SODA: A Simplified Operating System for Distributed Applications

by

Jonathan Kepecs
Marvin Solomon

Computer Sciences Technical Report #527

January 1984

SODA: A Simplified Operating System for Distributed Applications

by

Jonathan Kepecs
Marvin Solomon

This work supported in part by the Defense Advanced Project Research Agency under contract N00014-82-C-2087 and the National Science Foundation under grant MCS-8105904.

This thesis is dedicated to the memory of Cassandra Kepecs (1961-1980).

I wish to thank my advisor, Marvin Solomon, for his constant support, amazing stores of knowledge, and friendship. Without his direction, this thesis would never have been written. I also wish to thank the other readers on my committee who have also been good friends and selflessly given their time, energy and ideas: Raphael Finkel and Randy Katz. Thanks also to professors Jim Goodman, Larry Dahl, and Larry Travis for participating in the final thesis defense.

My friends have been a continual source of entertainment, fertile intellectual stimulation and support. In particular, Linda Riewe, Jane Henderson, Russel Sandberg and Dan Schuh have been there when I needed them. Dan also provided invaluable help with Gremlin which was used to produce the diagrams in this thesis. Thanks to Jon Dreyer for teaching me machine language and strange computer words. Thanks to Paul Beebe for letting me use his office during the final weeks of writing this thesis. Thanks to Joe and Pat Bonner for being good friends and playing great baseball. Thanks to Dave Dewitt for getting me a job and keeping the LSI's working. Thanks to Charles Fischer for many interesting discussions about baseball and parsing. Thanks to Ray Bryant for being a great teacher and advisor. Thanks to everyone in the lab who made those all-nighters an interesting experience: Kishore, Nancy, Tad, Sue, Donn, Ken, Steve, Al, Gana, Lu and Qu. Thanks to Charlie Kepecs for letting me torment him without biting me too often. Finally, thanks to my parents, who put up with hours of loud practicing which required either deafness or patience of a very high order indeed.

TABLE OF CONTENTS

FIGURES, TABLES and PROGRAMS

0. *ABSTRACT*

A *communications adaptor* is a processor that provides efficient *inter-machine* message transport. This thesis proposes a design for a communications adaptor called SODA which also provides *interprocess* communications facilities. The primitives provided by SODA are tailored to the needs of distributed operating systems and high-level distributed programming languages, and are designed for simplicity and uniformity. Our work makes three important contributions to computer science:

(1) Available communications adaptors are not complete in the sense that additional kernel software is required to support process naming, creation and destruction. SODA is a novel communications processor which combines the intermachine message-passing capabilities of a communications adaptor and the interprocess communication facilities traditionally supplied by an operating systems kernel. As a result of combining these two functions, distributed operating systems can be built at reduced cost and with improved performance.

(2) To take advantage of increasingly more inexpensive processors, and to help achieve our goals of simplicity and reduced cost, we require that client processors are *not* multiprogrammed. As a consequence, the SODA primitives require careful design to avoid imposing limitations on applications code. An important result of this thesis is to demonstrate that a uniprogrammed system can function well in a distributed network.

(3) Most message-based systems provide an active SEND and a passive RECEIVE primitive. The SODA primitives provide an active RECEIVE and an active EXCHANGE as well. Our implementation shows that these primitives can be implemented with approximately the same performance as an active SEND. As a result, new styles of interprocess communication are feasible.

We demonstrate the utility of the SODA primitives with numerous programming examples, including a new solution to Dijkstra's "dining philosopher's" problem. A compact and efficient SODA implementation is presented which suggests that an inexpensive SODA processor should be possible to build, thus enabling inexpensive processors to afford to possess a SODA interface. Finally, the design rationale for SODA is given.

1. INTRODUCTION

The cost of processors is continuing to decline. As a result, it is becoming feasible to distribute computations among physically distinct nodes to achieve better performance, extensibility, enhanced reliability, and improved resource sharing capabilities [1].

In a distributed environment, an operating system no longer need consist of a monolithic collection of utilities such as file servers, command interpreters, and device drivers supported by a complex kernel. Instead, services may be scattered throughout the network. System utility processes can be treated like applications processes. We will refer to a process in a distributed system as a *client* process, since it uses the services provided by the operating systems kernel.

Our view of an operating system *kernel* is that it is an entity which provides just those services which *all* client processes will require. The kernel of a distributed operating system should not be concerned with providing direct support for system utilities. Instead, the kernel need only provide a vehicle for resource sharing that is necessary if client processes are to cooperate. We feel that this vehicle defines a necessary and sufficient functionality needed to construct a distributed operating system.

As the role of the kernel decreases, the importance of the communications interface increases as efficient interprocess communication becomes the predominating concern of the kernel. This thesis represents an effort to reduce system cost and improve performance by combining the functions of a distributed operating systems kernel and a communications interface into a single entity called SODA.

1.1. Kernel Requirements

In a distributed network, all nodes contain kernels supporting the same function which presents a uniform interface for all client processes, and allows clients to share the resources of the entire network.

The kernel of a distributed operating system should provide a reliable message exchange service, and mechanisms for process naming, process synchronization, process creation and process destruction.

The notion of simplicity is hard to pin down, yet it is cited by almost every author as a desired feature of his software. Wirth's programming language Pascal [2] and Patterson's computer architecture RISC [3] are examples of the successes of the design goal of simplicity. Hansen [4, 5] is emphatic that simplicity is one of the major ingredients of a reliable system. Yet, in spite of its widespread use, there is no well-accepted definition. Certainly, size is a factor. Given two implementations, we might say the smaller one is simpler, but it may require construction in a highly obtuse or inefficient manner while the larger one does not. Given two sets of operating system kernel primitives we might claim the smaller one is simpler but it may be quite difficult to perform ordinary tasks with it. Thus, an important requirement for a set of kernel primitives is to balance *simplicity* and *size*. Size can be measured by the number of primitives and the size of the kernel code. Simplicity is a more subjective metric: Programming effort for both kernel and application program implementation must be evaluated.

A distributed operating systems kernel should be well-suited to the needs of high-level distributed programming languages and sophisticated systems programs. It is not enough that it is *possible* to implement a given specification. The primitives provided should make for efficient programming or, at the least, should simplify construction of high-level language compilers.

If computers become cheap enough, it should be practical to allocate a single processor to each process and to each peripheral device on a network. With this *complete distribution* of activity, the overhead and complexity of multiprogramming is eliminated. As a result, interprocess communication efficiency becomes a key issue in determining system performance. Further, a kernel that supports only a single client process can be kept simple. One important thrust of this thesis is an exploration of the extent to which the restriction of one process per processor facilitates the design of a simple, powerful, and uniform kernel.

Traditional operating systems kernels will hide the details of memory management and device handling from the client. In a distributed environment these functions are not required by all clients. Further, in a heterogeneous network, the needs of each client for these services will vary widely from node to node. Thus, to be consistent with our kernel philosophy that only universally needed services be provided by the kernel, we exclude memory management and device handling capabilities from our kernel requirements.

