (is not performed at all). If it happens after the Reset, the transaction has committed (completed). A crash occurring between the Save and the Reset, causes recovery to start immediately after Save, and the

transaction commits (perhaps after repeating the Stable_writes). Thus once Save has been executed, the transaction is destined to commit. The two step commit protocol (Sections 2.2.3, 3.3.1) is an implementation of this technique for distributed transactions; messages are used for communication between distributed participants.

In order to implement the two step commit protocol with DASH, the semantics of communication in crash situations needs to be defined. The signalbox and data buffer are in volatile store and their contents are lost in a crash. Signals and data sent to crashed processes are lost. We believe that these semantics are simple enough to be easily implemented in networks. It is possible for a process sending a signal to crash after the signal has been stored in the destination's signalbox but before the destination takes note of it. Such "orphanned" signals should be taken into account in any protocol. A careful reader may have observed that the reliable signal protocol in Figure 6-3 can leave an orphan: it is possible that a signal arrives at a destination after a sender has raised a failure exception.

It is interesting to re-examine the code in Figure 3-5 in view of our understanding of atomic transactions to see how it achieves its crash resistance objective. The details of the interaction to send updates to the slaves do not appear in the figure; the master simply queries the slaves to ensure their health. If all the slaves answer positively, the master saves its state on stable storage and the transaction will eventually commit. The master then informs the slaves of its decision and they update their (stable) data bases. The character string argument in the Save_state operation is used to direct recovery efforts. (The recovery strategies mentioned in [Baer 81] can be programmed with DASH.)

The purpose of the preceding paragraphs is to explore the use of DASH in a crash-prone environment. While not specifically designed for crashes, it seems to be possible to use DASH if it is enhanced with stable storage facilities. A concrete implementation and the programming of more applications are necessary before more definitive statements can be made.

CHAPTER 7. DASH PERFORMANCE: DISTRIBUTED VS. SHARED

7.1 Overview

The implementation in the previous chapter may be used to compare the performance of distributed and shared memory versions of concurrent programs that use DASH. The performance in the distributed version is a function of the end-to-end transmission delay. Our goal is to estimate the range of delay over which the performance in the two architectures is comparable.

This study restricts itself to programs whose component processes execute on separate processors. Processes communicate using the DASH kernel. The program execution time depends on the process compute times, the kernel time and the communication time. Since there is only one process per processor, times for process scheduling and swapping are no longer pertinent. In the shared memory architecture, processes communicate via memory shared between them, using locks to sequence access to shared locations. The only additional cost in accessing shared memory (over local memory) is assumed to be the cost of any locking and unlocking operations performed[1]. In the distributed architecture, all process interaction occurs through signals and data transfers. These in turn are implemented using some reliable transport mechanism. The effect on program performance of varying the delay time introduced by this mechanism is the subject of interest. This end-to-end delay time is influenced by the underlying network characteristics as well as the

---

[1]The cost due to memory contention is ignored in this study

cost of the protocol necessary to achieve reliable transport. The end-to-end cost of transporting a message from a source process to a destination process is assumed to be the same regardless of source and destination. This assumption is fairly realistic in local-area networks.

Performance study of a specific problem requires time estimates for computation and kernel overhead. These are readily obtained by instrumenting a UNIX-DASH implementation of the problem. Using this data, a simulation model of the problem in the SIMULA language [Franta 77] was constructed, allowing the delay to be varied. Performance can then be expressed as a function of communication delay. Similarly, in the shared memory version of the problem, the implementation is instrumented for computation times, and access to shared memory is simulated.

If a problem is not communication intensive, it is reasonable to expect that the performance of the distributed version is constant for low delay times. This is because delay may be overlapped with computation. However, for larger delay times, the effect of the delay predominates and performance should degrade monotonically with increasing delay. Based on the observations of Lauer and Needham (1978), one would expect that if there is not much copying of data in the distributed case, the performance at low delay times should be in the neighborhood of that for shared memory. This intuition is graphically illustrated in Figure 7-1.

7.2 Problem Instrumentation

The UNIX implementation of DASH (Section 6.1) was used to program solutions to a problem in order to obtain time estimates for the computational segments of the problem and the kernel. DASH

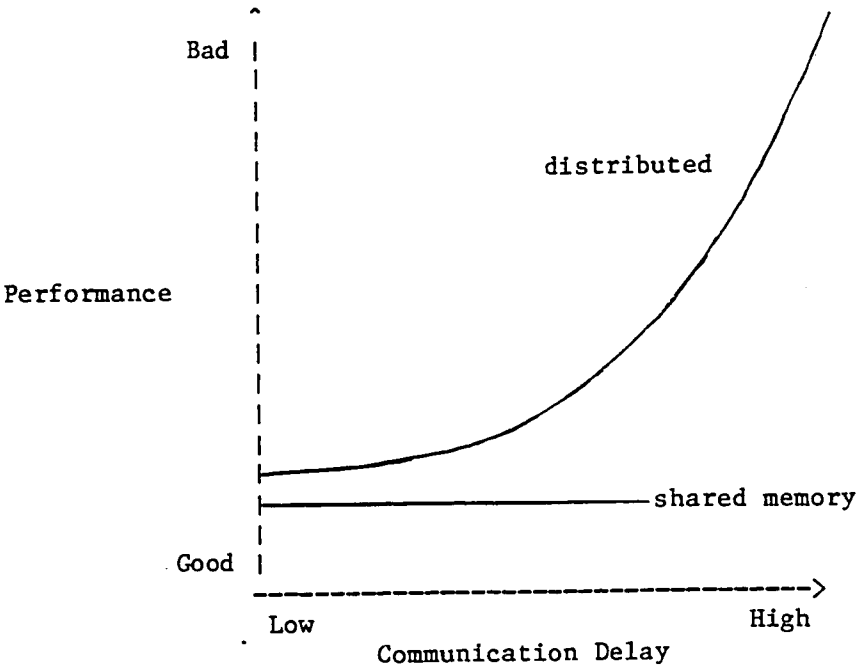


Figure 7-1: Expected Problem Performance

operations typically package signals and data before using the UNIX facility which reliably transports messages between processes. The segments to be measured (which include both problem and kernel computation) are thus delimited by requests to this reliable transport interface to send or receive messages. The problem chosen for study is first implemented using UNIX-DASH. Then in each process, the computation segments are identified. Execution times for computation segments are measured using methods described below. Estimation of segment time by counting instructions is also possible; however, the VAX cache and translation buffer may have uncertain effects on instruction execution times.

UNIX provides rather unsophisticated tools to measure execution times. At each tick of the 60 Hz hardware clock, a clock counter associated with the currently executing process is incremented. System calls exist to examine this counter during program execution. The durations of the computation segments for the problems considered are of the order of 1 millisecond on a VAX 11/750. Hence, repeated execution of segments is necessary in order to use the hardware clock. The process clock counter is examined prior to and after the segment repetition, and the loop overhead is accounted for. All measurements were performed with no other users on the system.

The accuracy of the measurement technique may be questioned for the following reasons. The process execution time (in 1/60 second units) is estimated by the number of clock ticks during execution. Other activity in the system could adversely effect this estimate. It is hoped that using a large number of repetitions in a lightly loaded system would minimize this effect. Also, repetition of small segments may be expected to yield lower values of compute

times because the cache would soon be loaded with the contents of the memory locations referenced in the segment. While inaccuracies of the types mentioned may affect the measured values, the results should still be useful in understanding the gross behavior of concurrent programs.

7.3 Performance of Bounded Buffer

The behavior of the bounded buffer problem (introduced in Section 2.2.2) in the environment hypothesized in Section 7.1 is analyzed here. This problem is chosen because it is typical of client-server problems, it is fairly simple, and it has been used in the literature to demonstrate both shared memory mechanisms [Hoare 74, Campbell 74] and distributed mechanisms [Hoare 78, Brinch Hansen 78]. (Figures 6-1 and 6-2 contain parts of the UNIX-DASH programs which were instrumented in this effort.) The producer processes are modelled as:

```
loop
    delay prod_time; /* models portion production */
    interact with bounded buffer to store portion;
end cycle;
```

Consumer processes are similarly modelled with a delay of cons_time followed by interaction with the bounded buffer to fetch a portion.