1.2. Communications Adaptors

A *communications adaptor* (CA) is a processor that provides efficient intermachine message transport. An example is the Arpanet IMP [6]. However, most proposals for communications adaptors support only *intermachine* rather than *interprocess* communication (IPC). Support for IPC must include process naming facilities and synchronization mechanisms. These functions are typically provided in existing systems by high-level programming languages or operating systems kernels.

Spector [7], Mockapetris [8] and Leblanc [9] among others have recently pointed out the need for more powerful communications adaptors. In addition to

providing an IPC capability, a CA should meet the following requirements:

Uniformity
: A CA should permit different nodes to communicate by providing a standard network interface.

Reliability
: The CA must use a communications protocol that ensures reliable in-order transmission of messages.

Efficiency
: The CA should free the host processor from the burden of supplying reliable message transport protocols [6]. This is especially important for very high speed networks, as host performance may be severely degraded if the host must spend a large portion of its time managing the low-level details of communications protocols.

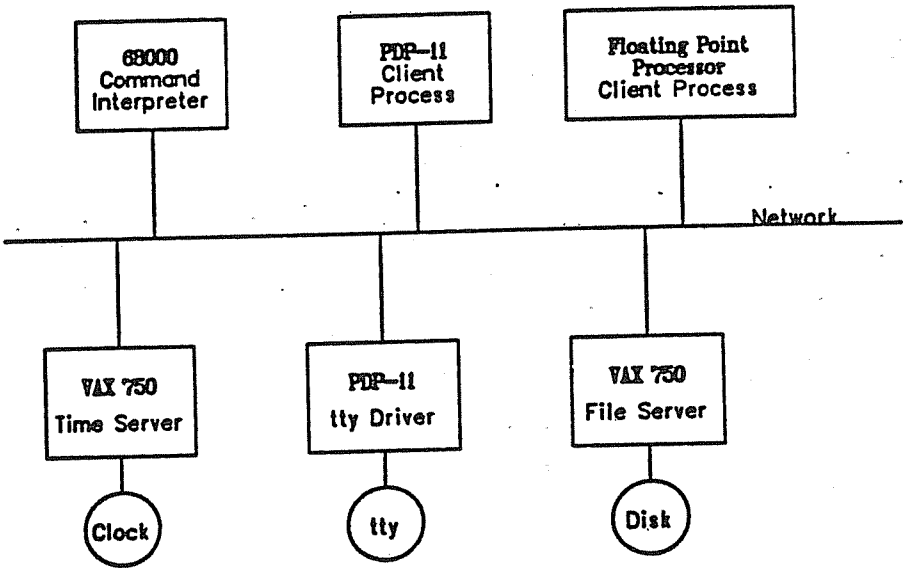
Low Cost
: A CA should be inexpensive so that even cheap processors may use one without adding significantly to the cost of the processor. As a result, the CA implementation should be compact so that a small processor (perhaps even a single chip) can perform the CA functions.

1.3. SODA

We have discussed our requirements for a distributed operating system kernel and for a communications adaptor. Our thesis is that it is possible to combine the two: We show that it is feasible to design a CA which provides operating systems support (at least in a uniprogramming environment) in addition to message handling capabilities. In addition, we show that this CA can be implemented compactly and simply enough that an inexpensive hardware realization (possibly in VLSI) should be possible.

In a typical SODA network, each processor possesses a SODA interface that provides IPC facilities. Some clients may provide a database service or a time service; some may control peripheral attachments such as disks or terminals; and still others may be available for executing application programs.

Typical SODA Network



A SODA interface provides a simple and powerful set of primitives that provide efficient IPC mechanisms. It will allow a node to function without further kernel support in a distributed environment. Thus, in addition to the ability to send messages reliably to another *machine*, SODA will support communication between *clients*.

Our work makes three major contributions to the field of distributed operating systems research:

(1) We show that an inexpensive communications interface can be built which supplies a set of primitives sufficient for processes to cooperate efficiently in a distributed environment without additional kernel support.

(2) We explore the kinds of communications primitives that are necessary to support high-level applications in a system consisting of networked *uniprogrammed* processors.

(3) Most message-based systems provide an active SEND and a passive RECEIVE primitive. The SODA primitives provide an active RECEIVE and an active EXCHANGE as well. Our implementation shows that these primitives can be implemented with approximately the same performance as an active SEND. As a result, new styles of interprocess communication are feasible.

In the remainder of this thesis, we present the design of SODA and support our claim that an inexpensive communications adaptor which provides IPC support can be constructed. We will also show that SODA is simple both to use and to implement.

Chapter 2 surveys related work. Chapter 3 describes the SODA design in detail. In chapter 4 we illustrate the power of SODA by presenting examples of how SODA can be used to build distributed programming language constructs and solve typical problems in the construction of a distributed operating system.

Chapter 5 supports our claim that SODA is simple by reporting on a compact and efficient experimental implementation. Chapter 5 also substantiates our claim that active RECEIVE and EXCHANGE can be implemented with performance comparable to active SEND. In chapter 6 we explain the reasons for some of the decisions we made in designing SODA and discuss possible extensions. Finally, chapter 7 summarizes our results and suggests directions for future research.

2. RELATED WORK

Inter-process communications facilities are provided by processors, operating systems kernels, and programming languages. Most provide mechanisms to create and destroy processes and to enable them to exchange data. In this chapter we will examine some other work in these three areas. Our survey is meant to illustrate common features found in local area networks and is not comprehensive.

2.1. Definitions

To assist in this discussion, we define some terms with the help of five taxonomies of parallel processing: Cashin [10], Spector [7], Andrews and Schneider [11], Bacon [12], and Finkel [13]. Plasmeijer [14], Goodenough [15], and Casey [16] also provide some useful terminology.

PROCESS ALLOCATION
Several processes may coexist on a single physical machine. If the number of processes allowed is one this is called *uniprogramming*, and if more than one process is allowed it is called *multiprogramming*.

PROCESS CREATION
Loading a program on a (foreign) processor may require that the processor being loaded be named *specifically* or it may be *anonymous*. The code that is loaded may be *active* (a process with its own locus of control) or *passive* (such as a procedure that may be called remotely).

SHORT TERM SCHEDULING
When there is a choice of selecting from a set of suitable processes, *priorities* may be provided to grant the more important processes better access to the CPU.

PROCESS INDEPENDENCE
When a process is created, it may have no logical connection with its parent (be *independent*) beyond obtaining some initial capabilities, or it may be *dependent* on its parent. For example, it may be required to inform the parent about its termination or to ask the parent for permission to perform certain functions.

PROCESS DESTRUCTION

A process may be terminated by another process or by its own volition. Termination may be *immediate*, asynchronously interrupting the process to be terminated, or *delayed* (the process to be terminated has a chance to clean up before dying). Notification to other processes about the death of another process may be *automatic* or *explicit* (the dying process must specifically inform others).

EXCEPTIONS

Exceptions (that may take the form of special messages) may be raised *synchronously* or *asynchronously* in a process and may or may not be propagated through a chain of ancestors of the process. Either the process raising the exception, the process receiving the exception, or both may be disrupted. There are three categories of exception handling [15]:

(1) ESCAPE exceptions that require termination of the operation raising the exception.

(2) NOTIFY exceptions that forbid termination of the operation raising the exception and require its resumption after the handler has completed its actions.

(3) SIGNAL exceptions that permit the operation raising the exception to be either terminated or resumed at the exception handler's discretion.

MESSAGE SYNCHRONIZATION

1) Non-Blocking (asynchronous) Send/Receive
The sender or receiver proceeds immediately after issuing a send or receive.

2) Simple Rendezvous (synchronization send)
Whichever party arrives first (at a send or receive) waits for the other. At that point, parameters are exchanged and both parties continue.

3) Extended Rendezvous (remote procedure call)
Extended rendezvous is the same as simple rendezvous except that the sender remains blocked after rendezvous until the receiver replies. The remote procedure call discussed in §4.2.2 is in this category.

NONDETERMINISM CONTROL

When several conditions can become simultaneously true, the corresponding actions may be handled in a *predictable* fashion or picked at *random*. Conditions may include the states of local variables, the contents of messages, and the synchronization of two parties engaged in rendezvous. *Guarded Commands* [17] are available as a mechanism for handling nondeterminism. Other proposals, such as alternation of paths in path expressions [18] are similar. Guarded commands may be used to select one of several send *(output guards)* or receive *(input guards)* requests. When non-determinism control is applied to the receiving of messages it is referred to as *selective receipt*.

TARGET GROUP SPECIFICATION (N-out/M-in)

N is the number of possible destinations a sending process can specify in a

single "send" primitive; M is the number of sources a receiving process can specify in a single "receive" primitive.

COMMUNICATION METHOD

Processes exchange data either by *sharing memory* or *exchanging messages*. In the former method, processes communicate by addressing memory. Synchronization methods aim at ensuring atomic data access. In the latter method, processes address each other. Synchronization methods aim at coordinating process activity. Either method may be used to emulate the other.

PROCESS NAMING

There are two essential ways in which processes may be addressed:

1) Facility Names

The sender need not know the exact name of the receiver but must know some attributes that describe the function of the receiver.

2) Explicit Names

The sender must specify an exact destination name. Names may be created dynamically or at compile time. Some means must be provided to give the sender access to a well-known name. The name may be *published* (available in a manual) or *discoverable* (available by making inquiries to a name server).

PORTS

A port is a queueing point for messages which is either bound to the name of a process or has its own independent identity. An *input port* is a port which may be written by several processes and read by one process. An *output port* is a port which may be read by several processes and written by one process. A *free port* is a port which may be read and written by several processes.

LINKS

A *link* combines the notion of *capability* [19] with that of a logical communications path. Additionally, an end of a link may be moved to another process transparently to the process at the other end of the link. Notifications about process and link destruction, restrictions on passing links to other processes, and limitations on the type of message sent on a link may be included.

MESSAGE SCREENING

Certain messages addressed to a client are never delivered to the client. This is because the kernel that receives incoming messages for demultiplexing among clients rejects some messages. Most systems provide for at least the rejection of messages not addressed to a particular processor. A rejection may also be based on client-supplied information such as size or type information bound to the message. The purpose of screening is to avoid interrupting the receiving client to handle messages that may more efficiently be processed by the kernel.

COMMUNICATION PATH
Once a logical communication path has been set up, it may allow only one-way conversation *(simplex)*, two-way conversation *(duplex)*, or it may allow an arbitrary number of parties to communicate *(party line)*.

MESSAGE PATHS
Messages may have to be *routed* by intermediate machines (not the final destination) in which case the cost of sending a message is not uniform. Alternatively, messages may be sent directly to the destination machine.

2.2. Programming Languages

We discuss here a variety of proposals for programming languages designed to be used in message-based distributed environments.

2.2.1. Communicating Sequential Processes (CSP)

CSP [20] achieves message synchronization by simple rendezvous. Target group specification is one-out/one-in; process naming is explicit by both parties in a communication. Guarded commands are used to control nondeterminism. Input guards may be used within guarded commands but output guards are not allowed (later implementations [21, 22] provide output guards). The lack of output guards complicates some algorithms because reading and writing are not symmetric. The explicit naming requirement makes it difficult to provide library services since such a service would have to know the name of its requester *a priori*. Messages are screened by the data types of the enclosed objects.

2.2.2. Distributed Processes (DP)

DP [23] is a proposal for a uniprogrammed system designed to handle real-time events. Target group specification is one-out/many-in as the receiver does not specify its callers and the sender explicitly names the receiver. Message synchronization is by extended rendezvous. Guarded commands and regions provide nondeterminism control. An initial statement is executed until it terminates or

waits for a condition within a guarded region to become true. At that point, external requests (equivalent to extended rendezvous) can be processed which possibly cause a condition in an earlier operation to become true. Screening on parameter types in a remote invocation is performed. The guarded command structure differs from Dijkstra's [17] in that if all guards fail, the process waits for a random guard to become true instead of skipping the statement. Because the sender is blocked until the operation it invoked completes, highly concurrent algorithms are difficult to express. Also, implementation of DP could be quite difficult since guards must be reevaluated in the context of their own containing scope (guarding conditions may include the state of local variables).

2.2.3. Input Tools Processes (ITP)

ITP [23, 14] is based on a language (Input Tool Model (ITM)) [24] designed especially for management of input-driven interactive programs. This model is similar to Path Expressions [18]. Program control is managed by a parser that matches "input rules" to determine the next procedure to call. When an input activates a "tool", that tool executes and passes control back up to the enclosing tool, informing that tool that it has successfully matched the pattern named by the enclosed tool. This activity continues until the outermost set of input rules is matched. A small number of pattern matching primitives are provided.

ITP builds on ITM by adding send and receive primitives. "Receive" specifies a message and the type of the message. "Send" gives a message that must match the type specification in the corresponding receive. To name a destination, "send" may use the process name and/or an entry name ("basic tool name"). A receiver may receive from any sender or a particular sender. Broadcast is achieved by omitting the process name in a send operation. Additionally, a form of

group broadcast may be specified ("interleaved process set"). Message synchronization is by simple rendezvous and target group specification varies from one-out/one-in to many-out/many-in. A form of guarded command ("selected process set") is also available. The parser in ITM is modified to manage the additional features provided in ITP. An exception handling mechanism in ITM ("escape tools") is being considered for use in ITP.

2.2.4. ADA

ADA [25] is a language not strictly designed for a distributed implementation. Message synchronization is by extended rendezvous; a form of guarded command is available to control nondeterminism and target group specification is one-out/many-in. Exceptions are not automatically propagated and only failure exceptions may be raised asynchronously. Exception handling is in the ESCAPE category. The sharing of memory and activation records among tasks (tasks may be nested) precludes a straightforward message-based communication unless the language is restricted.