The following parameters are needed to describe an instance of the bounded buffer problem: number of slots in buffer, number of producer processes, prod_time distribution, number of consumer processes, cons_time distribution, and portion size. The basic experiment performed investigates a small subspace of the large parameter space. However, from the insight gained in the experiment, the behavior of the problem at other points in the parameter space

may be inferred. The parameters used are: one buffer process with 5 slots, 15 producers, 15 consumers, a negative exponential prod_time distribution with means ranging from 0 to 200 ms, 0 cons_time, and a portion size of 8 bytes. The choice of the number of slots and the number of producers and consumers has been influenced by an empirical study [Haridi 81] which uses the bounded buffer as a benchmark in comparing two concurrent languages on an LSI-11 microcomputer. Since exponential distributions are favored in analytic modelling, because they are mathematically tractable and provide reasonable results, one is used for prod_time. Mean cons_time is set at 0 arbitrarily (variations on the basic experiment include non-zero mean cons_time). The portion size is that used in a test implementation of the bounded buffer.

7.3.1 Simulation of Distributed Solution

At this stage, a further assumption is made: the delay experienced by data is the same as that experienced by a signal. This assumption is examined further in the discussion on local-area networks in Section 7.3.3.

Interactions with Buffer

The bounded buffer is implemented as a process which serves both producers and consumers. The interaction between a producer, a consumer and the buffer server was illustrated in Figure 3-1 (page 44). Producers compete for the databuffer of the Buffer server. So, they send request signals (of type insert) to the Buffer server. When the server chooses to honor the request, it sends an ok signal to the appropriate producer. On receipt of this signal, the producer may perform a data transfer. Likewise, consumers send remove signals

to the Buffer server. The server responds by transferring data to the requesting consumer. At any one time, the server is working on at most one request.

From the interaction pattern described above, one may deduce how performance degrades as transmission delay increases. The performance measure is the throughput (number of portions leaving per unit time) of the buffer server. In the case when the server is saturated (i.e. never idle from lack of requests), throughput is inversely related to the time spent on each portion by the server: servicing the insert signal, dispatching the ok signal, awaiting the arrival of data, storing the data in its buffers, retrieving it, and dispatching it to a consumer via a data transfer. As delay increases, the only actions affected are the transmission time for the ok signal and the data transfer by the producer; the time for each of these increases by the amount the transmission time increases. So, for the saturated case, the inverse of the throughput increases linearly, with a slope of 2, as the transmission delay increases. If the server is not saturated, then as transmission delay increases, the server will reach saturation--since the time servicing a portion increases with delay because a transmission exchange with the producer is required. Hence, if inverse throughput (time/portion) is used as a performance measure, we may conclude that as transmission delay increases, the asymptotic inverse throughput increases linearly (with a slope of 2). Because of this result, inverse throughput is chosen as the performance measure for the bounded buffer problem.

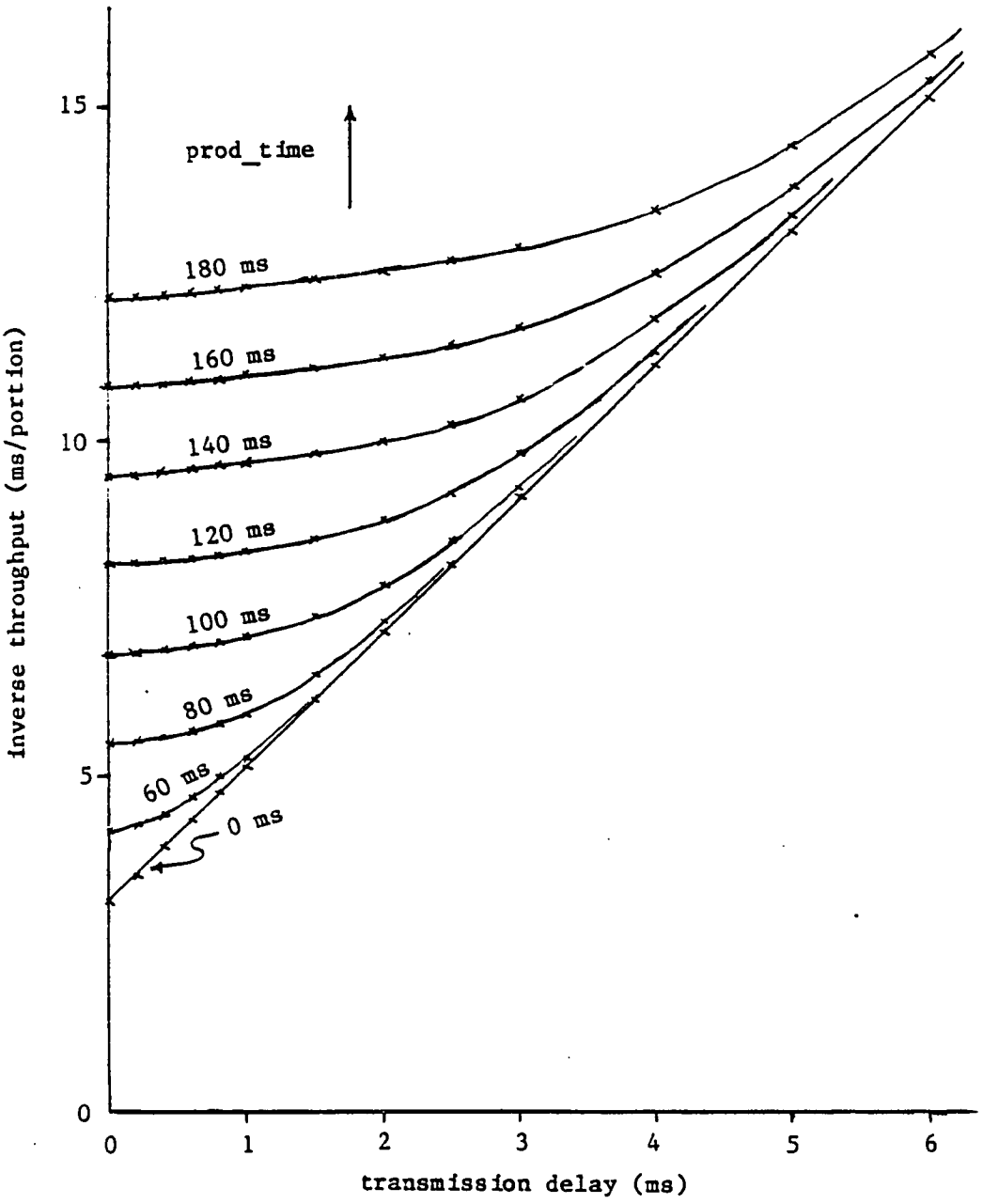


Figure 7-2: Performance of Bounded Buffer

Simulation Results

The inverse throughput values obtained from simulation have a 95% confidence interval of at most ±5% of the experimental value. Figure 7-2 shows performance for transmission delays ranging from 0 to 6ms and prod_time varying from 0 to 180 ms. A separate curve is drawn for each value of prod_time used. For slower producers (large prod_time), the curves are fairly flat at small transmission delays and approach a linear asymptote for larger values of delay. This is consistent with the behavior anticipated in Figure 7-1. When producers are fast, the inverse throughput blends into the asymptote at much smaller delay values. The measured slope of the asymptote is 2, agreeing with the prediction of the preceding paragraph. The 0-delay inverse throughput of the asymptote is measured as 3.17 ms. This corresponds to the (0-delay) service time per portion of a saturated server and agrees with the saturated server service time per portion obtained from the instrumentation.

It is interesting to examine the 0-delay inverse throughput for each of the curves. Two factors influence 0-delay behavior: average interarrival time of producer requests and service time (processing time per request). For small prod_time, the server is saturated, causing the inverse throughput to be close to the server time. For large prod_time, performance is dictated by the inter-arrival time (which can be approximated by prod_time/#producers). In any case, the 0-delay throughput cannot be less than max(servicetime, prod_time/15). The transition between the two types of behavior can be said to occur when servicetime is equal to prod_time/15. For a servicetime of 3.17 ms, the transition occurs at a prod_time of about 48 ms. It can be observed in Figure 7-2, that for prod_time much

larger than 48 ms, the 0-delay inverse throughput is close to prod_time/15. For very small prod_time, the 0-delay inverse throughput is approximately equal to servicetime (3.17 ms).

The above analysis makes some approximations. The inter-arrival time at the buffer is approximated by prod_time/15. This is reasonable for large values of prod_time (when the time spent by the producer interacting with the buffer is relatively small). No attempt has been made to model performance in the vicinity of the transition point. In this region, both inter-arrival time and service time must be considered in performance analysis. Despite these shortcomings, the analysis above is useful in obtaining quick estimates of program performance.

Variations of Experiment

Empirical studies of some variations of the basic experiment have been made. In one study, the prod_time was held constant at 0 and cons_time varied from 0 to 200 ms. The results did not significantly differ from Figure 7-2 (with prod_time replaced by cons_time). When the simulation was run with equal values of prod_time and cons_time, there was a small deterioration of performance (as expected). Also, giving producer requests or consumer requests priority in the bounded buffer process did not affect performance much.

Finally, the effect of increasing server time (in the buffer) was studied empirically. The results showed the asymptote sliding upwards by the amount the server time increased. The 0-delay performance was observed to be consistent with estimates obtained using an asymptotic analysis similar to the one above.

7.3.2 Simulation of Shared Memory Solution

The bounded buffer is represented as a monitor, with procedures to store and fetch portions. Portions are copied rather than passed by reference. This decision was made in order to avoid having a portion allocator, for the sake of simplicity. A lock is associated with the monitor. Monitor procedures begin by acquiring the lock and release it on completion. Monitor condition waits are implemented with kernel waits and an explicit queue of waiting processes is maintained for each condition. Notify on a condition causes a signal to be sent to a process selected from the condition queue. Direct use of the UNIX implementation of the bounded buffer for performance measurements is infeasible: the use of files to simulate shared memory yields unrealistic performance; also, the presence of multiple processes on one processor violates the assumptions about the experimental environment stated in Section 7.1. The monitor solution was instrumented to measure computation segments delimited by lock acquisition, kernel signals, kernel waits and lock release.

prod_time (ms)	inverse throughput (ms)
0	2.40
20	2.53
40	2.99
60	4.18
80	5.48
100	6.80
120	8.13
140	9.46
160	10.80
180	12.13
200	13.47

Table 7-1: Shared Memory Performance

The results of the simulation appear in Table 7-1. The inverse throughput exhibits the same limiting behavior discussed earlier--closely related to the server time for small prod_time, and approaching the mean inter-arrival time for large prod_time. Variations of parameters similar to those reported in Section 7.3.1 were made with similar results on performance.

7.3.3 Discussion

On comparing the performance of the shared memory bounded buffer with the 0-delay performance of the distributed version, it may be noted that the former is about 25% more efficient than the latter for small prod_time. As prod_time increases, the buffer service time becomes less of a factor in determining the performance; hence, the shared and distributed performances become similar. The time spent in the kernel affects the buffer service time and is relevant only near server saturation (low prod_time).

The reasonably similar performance values for shared and distributed communication is reminiscent of the Lauer-Needham hypothesis (Section 5.1). Their environment assumes underlying shared memory to pass information by reference. In our experiments, the 0-delay distributed case and the shared memory case both copy parameters and result in performances that do not differ significantly. This reinforces the hypothesis that distributed and shared solutions to a problem perform comparably, assuming there is no transmission delay and the same parameter passing technique (by reference or by copying) is used in each case.

Having observed comparable performance for 0-delay, it is interesting to determine how long this remains true as delay increases. In other words, we need to find the range of delay times

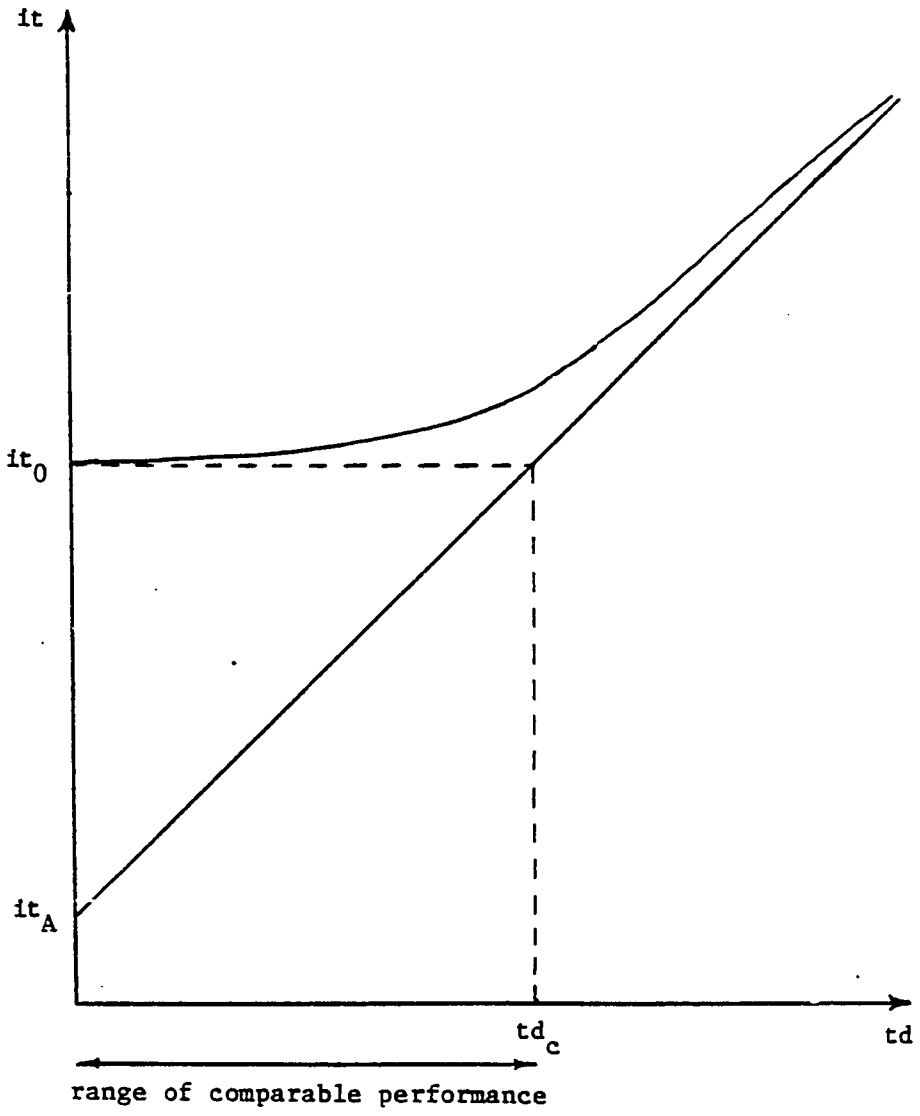


Figure 7-3: Range of Comparable Performance

for which a performance curve in Figure 7-2 remains "flat". A possible measure can be obtained by drawing a horizontal line from the 0-delay intercept of the prod_time curve of interest, up to the asymptote. The transmission delay at which this line intersects the asymptote is defined as the upper limit of the comparable performance region. This is illustrated in Figure 7-3. If the equation of the asymptote is known to be

$$it = slope*td + it_A,$$

where it = inverse throughput
it_A = intercept of asymptote
td = transmission delay

and the 0-delay inverse throughput (it_0) is given, then the cutoff (td_c) can be calculated as follows:

$$td_c = (it_0 - it_A)/slope$$

This suggests that td_c could be predicted (without simulation) if one could examine the bounded buffer problem and estimate the asymptote (it_A,slope) and it_0. We have already seen how to estimate the asymptote (Section 7.3.1). If the server is not saturated, otherwise, by our definition $td_c \simeq 0$, it_0 can be approximated by prod_time/#producers. Now, by a quick analysis of the bounded buffer, we can obtain an estimate for the range of transmission delay over which the performance of a distributed implementation is not much worse than that of one using shared memory.

Since end-to-end transmission delay is central to this chapter, delay in a specific local area network is examined, to place our results in perspective. The Ethernet [Metcalfe 76] is chosen, because it has been in operation at Xerox for over 6 years, and has been discussed much in the literature. While performance of the low

level packet transport mechanism has been studied extensively [Shoch 80], only recently have there been attempts to measure the cost of end-to-end transport. Spector (1981) has implemented a remote store instruction on Xerox Altos connected by a 2.94 megabit Ethernet. The instruction is robust against communication failure (lost packets), but may not work if processors fail. If implemented in software, the instruction executes in 4.8 ms; in firmware it takes 0.155 ms (a speedup by a factor of 30). In another study [Nelson 81], reliable remote procedure calls are implemented on a 2.94 megabit Ethernet. The calls function well in the presence of communication failures and processor failures. Times reported for the calls vary depending on the customization of the protocols and the degree to which microcode is used; they range from over 30 ms to under 1 ms. The complexity of implementing reliable signals and data transfers is in between those for remote store and remote procedure call (Section 6.2). Hence, it is felt that end-to-end transmission delay (for signals and data) of the order of 1 ms is achievable on an Ethernet. Nelson's data also indicates that transmission delay increases rather slowly as the number of parameters in the remote procedure call increase. This reassures us that the assumption in Section 7.3.1 (that signals and data experience the same transmission delay) is not a poor one, and that the range of transmission delays used is of practical interest.