2.2.5. *MOD

*MOD [26,27] is a proposal based, like ITP, on an existing language (Modula) [28] and provides a hybrid communication approach. Processes on a given machine may communicate via shared memory and provision of multiprocessors offers a chance to improve performance (but is not required). Non-local processes communicate via ports. Selective receipt is provided by *regions*. Ports provide either asynchronous or extended rendezvous communication. A process may be created to run in parallel with the parent, or the parent may wait for the process to complete and return a value (a *functional* process). The target

group specification is one-out/many-in. Processes are independent so strict bottom-up termination of a task hierarchy is not required. Short term scheduling flexibility is provided by assigning priorities to processes. *MOD has as a major design goal the ability to provide as much flexibility as possible (to avoid constraining the user to its design decisions).

2.2.6. Communication Ports (CP)

CP [29] is a uniprogramming proposal based on DP that adds two more features: a way to specify a set of acceptable senders, and a mechanism for the receiver to "disconnect" callers at will. The first feature implies that target group specification includes one-out/one-in (along with one-out/many-in as in DP). The second feature allows flexible scheduling of request handling. As a result, both simple and extended rendezvous are available. A wide class of scheduling problems can be solved with "disconnect", and less explicit interaction (e.g., asking permission to proceed after a send) may be required in some client-client communication protocols.

2.2.7. Extended CLU (ECLU) and ARGUS

ECLU [30,31] is similar to *MOD in many aspects. *Guardians* (a collection of processes that communicate via shared memory) are the fundamental entities in ECLU. A guardian communicates with another via strongly typed messages which are sent asynchronously. Guardians provide ports to receive messages on. If a receiving port runs out of room for a message, the message is simply discarded and higher-level protocols are used to determine the necessity of resending. Further, messages are not guaranteed to be received in the order sent.

ARGUS [32] is a later version of ECLU in which guardians supply (instead of ports) a new process invocation (*handler*) for each message receipt. Both ECLU and ARGUS provide mechanisms for ensuring reliability in the presence of crashes.

2.2.8. Plits

Plits [33] is a language proposal that includes both asynchronous sending and modest typechecking of messages. A message consists of a set of name-value pairs (slots) that are packaged and shipped out as a data record. Message screening mechanisms reject messages of improper structure, but type checking is not automatic so messages about which nothing is known can be received. In addition, a "transaction key" may be specified in a send or receive. The "transaction key" is some additional (network-wide unique) information the client supplies about a message that a receiver may inspect as a way of screening incoming messages. The target group specification is one-out/many-in but may be restricted to one-out/one-in with the transaction key.

2.3. Operating Systems

We discuss here some other distributed operating systems kernels. A kernel differs from a language primarily in the level of services provided. As an example, a programming language may provide remote procedure call as a primitive; however, it may be implemented on top of a kernel that provides other forms of message synchronization.

2.3.1. Arachne

Arachne [34] is a distributed operating system with capability-based communication (simplex links). Both asynchronous send and asynchronous receive

are possible, but the latter is restricted to a special "catcher" routine that has limited access to other service calls. Target group specification is one-out/many-in. A set of "channels" may be specified to select among senders that possess a link to the receiver.

Process creation is anonymous: a new process is loaded on any free machine. Arachne is a multiprogramming system with a non-preemptive scheduling mechanism. Processes are dependent. It is intended that each client speaks to a "resource manager" process to obtain standard operating systems services. A "connector" program that provides a semi-automatic service to bind process names of a set of cooperating processes together is available [35].

One aspect of Arachne is that many system facilities normally provided by the kernel can be provided as client programs. A disadvantage is the ad-hoc features necessary to support this goal (such as having clients with special powers to load processes via the filesystem). In addition, the ability to place restrictions on link use (such as disallowing link duplication) is often more of a hindrance to problem solution than a help and is not used much in practice.

2.3.2. Gaggle

Gaggle [36] supports a uniprogramming discipline along with the concept of links. Both synchronous and asynchronous send and receive are possible and may be freely mixed. Gaggle supports byte streams, and the kernel buffers bytes transported between the user and the system. A number of predefined links to system utilities are provided at process creation time. Gaggle is similar to Arachne except that Gaggle links support duplex communication.

2.3.3. Thoth

Thoth [37] is a message-based multiprogrammed system designed for portability. All messages are 8 words in length. The most relevant (to our discussion) feature of Thoth are its unique message-passing primitives. They are:

(1) rid := Send(mesg, pid)
A synchronous send. "mesg" is sent to the process named by "pid". When the message has been replied to (by the Reply primitive), the sender is unblocked and returned the process id ("rid") of the receiver that issued the Reply (need not be the same as "pid"). "mesg" will be overwritten by Reply.

(2) Receive (mesg, pid)
A blocking receive. The receiver blocks until a message from process "pid" arrives. Then the message is stored in "mesg". The pid need not be specified in which case any message addressed to the receiver is obtained.

(3) Reply (mesg, pid)
Reply to process "pid", unblocking that process and supplying "mesg" as a return value.

(4) Forward (mesg, fromid, toid)
A non-blocking directive that causes "mesg" to be sent to process "toid" but appear to have been sent by process "fromid". Process "fromid" must be currently blocked awaiting a Reply.

2.3.4. Charlotte

Charlotte [38] is, like Arachne, a link-based multiprogrammed system but in this case, links are full duplex. Charlotte is built on top of a *nugget* [*39*] which is a communications kernel. The nugget provides reliable machine to machine messages with asynchronous notification of the client (the Charlotte kernel) upon successful receipt and send completion. Selective receipt is not provided by the nugget.

In Charlotte, both sending and receiving are asynchronous but notification of completed sends or message receipt is synchronous. Charlotte also provides a way to cancel either a pending send or a pending receive operation.

2.4. Hardware Support

Some message-based systems are improved by hardware support. This support may take the form of providing dedicated microprocessors (which may be microcoded) to perform protocol tasks, specialized protocol chips, or processors with message-passing instructions incorporated into their native instruction set. We discuss here some proposals that provide hardware assistance for IPC.

2.4.1. Cm* and StarOS

Cm* [40] is a multiprocessor architecture that supports either shared memory or messages. A cluster of machines with local memory are interconnected via a *map bus*. Clusters themselves are interconnected by a *K-map*. The K-map is microcoded to provide (in conjunction with host software) the desired IPC primitives. A major consideration for using Cm* is the relative cost of accessing memory. Intercluster references are more time-consuming than intracluster references which in turn are more time-consuming than local references.

StarOS [41] is a capability-based operating system that executes on Cm*. It provides *mailbox* primitives for IPC. A mailbox is an object that can buffer a single data word or a single capability. Sending to and receiving from a mailbox is asynchronous. A process may associate *(register)* a signal with a given mailbox and block on that signal until receipt on that mailbox occurs. If a mailbox is full, a send to it fails immediately. Support for mailboxes is partially provided by microcode and partially by software.

2.4.2. Leo

Leo [42,43] is a system of eight uniprocessing computers designed to act as a single personal workstation. It employs a microprocessor (an Intel 8085) to

perform message handling tasks. Each processor (and hence, each process) is statically named by its function (i.e., the fileserver is on a fixed processor which always contains the fileserver process). One processor is allocated for applications programs; the remainder have pre-defined functions (e.g., editor, printer, tty-driver). Most forms of message synchronization including asynchronous send and rendezvous synchronization are possible.

The transport protocol works as follows:

(1) The requester client issues an *order* to a server client which is a message of a small fixed maximum length (150 bytes). Orders are sent asynchronously and are queued by the destination kernel. At most one uncompleted order between server and requester may exist so space for orders can be statically preallocated.

(2) The requester issues a WAIT for a *response*. When the response arrives, the order is said to have *completed*.

(3) The server client issues a WAIT command. The server then blocks until an order is available. WAIT may select a particular sender. A special kind of order is provided *(telegram)* which will interrupt the server.

(4) The server issues zero or more TRANSPORT commands that cause data to be transferred in either direction. The requester should not tamper with data being sent by a TRANSPORT command until the request completes.

(5) The server issues a *response* that, like an order, contains at most 150 bytes. A response may interrupt the requester when issued as a telegram by the server.

The size of an order is sufficient to handle most needs of the system and the TRANSPORT facility is used rarely (primarily for file transfer). Telegrams are currently used only for handling error conditions [44].

2.4.3. The Intel iAPX432

The iAPX432 [45] provides microcode support for a wide range of IPC mechanisms. The computer architecture supports distributed IPC in a unified model that treats all objects (messages, processes, data structures) in a similar, capability-based fashion (all object access is via an object descriptor that defines access rights to the object). The iAPX432 is a shared-memory system

that supports messages as a communication vehicle as do Thoth [37], and Demos [46]. Messages are sent via ports. Message synchronization is simple rendezvous: The sender is blocked until a receive is performed on the associated port. Both input and output ports are provided. A mechanism ("surrogate receive") is provided to receive on several ports by setting up a single port to that a group of ports can send. Port queues may be dynamically rearranged. Exceptions are classified by severity and are either treated as normal messages or emergency messages that interrupt the current process.

2.4.4. The Distributed Computing System (DCS)

In DCS [47,8,48] processors are connected in a ring. A ring interface provides the line driving mechanism which, in addition to managing low-level line protocols (e.g., bit stuffing and error detection), helps support the higher-level communication protocol. In particular, the interface, in conjunction with host software, provides process name screening based on associative pattern lookup, and a software sequencing protocol that eliminates duplicates. The interface itself provides four kinds of acknowledgements that assist the software protocols:

(1) The message was addressed to a nonexistent process.
(2) The message was successfully delivered to at least one process.
(3) The message was not accepted by any process but the address was recognized by at least one interface.
(4) The message was recognized by at least two hosts; at least one of which accepted the message and at least one of which did not accept the message.

Because process names are matched associatively, addressing is completely location independent and the process migration mechanism (but not necessarily the policy) is trivial. However, it is probably not a practical approach for building

either very fast or very inexpensive interfaces that must minimize the amount of logic used. Broadcasting can be achieved by using the same name for several processes. When a service is requested, that request is broadcast throughout the network and bids on the request are returned. Process names may be installed by the DCS kernel in the interface name table on behalf of clients.

The DCS system provides three IPC primitives:

(1) Send(processname, message)
Block the sender until "message" is delivered to the queue maintained by the destination process specified by "processname".

(2) Ctrl(processname, message)
Same as Send but deliver "message" to the DCS kernel.

(3) Recv(Buffer, Timeout)
Wait (or time out in "Timeout" clock ticks) for a message to appear on the local message queue and place the first message from the queue in "Buffer". If "Buffer" is NIL, return the status (empty/nonempty) of the message queue.

2.4.5. Intel Israel 82586

Intel [49] has developed a sophisticated ethernet controller that serves as a protocol co-processor. In this system, the 82586 chip shares memory and control lines with the CPU and provides a relatively high-level interface to the ethernet. Besides providing the standard line protocol functions such as bit stuffing/unstuffing, random backoff, collision detection and enforcement, it also provides channel-like buffer management and broadcast addressing capabilities.

The client may provide linked lists of commands that inform the 82586 of a list of buffers to be transmitted. Once the list is prepared, the buffers named in the command list are transmitted without client intervention. On receipt of a message, the 82586 automatically buffers the message and inserts it into a linked list of incoming messages maintained in client memory. A buffer of the size of the incoming message is allocated for each arrival. The client may specify that an

interrupt should be generated when a given number of frames have been received. The 82586 does not provide reliable transmission primitives, but it does provide powerful primitives useful for high speed data transmission.

2.4.6. TCP

TCP [50] is a communications protocol designed for packet switched networks to support hosts possessing quite different transmission rates and latencies (e.g., high-bandwidth high latency satellite and moderate-bandwidth low latency conditioned phone lines). Hosts will be quite diverse so the protocol must be provide a fairly low-level service that can be used by all clients. TCP is supported by a datagram service (IP) [51]. TCP provides reliable messages exchanged over a full-duplex path between two unique sockets (a socket identifies a client process).

TCP message primitives are connection-based (i.e., require an explicit connection to be established before communication can begin) and include:

(1) Open (foreign_socket, ...): connection_id
Open a connection to a foreign host. Open must be executed by both ends of a connection.

(2) Send (connection_id, buffer, ...)
Deliver "buffer" to a remote queue.

(3) Receive (connection_id, buffer, ...)
If no data is available, the receive request is queued and control returns to client; otherwise "buffer" is filled from the receive queue. In the former case, the client will be informed via interrupt when a message has arrived.

(4) Close (connection_id)
Half-close a connection. Messages will no longer be sent from the closing side but may still be received by it.

(5) The client is informed by software interrupt of events such as the availability of a message or a connection failure.

The main objections to using TCP in a local network are that connections are expensive to set up and tear down; a prohibitive cost in an environment where connections are frequently established and broken. Also, TCP provides many

features (such as windowing to handle high-bandwidth, high-latency message traffic) that are not required in a local network environment.

2.4.6.1. TCP Hardware

Quanta Microtique [52] has announced a TCP chip that supports a single TCP connection; several chips may possibly be "ganged" to provide multiple connections.

Mockapetris [8] has proposed a *filter* which is a highly specialized and very fast processor that, together with a microprocessor that has DMA access to host memory can perform most of the TCP protocol without host intervention.

2.5. Conclusions

We surveyed several proposals for the support of distributed programming. If an operating system is to support several distributed programming languages it must not restrict the style of interprocess communication. For example, an operating system that provides only synchronous SEND and RECEIVE primitives would not be well-suited to implement a language that supports asynchronous SEND and RECEIVE. On the other hand, if an operating system makes no design decisions and supports a wide class of IPC mechanisms, it becomes large and complex; certainly not in keeping with our stated goal of simplicity.