7.4 Performance Prediction for Other Problems

Having analyzed the bounded buffer extensively, a strategy for problem analysis suggests itself. First, the interaction between processes in terms of kernel operations is identified. A process is

isolated which is likely to determine overall performance[2]. In server problems, the server process is invariably the one on which to focus. Then the 0-delay inverse throughput of this process for infrequent interactions is calculated as a function of the rate of interaction and other problem parameters. Next, the inverse throughput at non-zero delay for saturated interaction is computed. From this the asymptote may be surmised. We now have the information needed to perform an analysis similar to that in Figure 7-3 to determine the range of comparable performance for distributed and shared memory implementations. This technique is illustrated in this section through application to some other problems.

7.4.1 Server Problems

Though the bounded buffer falls in this class, it is instructive to consider some other examples. There are many problems which make a single class of requests to a server and await completion (e.g., alarm clock). The analysis for such problems is straightforward. Figure 7-4 shows typical client-server interaction involving handshaking to obtain the input parameters. The server 0-delay inverse throughput for infrequent interactions can be estimated from the time spent by each client between server requests. The asymptote once again has a slope of 2 because of the handshaking required to get the input parameters. The 0-delay intercept of the asymptote is the server time required for each request. Since this problem is quite similar to the bounded buffer, detailed analysis of

---

[2]This is easily done for simple problems; in general, such a process may not be obvious.

it is not presented here. Instead, another example is carefully studied.

Consider a server which allocates single units of a resource (e.g., a buffer allocator). Client processes request a resource unit and eventually release it back to the server. The client server interaction is modelled in Figure 7-5. Clients signal the server to request a resource. The server eventually honors the request by transferring information about the allocated resource to the client. The client proceeds to use the resource and the server is free to service other requests. When the client has finished using the resource, it signals the server to indicate this. It is assumed that the characteristics of the resource need not be sent back to the server, and hence a signal suffices.

For our analysis, let there be m clients, and n resources in the server. Let each client compute for time tc between the times it holds the resource. Once it acquires the resource, it uses it for time tu. The server time to process a resource request (when a resource is available) is ts_{req}. A release is processed in time ts_{rel}. As before, the transmission delay is assumed to be td. Inverse throughput at the server (it) is used as the performance measure. First consider the limiting case where there are infrequent server requests (tc is very large) and td is zero. In this case, the request rate determines the inverse throughput. Each client requests the resource every $tc+ts_{req}+tu$ time units. Since there are m clients,

$$it = (tc + ts_{req} + tu)/m, \qquad \text{(tc very large).}$$

The second limiting case occurs when the server is saturated, because tc is small. In this case, while a client holds an allocated buffer,

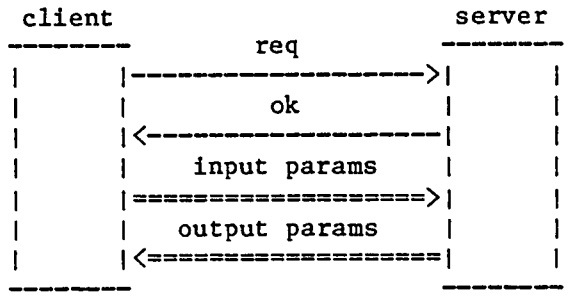
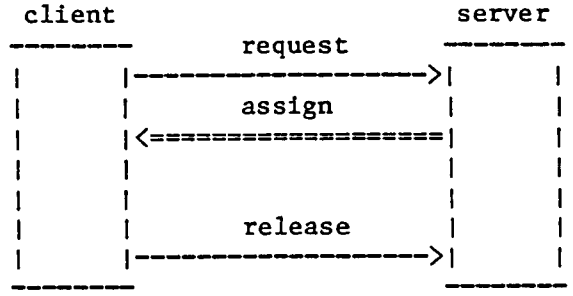


Figure 7-4: Client interactions with a simple server




```
client:
    loop
        compute (for time tc);
        request resource;
        wait till assigned;
        use resource (for time tu);
        release resource;
    end
```

Figure 7-5: Interactions with a resource allocation server

n-1 requests and n-1 releases are made by other clients. Hence, the time at the server between the client's request being honored and the release of the resource allocated to the client is max(tu+2*td, (n-1)*(ts_{req}+ts_{rel})). Hence n requests are serviced in time ts_{req} + max(tu+2*td, (n-1)*(ts_{req}+ts_{rel})) + ts_{rel}. This yields

$$it = \max[(tu + 2*td + ts_{req} + ts_{rel})/n,\ (ts_{req} + ts_{rel})], \qquad \text{(tc very small)}$$

The asymptote has a slope of 2/n. If in addition td is 0, then

$$it = \begin{cases} ts_{req} + ts_{rel}, & tu \leq (n-1)*(ts_{req} + ts_{rel}) \\ (tu + ts_{req} + ts_{rel})/n, & tu \geq (n-1)*(ts_{req} + ts_{rel}) \end{cases}$$

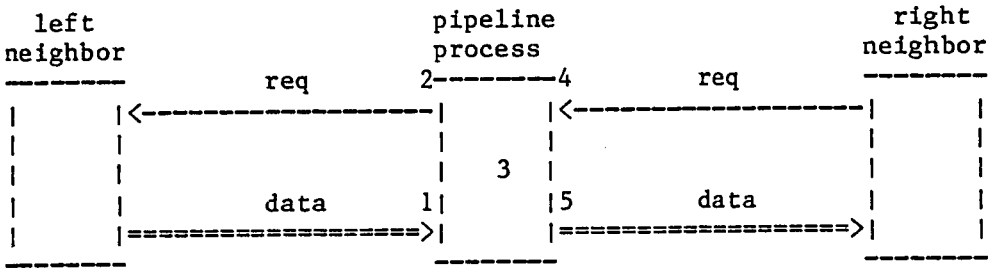
When tu is larger than (n-1)*(ts_{req} + ts_{rel}) (which is reasonable if n is not very large, since ts_{req} and ts_{rel} should be small), the intercept of the asymptote is (tu + ts_{req} + ts_{rel})/n. The transition between the two limiting cases occurs when

$$tc = tc_{trans} = m*\max[(tu + ts_{req} + ts_{rel})/n,\ (ts_{req} + ts_{rel})] - ts_{req} - tu$$

For tc greater than tc_{trans}, the first limiting case should be used to approximate the 0-delay intercept for the performance curve. With these results, the cutoff transmission delay can be determined (Section 7.3.3).

7.4.2 Pipeline Problems

Squash asterisks, sorting array and Eratosthenes' sieve are examples of pipeline problems. For our analysis we assume that all the processes in the pipeline are identical; hence, the throughput of the pipeline can be determined by examining any component stage.




```
pipeline process:
    loop
        1. await data from left; copy it out;
        2. request data from left;
        3. compute (for time tc);
        4. await request from right;
        5. transfer data to right;
    end
```

Figure 7-6: Interactions in a pipeline

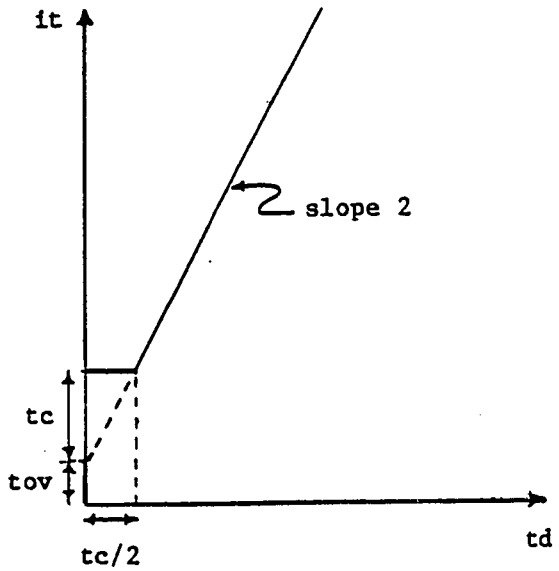


Figure 7-7: Performance of a pipeline

Figure 7-7 depicts one such stage (process) of a pipeline where information flows from left to right. The process receives data from the left, copies it out of its buffer immediately (making the buffer available for the next input) and sends a signal to its left neighbor requesting further input. It then computes for time tc, waits until its right neighbor is ready to accept the output and finally sends the output data. Transmission delay is td and the overhead time in using the kernel and copying the input data is represented by tov.

The two limiting cases in this problem arise when tc is large and tc is small compared to td. In the first case, by the time step 3 in Figure 7-7 is completed, the request for input from the right neighbor is already present (because a fresh request from the right arrives approximately time 2*td later, in response to data sent to the right in the previous cycle). Hence,

it = tc + tov.

In the second case, the process has to wait at step 4 for the request signal from the right. This signal arrives about time 2*td after the current cycle began. Hence,

it = 2*td + tov.

This is the equation of the asymptote: it has a slope of 2 and a 0-delay intercept of tov. The performance is sketched as a function of td in Figure 7-7. This completes the analysis of the pipeline problem.

7.5 Summary

This chapter has attempted to compare the cost of distributed and shared memory implementations of a problem using DASH. As part

of this, the effect of transmission delay on a distributed implementation of a problem has been estimated. An empirical study of one problem, the bounded buffer, has been performed in detail for a hypothetical single process per processor architecture. Data obtained from measurements made on a UNIX-DASH implementation of the problem was used in a simulation of the architecture. Simulation results are reasonably close to simple analytic predictions based on asymptotic bounds. The performance of the bounded buffer in a simulated shared memory architecture does not differ significantly from the performance of the distributed solution at 0 transmission delay. For high delay values, the inverse of the throughput for the distributed implementation approaches a linear asymptote. A criterion has been suggested for determining the range of transmission delay over which the performance in the two architectures is comparable. Finally, the performance of some other problems have been predicted by using this criterion on the results obtained from a quick asymptotic analysis of each problem.

CHAPTER 8. CONCLUSIONS

8.1 Looking Back

The DASH kernel demonstrates the feasibility of using a unified approach to distributed, shared and hybrid communication. A small set of simple kernel operations has proven adequate for the needs of a wide variety of programming applications. The kernel has been used to model a number of common communication mechanisms; hence its expressive power is no less than any of these. By reflecting the characteristics of the implementing physical architecture, the kernel encourages the programmer to take advantage of the efficiencies offered by the architecture. This could necessitate changes to programs when the implementing architecture changes; should such changes be anticipated, the programmer should be constrained to use high-level architecture-independent mechanisms, which in turn are implemented in DASH for the architecture at hand. The current static nature of DASH is a limitation: communication involving dynamic processes and dynamic locks is not supported.

The subject of programming in hybrid environments has been given special attention. A number of situations have been identified where a process is involved in both distributed and shared communication. Methods for programming such situations have been suggested, which bring together features for both distributed and shared programming. The use of such methods has been illustrated through several examples.

The uniprocessor implementation of DASH indicates the practical nature of the kernel. The implementation was done using the existing UNIX features and required almost no changes to UNIX.

However, performance would be greatly improved if UNIX supported shared memory. Also, a convenient procedural interface to the C programming language was developed. It is believed that implementation on other uniprocessor operating systems and the provision of interfaces in other programming languages would be straightforward. Preliminary indications suggest that DASH is readily implementable on a network of processors, especially a local area network. However, experience with an actual network implementation is needed in order to evaluate the suitability and efficiency of DASH.

The somewhat low-level nature of DASH, while providing flexibility, may appear cumbersome to programmers of large applications. In such cases, it is expected that a higher level mechanism suitable for the particular application will be chosen and programmed in DASH by a "systems" programmer; libraries of such mechanisms can be developed to facilitate application programming. The routines to support programming in CSP and Ada are examples of such utilities.

Another use of DASH is as a description tool. Algorithms that consist of process interaction (especially hybrid interaction), may be expressed using DASH operations for communication. Further, DASH may be used to describe the semantics of new communication mechanisms (analogous to the use of semaphores in describing the semantics of monitors).

One may gain some perspective by seeing where the DASH kernel stands with respect to other interprocess communication facilities discussed in this work. Figure 8.1 places mechanisms in two columns: one for shared memory and the other for distributed communication. The higher a mechanism is on each column, the more "structured" view

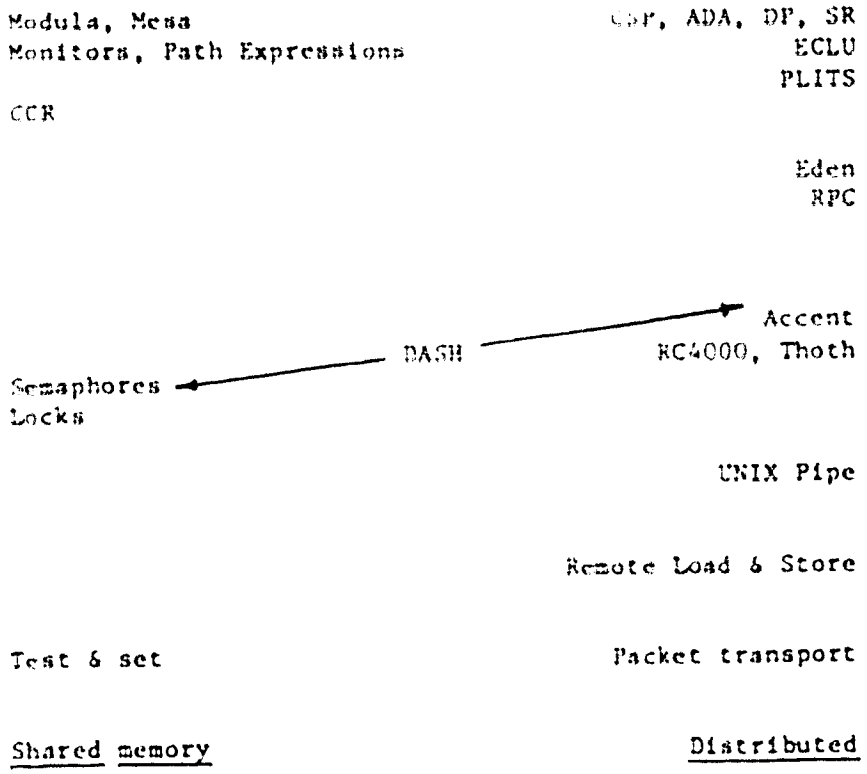


Figure 8-1: Mechanisms for Interprocess Communication

of communication it offers. For example, semaphores and locks are comparable to each other, and provide a somewhat lower level of abstraction than the message passing facilities in Thoth and RC4000. The DASH kernel operations fall roughly in the middle of both columns; in the figure DASH is listed in between the columns with arrows extending to each column. The arrows also indicate that DASH blends together the two worlds represented by the two columns. It provides an abstraction comparable to semaphores in the shared memory world, and at a level somewhat higher than message passing in the distributed world.

8.2 Looking Ahead

In Section 3.4 an extension to DASH allowing for dynamic processes was suggested. In addition to designing process creation primitives, one would have to be concerned about issues such as process naming and communication with nonexistent processes. The former issue requires a method for name allocation, and some means for transmitting a name between processes. The latter issue relates to the detection, generation and handling of exceptions.

General purpose communication mechanisms for high level languages can be implemented using DASH. However, in Section 3.3.2 it was observed that under certain structured patterns of process interaction, it is possible to optimize the number of DASH transmissions performed. This suggests the following areas for further study: the identification of structured interaction patterns, and the design of programming language features which would allow a compiler to perform such optimizations.

While using the uniprocessor implementation of the kernel in programming applications, the absence of debugging facilities for multiprocess programs was painful. This is a problem that is often overlooked by the designers and implementors of systems for concurrent programs. To facilitate program development, a multiprocess debugger needs to be designed and implemented. In addition to the usual features found in single process debuggers, it should be possible to display and modify the communication state of a given process as reflected by the DASH data structures. Also, while constructing a network implementation of DASH, some effort would be required in developing adequate debugging tools.

In comparing the performance of problems in distributed and

shared environments, an asymptotic analysis was used. By constructing a simple queueing model, it may be possible to better estimate performance for regions where the limiting conditions do not hold. Another extension of practical interest would be to simulate the sharing of processors to estimate the related cost. A model is also required for the impact of the operating system and other processes on a given process; these factors should be included in the comparative analysis.