Available operating systems and hardware support so far has been geared to particular styles of programming; the needs of a system that supports several styles of IPC have not yet been completely met. LEO, for example, provides a flexible set of message passing primitives but the process naming mechanism supports only static interconnections. In the next chapter, we present the design of a communications interface (SODA) that can support a large class of IPC mechanisms and that has a small, unified set of primitives.

3. SODA PRIMITIVES

This chapter presents the SODA primitives. The purpose of this chapter is to explain what SODA is, not how it is used (chapter 4), how it is implemented (chapter 5) or why it is the way it is (chapter 6).

3.1. Definitions

CLIENT
: A process executing on a particular client processor.

KERNEL
: The SODA machine that supplies interprocess communications facilities to the client. The kernel and client communicate by shared memory. We use the terms "SODA kernel" and SODA interchangeably.

REQUESTER
: The client that initiates a data exchange.

SERVER
: The client that completes a data exchange requested by another. The same client may be a requester in one data exchange and a server in another.

HANDLER
: A section of client code that is executed in response to an interrupt generated by the kernel. When the handler is invoked it enters a BUSY state. When the handler completes it returns to an IDLE state.

TASK
: The main part of the client program that is always executing unless the handler has been invoked. When the handler completes, the task continues from the point of interruption.

BUFFER
: A descriptor that indicates the size and location of a contiguous region of shared (between client and kernel) memory.

MID
: A network-wide unique identifier (machine id) assigned to each node. A kernel will detect messages directed to the address specified by its MID or to all kernels (BROADCAST messages).

3.2. Architectural Assumptions

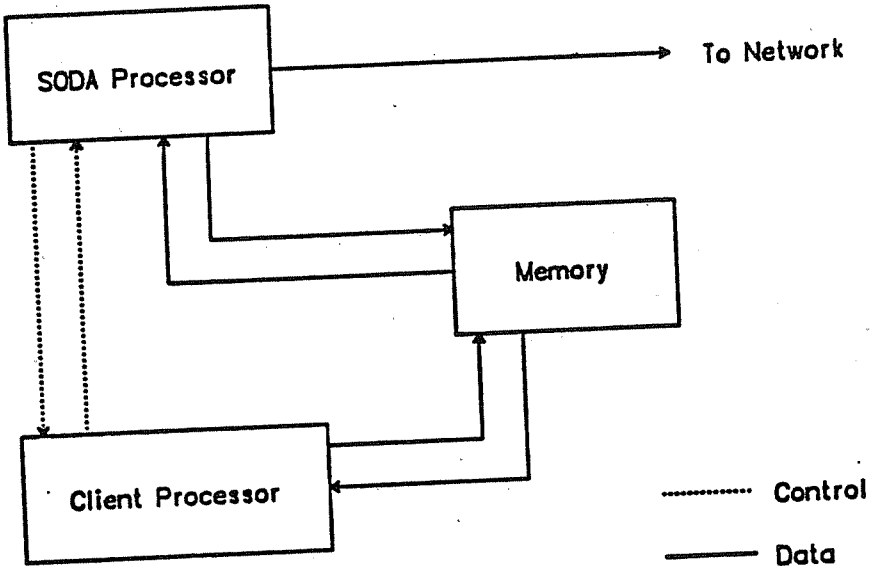
We assume the existence of a SODA network as shown in §1.3. Each node in the network is composed of two processors: a kernel processor and a client processor. The kernel processor provides the SODA primitives; the client processor executes an application program. The client processors need not be identical but the attached kernel processors must all provide the same functionality. Also, each client processor must have the same sized bytes (8-bits) so that message transfer between two machines always makes sense. Each kernel processor has a small amount of private memory for performing its own work as well as access to some parts of the client processor's memory.

The client processor may issue commands to the kernel via control lines, and the kernel will delay return of control from such a command until the command completes. On some machines (such as the PDP-11), a *busy bit* in a device register will be cleared to indicate command completion. The effect of this delay is to suspend client execution until the kernel completes the command (i.e., kernel commands are executed atomically). In most cases, the kernel returns as soon as it has noted the client command. Some SODA primitives may require a short, bounded interval before control is returned to the client. The kernel may also interrupt the client processor. SODA commands are atomic: The kernel will not interrupt the client when executing a command on behalf of the client. Every kernel has a connection to a high-speed, local area network with broadcast capabilities. The client can only access the network by interaction with its kernel coprocessor.

The client processor may be attached to peripheral devices. The operation of these devices is independent of the operation of the kernel. However, the interrupt mechanism used by the kernel may be shared with other peripheral dev-

ices. For example, on a PDP-11, the SODA kernel may interrupt the client by asserting an interrupt in the same way that a disk interrupt is issued.

SODA-Client Architecture



The SODA kernel provides ten primitives to the client. These include: request for bidirectional data transfer (non-blocking with asynchronous notification of the destination client); the data transfer itself (synchronous, with asynchronous notification of completion given to the requester); control of asynchrony by disabling/enabling asynchronous notifications; and a name service by which clients can advertise and discover services with private or well-known names. Additional mechanisms allow one client to terminate or initiate another.

The SODA kernel provides a compact set of primitives sufficient to enable an attached client to function as an active member of a community of clients. The features provided by the kernel fall roughly into three categories: *message passing primitives*, *naming facilities*, and *process control functions*.

3.3. Message Passing Primitives

The SODA kernel provides reliable point-to-point message exchange service which means that, in the absence of processor crashes, messages exchanged between two cooperating machines are guaranteed to arrive safely, in order, and without duplication. SODA makes some guarantees about behavior in the presence of crashes (discussed below). We assume that the kernel can detect errors due to transient subnetwork problems such as packet collisions or noise-induced errors and that a packet retransmitted enough times will eventually arrive undamaged.

Messages may vary in size from zero bytes up to a fixed maximum. Messages are only exchanged by distinct processors; there is no provision for local messages.

3.3.1. REQUEST

To initiate data transfer, the requester issues a REQUEST that specifies the SERVER SIGNATURE (defined in §3.4.1) of a destination server, a one-word argument, and two buffers. REQUEST is non-blocking: When it has been noted by the requester's kernel, a *transaction id (TID)* is returned to the client and the client is allowed to proceed. If the server named in the REQUEST does not exist, an error will be detected when the requester kernel attempts to process the request and the requester's handler will be invoked with an UNADVERTISED error status. TID's issued on a given client processor are guaranteed to be unique. The pair <MID, TID> (called the REQUESTER SIGNATURE) uniquely identifies a request across all time throughout the entire network.

The requester's kernel establishes a connection with the server's kernel and informs the latter (via the server's handler) that a request has been made. The server's handler is provided with an indication that a REQUEST has been made, the REQUESTER SIGNATURE, the particular SERVER SIGNATURE used by the REQUEST, the argument, and the buffer sizes supplied by the REQUEST. The server need take no immediate action on this request and may simply exit the handler.

3.3.2. ACCEPT

Eventually, the server may decide to honor or "complete" the request. It does this by issuing an ACCEPT which causes data to be exchanged and the requester's handler to be invoked. ACCEPT specifies a REQUESTER SIGNATURE, a one-word argument, and two buffers. When the ACCEPT completes, the contents of the first buffer supplied by the associated REQUEST will be transferred to the first buffer supplied by the ACCEPT and the contents of the second buffer sup-

plied by the ACCEPT will be transferred to the second buffer supplied by the REQUEST. In addition, the argument will be provided to the requester's handler. Zero-length buffers may be specified to inhibit data transfer in one or both directions. The REQUEST is termed a PUT, a GET, an EXCHANGE, or a SIGNAL, depending on whether data is transferred from requester to sender, from sender to requester, both ways, or neither way, respectively. If the ACCEPT encounters a requester which has crashed, a CRASHED status is immediately returned and the server is unblocked.

The kernel blocks the server until the data exchange is complete and then returns the server a status code indicating the success or failure of the ACCEPT. The requester is informed via its handler when the ACCEPT has completed and is told how much data was transferred in each direction by the ACCEPT. ACCEPT blocks until data exchange is complete but it is guaranteed to complete within a bounded amount of time. If an ACCEPT encounters a BUSY or CLOSED requester handler (§3.3.4), then the ACCEPT will complete and the handler interrupt will be queued by the requester's kernel.

The remainder of this section discusses some of the more subtle details of the message-passing primitives.

(1) Results are unpredictable if the requester accesses the buffers given in a REQUEST before it is notified of request completion.

(2) There is no way for a server to inspect the first buffer before sending the second in a single ACCEPT.

(3) REQUESTS issued from the same requester to the same server will always be delivered (but not necessarily ACCEPTED) in the order they are issued.

(4) The effect of passing overlapping buffer arguments to either ACCEPT or REQUEST is undefined.

(5) Only a fixed number (MAXREQUESTS) of uncompleted requests from any one requester are permitted. The REQUEST fails if more than MAXREQUESTS requests remain uncompleted.

(6) An ACCEPT will fail (and be returned a CANCELLED status) if issued by a different client (on a different processor) than that named in the matching REQUEST. Thus a client may not attempt to ACCEPT a REQUEST it did not receive by guessing REQUESTER SIGNATURES.

If a REQUEST encounters a handler that is CLOSED and/or BUSY in the server, the requester's kernel will keep trying to deliver the request. After a REQUEST has been delivered, it is monitored by the requester's kernel. If the server should crash, the REQUEST will complete with a failure indication. A client that loops forever inside its handler or never opens its handler is not considered to have crashed. The SODA kernel provides a means to terminate such clients (§3.5.3).

3.3.3. CANCEL

If the requester tires of waiting for a REQUEST to be ACCEPTED, it may CANCEL the request. CANCEL may block the requester for a short (bounded) time. CANCEL returns a status code indicating whether the CANCEL was successful or not. CANCEL fails whenever the REQUEST completed (for any reason, including REQUEST failure) before the CANCEL could take effect. In other words, if the handler is invoked to indicate the completion or failure of a given request, cancelling that request will always fail. CANCEL will delay the requester only long enough to ensure that completion is not imminent and to guarantee that a server

attempting to ACCEPT a cancelled REQUEST will be informed of the cancellation.

3.3.4. OPEN, CLOSE, ENDHANDLER

When the handler is invoked, control passes temporarily from the TASK (§3.1) to the handler. Handler invocations do not nest. If a request completion encounters a BUSY or CLOSED handler, the handler interrupt is delayed until the handler is IDLE and OPEN, but the server is not delayed by issuing an ACCEPT to a BUSY or CLOSED requester. As long as a REQUEST has a valid address, the requester's kernel will keep trying to deliver the REQUEST to a server with a BUSY or CLOSED handler. ENDHANDLER is used by the client to signal the termination of the handler. The handler and task share portions of the same memory. To provide synchronization, the client may use the CLOSE primitive to disable the handler and the OPEN primitive to reenable it. The handler is only invoked if it is OPEN and not currently executing (IDLE). OPEN and CLOSE may be issued from within the task or the handler. If executed from within the handler, they have no visible effect until ENDHANDLER is executed.

To summarize the uses of the handler: it is invoked to inform a server of an incoming REQUEST and it is also invoked to inform a requester of REQUEST completion. The handler is either BUSY or OPEN/IDLE or CLOSED/IDLE. The handler may only be invoked when OPEN and IDLE.

3.4. Naming Facilities

We discuss in this section the ways in which one client may address another in a REQUEST.

3.4.1. ADVERTISE, UNADVERTISE

A server may ADVERTISE one or more PATTERNS to which it will respond. It may also UNADVERTISE a previously ADVERTISED pattern. A PATTERN is simply a bit string containing PATTERNSIZE bits. PATTERNSIZE is large enough that the number of available distinct patterns is effectively inexhaustible given reasonable assumptions about the rate at which new patterns are generated.

The server name used in a REQUEST is called a SERVER SIGNATURE and consists of a <MID, PATTERN> pair. If, when a request is received by a server, the pattern in the server signature does not match any advertised pattern, the request is completed by the server's kernel and the requester's handler is invoked with an indication that the request failed. If the pattern matches and the server handler is OPEN and IDLE, the server handler is invoked and supplied the pattern used in the SERVER SIGNATURE, the REQUESTER SIGNATURE, the argument, and the sizes of the buffers given by the REQUEST.

The requester is not informed about a *successful* match until the server issues a ACCEPT. Once a REQUEST has been delivered to the server handler, screening on the pattern is no longer applied. Thus, UNADVERTISE on a pattern will not affect a REQUEST that has arrived at the server handler but not yet been ACCEPTED.

3.4.2. GETUNIQUEID

GETUNIQUEID supplies the client with a pattern that is unique with respect to all invocations of GETUNIQUEID throughout the entire network. Ideally, unique ids are generated at random (see discussion in §5.4). There are no restrictions on the distribution of patterns; clients are free to inform other clients of the existence of a pattern. The pattern returned by GETUNIQUEID must contain less

than PATTERNSIZE bits. This permits clients to use preassigned well-known names containing defined fields by reserving a bit in the pattern for distinguishing between random patterns (generated by GETUNIQUEID) and well-known patterns.

It is perfectly valid for several clients to ADVERTISE the same pattern. Using GETUNIQUEID to create all patterns ensures that no unintentional duplicates exist. SODA makes no requirement about the distribution of patterns created by GETUNIQUEID, so it may be possible for malicious clients to guess patterns advertised by other clients fairly easily. Any security mechanism should rely instead on the unforgability of the requester MID seen by the server and the unforgability of REQUESTER SIGNATURES (§3.3.1) in ACCEPTS.

3.4.3. RESERVED and CLIENT PATTERNS

There are two classes of patterns (distinguished by a bit in the pattern). One class is primarily used by SODA itself, and may not be either advertised or unadvertised by the client. The other class is for client use. The former (RESERVED PATTERNS) are bound to routines built into the kernel, and execution of these routines cannot be impeded by the client handler state. The latter (CLIENT PATTERNS) are bound by the client via ADVERTISE.