Finally, it would be useful to formally specify the kernel operations. Such a specification would convey the precise meaning of the operations to a kernel user as well as to an implementor of the kernel. It is desirable that the specification be usable in proofs of program properties e.g., partial correctness, termination, absence of deadlock. The algebraic specification technique [Guttag 77] is a possible candidate. In a recent thesis [Mallgren 81], this technique has been extended to include parallelism. While we believe that it is possible to specify the kernel with an algebraic technique, we are unsure as to how such specifications can be used to prove program properties. Another approach is to attempt to develop proof rules for the kernel operations (similar to the proof rules that have been suggested for CSP [Levin 79, Apt 80]). Two problems are foreseen in using the CSP proof rules approach: asynchronous signals are harder to describe than synchronous CSP operations; also, since CSP communication resembles a remote assignment, on completion of communication, the receiver has considerable state information about the sender--this is not possible with signals. Perhaps, the effect of statements that surround the signal and wait will need to be included in arriving at meaningful proof rules and assertions. Much needs to be done in this area yet.

REFERENCES

[Ada 79a] Ichbiah, J.D. et al. Preliminary Ada Reference Manual. SIGPLAN Notices 14(6), June 1979. Part A.

[Ada 79b] Ichbiah, J.D. et al. Rationale for the Design of the Ada Programming Language. SIGPLAN Notices 14(6), June 1979. Part B.

[Andrews 81] Andrews, G.R. Synchronizing Resources. TOPLAS 3(4):405-430, October 1981.

[Apt 80] Apt, K.R., Francez, N. and de Roever, W.P. A Proof System for Communicating Sequential Processes. TOPLAS 2(3):359-385, July 1980.

[Baer 81] Baer, J.-L., Gardarin, G., Girault, C. and Roucairol, G. The Two-Step Commitment Protocol: Modelling, Specification and Proof Methodology. In The Fifth International Conference on Software Engineering, pages 363-373. IEEE, March 1981.

[Bernstein 80] Bernstein, A.J. Output Guards and Nondeterminism in Communicating Sequential Processes. TOPLAS 2(2):234-238, April 1980.

[Brinch Hansen 70]
Brinch Hansen, P. The Nucleus of a Multiprogramming System. Communications of the ACM 13(4):238-241, April 1970.

[Brinch Hansen 73]
Brinch Hansen, P. Operating System Principles. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

[Brinch Hansen 78]
Brinch Hansen, P. Distributed Processes: A Concurrent Programming Concept. Communications of the ACM 21(11):934-941, November 1978.

[Campbell 74] Campbell, R.H., and Habermann, A.N. Lecture Notes in Computer Science. Volume 16: The Specification of Process Synchronization by Path Expressions. Springer-Verlag, 1974, pages 89-102.

[Cheriton 79a] Cheriton, D.R., Malcolm, M.A., Melen, L.S. and Sager, G.R. Thoth, a Portable Real-Time Operating System. Communications of the ACM 22(2):105-115, February 1979.

[Cheriton 79b] Cheriton, D.R. Multi-process Structuring and the Thoth Operating System. Technical Report 79-5, Computer Science Dept., University of British Columbia, March 1979.

[Courtois 71] Courtois, P.J., Heymans, F. and Parnas, D.L. Concurrent Control with "Readers" and "Writers". Communications of the ACM 14(10):667-668, October 1971.

[Dennis 66] Dennis, J.B. and VanHorn E.C. Programming Semantics for Multiprogrammed Computations. Communications of the ACM 9(3):143-155, March 1966.

[Dijkstra 68] Dijkstra, E.W. The Structure of the "THE" Multiprogramming System. Communications of the ACM 11(5):341-346, May 1968.

[Dijkstra 75] Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM 18(8):453-457, August 1975.

[Eden 81] Lazowska, E.D., Levy, H.M., Almes, G.T., Fischer, M.J., Fowler, R.J. and Vestal, S.C. The Architecture of the Eden System. In Proc. Eighth Symp. on OS Principles, pages 148-159. ACM, December 1981.

[Enslow 78] Enslow, P.H. What is a 'Distributed' Data Processing System? COMPUTER 11(1):13-21, January 1978.

[Feldman 79] Feldman, J.A. High Level Programming for Distributed Computing. Communications of the ACM 22(6):353-368, June 1979.

[Francez 80] Francez, N. Distributed Termination. TOPLAS 2(1):42-55, January 1980.

[Franta 77] Franta, W.R. The Process View of Simulation. Elsevier North-Holland, 1977.

[Guttag 77] Guttag J. Abstract Data Types and the Development of Data Structures. Communications of the ACM 20(6):396-404, June 1977.

[Habermann 72] Habermann, A.N. Synchronization of Communicating Processes. Communications of the ACM 15(3):171-176, March 1972.

[Haridi 81] Haridi, S., Bauner, J., and Svensson, G. An Implementation and Evaluation of Tasking Facilities in Ada - Summary. SIGPLAN Notices 16(2):35-47, February 1981.

[Hoare 74] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. Communications of the ACM 17(10):549-557, October 1974.

[Hoare 78] Hoare, C.A.R. Communicating Sequential Processes. Communications of the ACM 21(8):666-677, August 1978.

[Howard 76a] Howard, J.H. Signaling in Monitors. In The Second International Conference on Software Engineering, pages 47-52. IEEE, October 1976.

[Howard 76b] Howard, J.H. Proving Monitors. Communications of the ACM 19(5):273-279, May 1976.

[Jensen 75] Jensen, K. and Wirth, N. PASCAL: User Manual and Report. Springer-Verlag, 1975.

[Kernighan 78] Kernighan, B.W. and Ritchie, D.M. The C Programming Language. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1978.

[Kieburtz 79] Kieburtz, R.B. and Silberschatz, A. Comments on Communicating Sequential Processes. TOPLAS 1(2):218-228, October 1979.

[Lamport 74] Lamport, L. A New Solution of Dijkstra's Concurrent Programming Problem. Communications of the ACM 17(8):453-455, August 1974.

[Lampson 80] Lampson, B.W. and Redell, D.D. Experience with Processes and Monitors in Mesa. Communications of the ACM 23(2):105-117, February 1980.

[Lampson 81] Lampson, B.W. Lecture Notes in Computer Science. Volume 105: Atomic Transactions. Springer-Verlag, 1981, pages 246-265. (Chapter 11, in Distributed Systems: Architecture and Implementation, an Advanced Course, Lampson, B.W., editor).

[Lauer 78] Lauer, H.C. and Needham, R.M. On the Duality of Operating System Structures. In Proc. Second Int. Symp. on Operating Systems. IRIA, October 1978. Reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.

[Le Lann 77] Le Lann, G. Distributed Systems--Towards a Formal Approach. In Information Processing 77, pages 155-160. North-Holland, August 1977.

[Levin 79] Levin, G.M. A Proof Technique for Communicating Sequential Processes (with an example). Technical Report 79-401, Computer Science Dept., Cornell University, 1979.

[Liskov 79] Liskov, B. Primitives for Distributed Computing. In Proc. Seventh Symp. on OS Principles, pages 33-42. ACM, December 1979.

[Liskov 80] Liskov, B., editor. Report on the Workshop on Fundamental Issues in Distributed Computing. ACM, 1980. Appears in SIGPLAN Notices 16(10):20-49, October 1981, and in Operating Systems Review 15(3):9-38, July 1981.

[Lister 77] Lister, A. The Problem of Nested Monitor Calls. Operating Systems Review 11(2):5-7, July 1977.

[Mallgren 81] Mallgren, W.R. Formal Specification of Interactive Graphics Programming Languages. Ph.D. thesis, University of Washington, Computer Science Department, September 1981. Available as technical report no. 81-09-01.

[Mao 80] Mao, T.W. and Yeh, R.T. Communication Port: A Language Concept for Concurrent Programming. IEEE Transactions on Software Engineering SE-6(2):194-204, March 1980.

[Mesa 79] Mitchell, J.G., Maybury, W. and Sweet, R. Mesa Language Manual. Technical Report CSL-79-3, Xerox, Palo Alto Research Center, April 1979.

[Metcalfe 76] Metcalfe, R.M. and Boggs, D.R. Ethernet: Distributed Packet Switching for Local Computer Networks. Communications of the ACM 19(7):395-404, July 1976.

[Mohan 80] Mohan, C. A Perspective of Distributed Computing: Models, Languages, Issues and Applications. Technical Report DSG-8001, Computer Science Dept., University of Texas at Austin, March 1980.

[Nelson 81] Nelson, B.J. Remote Procedure Call. Ph.D. thesis, Carnegie-Mellon University, Computer Science Department, May 1981. Available as reports CMU-CS-81-119 or Xerox CSL-81-9.

[Odano 82] Odano, I. and Yenbut, V. The CSP Language. 1982. University of Washington.

[Parnas 78] Parnas, D.L. The Non-Problem of Nested Monitor Calls. Operating Systems Review 12(1):12-15, January 1978.

[Pouzin 78] Pouzin, L. and Zimmerman, H. A Tutorial on Protocols. Proceedings of the IEEE 66(11):1346-1370, November 1978.

[Rao 80] Rao, R. Design and Evaluation of Distributed Communication Primitives. In Proc. ACM Pacific '80, pages 14-23. ACM, November 1980. Also appears as Univ. of Washington, Computer Science Tech. Report 80-04-01.

[Rao 81] Rao, R. DASH: A Communication Kernel. 1981. University of Washington Computer Science Laboratory, Technical Note No. 133.

[Rashid 81] Rashid, R.F. and Robertson, G.G. Accent: A Communication Oriented Network Operating System Kernel. In Proc. Eighth Symp. on OS Principles, pages 64-75. ACM, December 1981.

[Ricart 81] Ricart, G. and Agrawala, A.K. An Optimal Algorithm for Mutual Exclusion in Computer Networks. Communications of the ACM 24(1):9-17, January 1981.

[Ritchie 74] Ritchie, D.M. and Thompson, K. The UNIX Time-Sharing System. Communications of the ACM 17(7):365-375, July 1974.

[Ritchie 78] Ritchie, D.M. UNIX Time-Sharing System: A Retrospective. The Bell System Technical Journal 57(6):1947-1969, July-August 1978.

[Shaw 74] Shaw, A.C. The Logical Design of Operating Systems. Prentice-Hall, Inc., Englewood Cliffs, N.J., 1974.

[Shaw 78] Shaw, A.C. Software Descriptions with Flow Expressions. IEEE Transactions on Software Engineering SE-4(3):242-254, May 1978.

[Shaw 79] Shaw, A.C. Software Specification Languages based on Regular Expressions. Technical Report 31, Institut fur Informatik, ETH (Zurich), June 1979.

[Shoch 80] Shoch, J.F. and Hupp, J.A. Measured Performance of an Ethernet Local Network. Communications of the ACM 23(12):711-721, December 1980.

[Silberschatz 79] Silberschatz, A. Communication and Synchronization in Distributed Systems. IEEE Transactions on Software Engineering SE-5(6):542-546, November 1979.

[Silberschatz 81] Silberschatz, A. Port-Directed Communication. The Computer Journal 24(1):78-82, January 1981.

[Spector 82] Spector, A.Z. Performing Remote Operations Efficiently on a Local Computer Network. Communications of the ACM 25(4):246-260, April 1982.

[Welsh 79] Welsh, J., Lister, A. and Salzman, E.J. Lecture Notes in Computer Science. Volume 79: A Comparison of Two Notations for Process Communication. Springer-Verlag, 1979, pages 225-254.

[Wettstein 78] Wettstein, H. The Problem of Nested Monitor Calls Revisited. Operating Systems Review 12(1):19-23, January 1978.

[Wirth 77] Wirth, N. Modula: a Language for Modular Multiprogramming. Software--Practice and Experience 7(1):3-35, January 1977.

[Yeh 80] Yeh, R.T. Private communication, June, 1980.

[Yenbut 81] Yenbut, V. and Odano, I. CSP Compiler Implementation. 1981. University of Washington.

APPENDIX A. DEFINITIONS FOR COMMUNICATION ARCHITECTURES

We first survey some of the definitions that have appeared in the literature. Then a set of informal definitions is given for the context of this work, in order to provide a good intuitive feel for the terms used.

In an attempt to characterize distributed data processing systems [Enslow 78], the following properties are considered essential for such systems:

- A multiplicity of general purpose (physical or logical) resource components dynamically assignable to tasks.
- A network (using a two-party cooperative protocol) providing communication between physically distributed components.
- A high-level OS unifying and integrating control of components.
- Services requested by name only (and not by server id).
- Resources which are autonomous and have cooperative interactions with other resources.

Further, the degree of decentralization of hardware, control and data base are examined in order to define boundaries for a variety of aspects of distributed computing--communication, data bases, operating systems, and algorithms. Perhaps because of this generality, the criterion for identifying distributed systems remains 'fuzzy'. However, the requirement placed on communication is relevant in our search for a definition.

LeLann (1977) has proposed a definition based on the states of a system. A system is viewed as a set of logical entities (processes) working together to achieve specific functions.

Associated with each entity is a state. The system state is a vector comprising of the component entity states. A system is centralized if there is an entity which always knows the system state; otherwise, it is decentralized. By applying this definition to the communication state of various processes, we may derive some insight as to what "distributed communication" is.

Since this section deals with processes, memories and their interconnections, some understanding of these terms is required. An informal definition of the term "process", based on the one in [Shaw 74, p. 58] is:

> A process is the activity resulting from the execution of a sequential program (with its data) by a sequential processor.

Logically, each process has its own processor. The logical processor could be one or more shared or unshared physical processors.

Each process in performing its activity needs to interact with memory for its program and data. Information can be stored in memory and retrieved unmodified at a later time. Memory is passive, in that it does not initiate activity but serves the process(es) which may access it. Processes, however are active components.

The connection between a process and memory is called a process-memory (P-M) link. Similarly, should two processes be directly connected (without going through memory), the connection is a process-process (P-P) link. A process views its P-M links differently from its P-P links. The delay in transferring data over the former is normally much less than over the latter. Often, the former has much larger bandwidth. Memory, being passive, is assumed to be always ready to transfer data to/from a process--memory contention is ignored. The P-P link is between two active

components. It has no built-in memory to buffer communication. Hence, both processes must be fully synchronized for data transfer to take place.

Examples of P-M links can be found in Load and Store instructions in assembly languages, or in the memory references required to implement assignment and expression evaluation in Algol-like languages. Process communication in CSP [Hoare 78] occurs over a P-P link: both processes actively participate in the synchronous transfer of information. Modelling of buffered communication (as in the RC 4000 system [Brinch Hansen 70]) requires more than just a P-P link. This is discussed later in this section.

Descriptions made using the terms informally introduced above are only appropriate at one level of abstraction. For example, what is considered a P-P link at one level may, at a lower level, be described in terms of a graph containing processes, memory, P-P and P-M links. The definitions that follow are also subject to the one level of abstraction restriction.

In the figures that follow, circles denote processes, boxes denote memory, solid lines represent P-M links and broken lines P-P links.

Memory M is directly accessible by a process P if there is a P-M link connecting P to M. For example, M_1 is directly accessible by P_1 in Figure A-1.

Memory is shared if it is directly accessible by more than one process; it is local if it is directly accessible by only one process. In Figure A-1, M_3 is shared by P_1 and P_2; M_2 is local to P_2.

Consider graphs whose vertices are either processes or memory and whose edges are P-M or P-P links (as appropriate). If such a graph is connected, it is called a system.

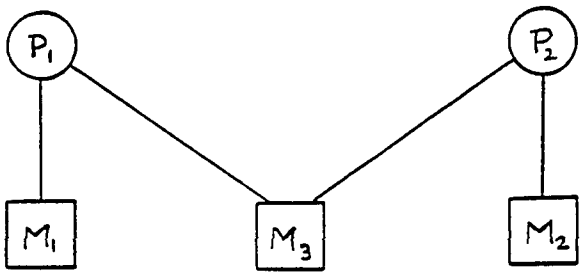


Figure A-1: A strictly shared system

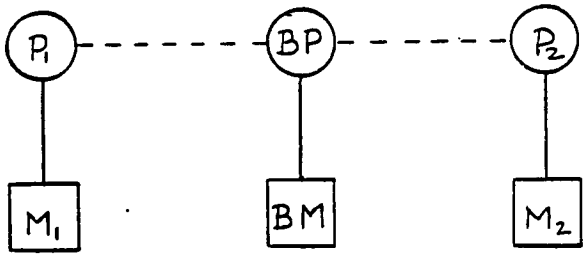


Figure A-2: A strictly distributed system

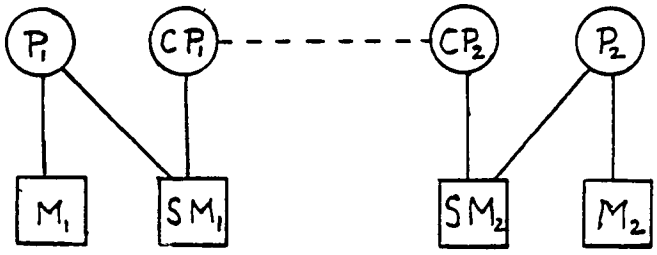


Figure A-3: A hybrid system

A system is strictly distributed if none of the memory is shared. The system in Figure A-2 is strictly distributed.

A strictly shared memory system has no P-P links. Figure A-1 contains an example of such a system.

A hybrid system is one which is neither strictly distributed nor strictly shared memory. (See Figure A-3 for one such system.) Many systems come under this category, though they may be closer to one or the other of the strict schemes.

Now let us look at some familiar systems in the light of these definitions. The UNIX [Ritchie 74] pipe is modelled in Figure A-2. The operating system plays the role of BP, the buffer process, using the pipe buffer memory BM to buffer communications between processes P_1 and P_2.

Consider communication between processes using buffered asynchronous primitives (as in PLITS [Feldman 79]). Figure A-3 models this situation, with the CP's being communication processes. P_1 can do a no-wait send to P_2 by writing into shared memory SM_1. CP_1 then takes the data and transfers it to CP_2.

The definitions presented above, though not formal, hopefully provide conceptual clarity. The commonly used term "distributed system" has not been defined precisely. It could stand for a strictly distributed system or for a hybrid system which has a small amount of shared memory.

APPENDIX B. MODELLING DASH KERNEL WITH CSP

The distributed features of the DASH kernel (Section 3.2.1) are implemented here in the language CSP [Hoare 78]; the purpose is to give an operational definition of DASH semantics. As is true with implementational specifications in general, there is a danger in reading this specification and confusing implementation strategies with the semantics of DASH. The language CSP was chosen, since it is the only "distributed" language whose statements have been axiomatized as of this date [Levin 79, Apt 80]. Comments on the experience of using CSP appear following the implementation.

Associated with every user process, we introduce a communication process which performs most of the kernel functions. A process is also introduced for each wait condition procedure the user needs. The kernel data structures (Signalbox, Buffer) are part of the communication process, together with code for all the kernel operations except for signal, transfer and wait: these are expanded inline in the user process. This break down of DASH functions among these processes was made necessary by CSP. Because signal and transfer have no-wait semantics, and CSP has fully synchronized primitives, we had to introduce a communication process (CP) at the receiver to receive signals and data, and to store them in the appropriate kernel data structures. Also, DASH permits two processes to signal each other, but two processes sending to each other in CSP would deadlock. Hence, the kernel output commands (signal, transfer) cannot be implemented in the CP, but must instead be expanded inline in the user process. The body of each CP consists of a repetition statement with a guarded command for each operation it implements. Wait condition procedures are naturally implemented as separate

processes. Since the wait primitive calls the wait condition procedure, which in turn calls CP (for test and datatest), it was decided not to place the code for wait in CP; instead it appears inline in the user process.

In Figure B-1, kernel operations appear on the left and the equivalent CSP code on the right. The notation used needs some explanation. The CP associated with a process p is denoted p.cp. In the transfer operation b[1..c] denotes the first c units of data starting at location b. (The code for transfer should check that c is less than bufsize.) Also, in the CSP code, the identifier result represents the return value of the boolean operations test and datatest.

The communication process is outlined in Figure B-2. The details of the actions performed on different inputs are suppressed. (They appear in Figure B-3.) The signalbox is represented as an array of signals. (This is an example of an implementation strategy being unnecessarily visible.) The process body consists of a nondeterministic repetition statement which services signals and data arriving from remote processes, as well as requests from its clients, i.e., its master or condition procedure processes invoked by the master. For convenience, we have taken the liberty of using "client" in the CSP code as well. For example, the master's request to block is ignored until there is external input that the master has not seen yet.

The implementation details in Figure B-3 need no explanation, except for the test operation. Each incoming signal is given a time stamp, which is stored in the sigbox array. When a test is invoked with a wild card argument (denoted *), the oldest matching signal is selected and returned.

```
signal(p,t)              p.cp!signal(t)
test(p,t,p',t')          mycp!test(p,t); mycp?(result,p',t')
discard(p,t)             mycp!discard(p,t)

transfer(p,b,c)          p.cp!data(c,b[1..c])
datatest()               mycp!datatest(); mycp?result
datasize()               mycp!datasize(); mycp?size
datacopy(b,i,n)          mycp!datacopy(i,n); mycp?b[1..n]
datareset()              mycp!datareset()

wait(condproc,arglist)   comment invoke select function:
                         condproc!arglist; condproc?(result,arglist);
                         comment await external input:
                         *[~result -> cp!block();
                                      condproc!arglist;
                                      condproc?(result,arglist);
                         ]
```

Figure B-1: Kernel interface

```
comment declarations and initialization
sigbox: (1..Pmax)(1..Tmax) integer; time: integer;
buf: (1..bufsize) integer; buffull: boolean; i,j,n: integer;
size: integer; freshinput: boolean;

i := 1; j := 1; comment initialize signalbox
*[i ≤ Pmax -> *[j ≤ Tmax -> sigbox(i)(j) := 0; j := j+1];
               i := i+1];
time := 0; buffull := false; freshinput := false;

*[ comment consider incoming input
   (i:1..Pmax) P(i)?signal(t) -> PROCESS_INCOMING_SIGNAL
 # (i:1..Pmax) P(i)?data(size,buf[1..size]) -> PROCESS_INCOMING_DATA

   comment consider service requests from my master
 # client?test(p,t) -> TEST
 # client?discard(p,t) -> DISCARD

 # client?datatest() -> DATATEST
 # client?datasize() -> DATASIZE
 # client?datacopy(i,n) -> DATACOPY
 # client?datareset() -> DATARESET

 # freshinput; master?block() -> freshinput := false
]
```

Figure B-2: Skeleton of Communication Process

```
PROCESS_INCOMING_SIGNAL:
        time := time-1;
        [ time < 1 -> time := timemax
        # time ≥ 1 -> skip;
        ];
        sigbox(i)(t) := time; freshinput := true

PROCESS_INCOMING_DATA:
        buffull := true; freshinput := true

DATATEST:
        client!buffull

DATASIZE:
        client!size

DATARESET:
        client!buf[i..i+n-1]

TEST:
        pinit,pfinal,tinit,tfinal,pmatch,tmatch: integer;
        i,j,max: integer;

        [ p=* -> pinit := 1; pfinal := Pmax
        # p≠* -> pinit := p; pfinal := p];
        [ t=* -> tinit := 1; tfinal := Tmax
        # t≠* -> tinit := t; tfinal := t];
        comment search sigbox for oldest relevant signal
        max := 0; i := pinit;
        *[ i ≤ pfinal ->
                j := tinit;
                *[ j ≤ tfinal ->
                        [ sigbox(i)(j) > max -> max := sigbox(i)(j);
                                                pmatch := i;
                                                tmatch := j
                        # sigbox(i)(j) ≤ max -> skip
                        ];
                        j := j+1
                ];
                i := i+1
        ];
        [ max > 0 -> client!(true,pmatch,tmatch)
        # max ≤ 0 -> client!(false,pmatch,tmatch)]

DISCARD:
        sigbox(p)(t) := 0
```

Figure B-3: Details of Communication Process

The entire implementation presented above takes less than a hundred lines of CSP code. From the (short) length of the code one may infer that the DASH kernel is conceptually simple.

Based on this paper implementation experience, some comments can be made on the CSP language. The requirement for explicit naming of the sender by the receiver is very inconvenient. We have circumvented much of this problem by extending the CSP notation with a process variable as the source argument of an input statement. This allows us to use "client" instead of listing the user process and its condition procedures in separate guards. A procedure call abstraction would add much to the language. Since this abstraction was not available, condition procedures had to be coded as separate processes. The Pascal *if* statement without an *else* clause, when modelled in CSP, requires a (dummy) guard corresponding to the *else* part. This is irritating to the programmer and makes the code less readable. Finally, the fully synchronized input/output statements of CSP make it easier to inadvertently write programs that deadlock.

VITA

Ram Rao was born September 5, 1951 in Hyderabad, India, the son of Mirle and Lakshmi Rao. Following completion of secondary schooling at Mayo College, Ajmer, India, he studied Electrical Engineering at the Indian Institute of Technology, Kanpur. He then attended the California Institute of Technology, where he received the Bachelor of Science degree with honors in Engineering and Applied Science in 1974, and the Master of Science degree in Engineering Science in 1975. From 1974 to 1976 he worked at the Jet Propulsion Laboratory in Pasadena, California. Later he attended the University of Washington, where he received the Master of Science degree in Computer Science in 1979. He will soon be joining the technical staff of Bell Laboratories at Indianapolis, Indiana.