3.4.4. DISCOVER

When a requester knows a pattern advertised by one or more servers, but not the identity of a specific server (its MID), it may use the special BROADCAST identifier in place of the MID in the server signature. Such a request is broadcast to each node in the network and interpreted by the associated kernel. If the pattern in the server signature has been advertised by the server, the MID of the server is returned to the requester's kernel. After sufficient time for all nodes to

respond has elapsed, the requester's kernel returns a list of MIDS that matched the pattern to the requester client. This kind of request (DISCOVER) is semantically equivalent to a GET from the point of view of the requester. The second buffer supplied in the REQUEST is used to store MIDS (up to the number that will fit in the buffer) and the requester handler is invoked to announce the completion of the request. DISCOVER is completely transparent to server clients: no information about a DISCOVER is ever presented to a client. SODA makes no reliability guarantees about DISCOVER's.

3.5. Process Control Functions

In this section we discuss mechanisms for creating and terminating client processes.

3.5.1. DIE

When a client wishes to terminate, it executes the DIE command. This command causes the kernel to reset its internal state and to clear all advertised client patterns. Uncompleted REQUESTS which were issued to a client that subsequently crashed or died via the DIE command will fail with a CRASHED return status.

3.5.2. Booting

After a client has died, the node is eligible to receive a new client. The kernel advertises one or more RESERVED PATTERNS (BOOT PATTERNS) that are indicative of the type of client processor and attached peripherals. Clients may DISCOVER the MID of machines of that type and then attempt to boot a particular machine by using the boot pattern and the newly-discovered MID. The requester

issues a GET that uses the signature <MID, BOOT_PATTERN>. If the server machine is still available, SODA UNADVERTISES the BOOT PATTERN, creates a new pattern via GETUNIQUEID (the LOAD PATTERN), converts the LOAD PATTERN into a RESERVED PATTERN, and ADVERTISES the LOAD PATTERN. Finally, the LOAD PATTERN is returned as the value of the GET.

Any client attempting to use the BOOT PATTERN in a REQUEST once it has been UNADVERTISED receives an immediate completion interrupt indicating a failure. SODA guarantees that the LOAD PATTERN will never conflict with any of the preassigned RESERVED PATTERNS (the BOOT PATTERNS and KILL PATTERN).

If a client is successful in obtaining a LOAD PATTERN, it may begin booting using the <MID, LOAD PATTERN> signature in a series of PUT's that are ACCEPTED by SODA. Contiguous client memory is used for the ACCEPT buffers. A SIGNAL (§3.3.2) using the LOAD PATTERN informs SODA to start the newly-booted client executing in its handler. When that handler completes and executes ENDHANDLER (§3.3.4), the new client begins executing its task. A second SIGNAL using the LOAD PATTERN will terminate the server regardless of its handler state. This mechanism allows the parent of a process to kill it at any time.

3.5.3. Killing

When the client dies, the BOOT PATTERNS are advertised again. SODA also has advertised at all times a KILL PATTERN. A client issuing a SIGNAL using the signature <MID, KILL PATTERN> will cause the client associated with MID to die. The KILL PATTERN is used to stop runaway clients and will cause client death regardless of whether its handler is available or not.

The kill action bound to the LOAD PATTERN is identical to the action bound to the KILL PATTERN. The difference is that the KILL PATTERN is a well-known pat-

tern that is distributed only to privileged clients, whereas the LOAD PATTERN may be used by the parent of a client or other clients informed by the parent. The LOAD PATTERN is created anew with each new client instantiation.

Some nodes may contain ROM bootstrap programs. Pressing a RESET button causes a core image to be loaded, the BOOT PATTERN to be unadvertised, and execution to begin in the handler.

3.5.4. Changing RESERVED PATTERNS

The machine with MID 0 has the privilege to alter the reserved KILL and BOOT patterns on all nodes in the network. Each kernel has a built-in handler bound to the reserved pattern SYSTEM. This handler will only accept REQUESTS from machine 0. The argument in the REQUEST specifies the action to be taken:

(1) Add a new BOOT PATTERN.

(2) Delete an existing BOOT PATTERN.

(3) Replace the KILL PATTERN.

3.6. Crash Semantics

There is no special mechanism needed to reestablish communications with a processor that crashes and reboots. No explicit connection between server and requester is ever set up. All that is required to send a message to a client is a valid SERVER SIGNATURE. In this section we discuss SODA behavior in the presence of client processor crashes.

3.6.1. Failed REQUEST and ACCEPT

Whenever the client processor crashes in such a way that SODA can detect the crash (by dying, being killed, or entering a hardware state which indicates a

hardware failure and that the kernel can detect), kernel state is lost. REQUESTS uncompleted by the crashed client will cease to be monitored and servers attempting to ACCEPT a REQUEST issued by a crashed client will be informed about the crash by the return value of ACCEPT. Further, if the crashed requester processor is rebooted, "stale" ACCEPTS for REQUESTS left uncompleted as a result of a crash will fail and a CRASHED status will be returned to the server.

A REQUEST will fail and be returned a status of CRASHED if the server crashes at any point before the server issues an ACCEPT. Similarly, an ACCEPT will fail and be returned a CRASHED status if the requester has crashed *before* the ACCEPT is issued. Once an ACCEPT is issued, no guarantees are made about informing the server about requester crashes or the requester about server crashes. CRASHED is distinct from CANCELLED that is returned to a server attempting to ACCEPT a CANCELLED request.

Any attempt to ACCEPT a completed REQUEST will result in a CANCELLED status returned to the server issuing the ACCEPT unless the requester client crashes and recovers before the ACCEPT is issued. In this case, CRASHED is returned. A client which executes DIE is treated as a crashed processor.

3.6.2. Probes

REQUESTS are monitored by a *probing* mechanism after delivery to the server handler. A *probe* is a short message sent periodically by the requester's kernel to verify that the server is still alive. If several successive probes fail, a crash is reported. It is not possible for a server to crash and reboot in such a way as to escape detection by the probing mechanism.

3.7. Summary of the SODA Primitives

3.7.1. Kernel-Client Interface

The kernel has access to all buffers used in REQUESTS and ACCEPTS, memory available for client core images (used during booting), and a "communications region" that is used for transmitting arguments and status information between kernel and client.

3.7.2. Naming

The SERVER SIGNATURE is the pair <MID, PATTERN>.
The REQUESTER SIGNATURE is the pair <MID, TID>.
RESERVED PATTERNS are bound to routines in the kernel; CLIENT PATTERNS are bound to routines in the client.

3.7.3. Addressing Primitives

1. ADVERTISE (CLIENT PATTERN)
2. UNADVERTISE (CLIENT PATTERN)
3. GETUNIQUEID() : CLIENT PATTERN

PATTERNS contain PATTERNSIZE bits.
GETUNIQUEID returns a pattern containing less than PATTERNSIZE bits.

A client may discover its own MID by examining the location MY_MID in the communications region.

3.7.4. Message Passing Primitives

4. REQUEST (SERVER_SIGNATURE, ARGUMENT, PUT_BUFFER, GET_BUFFER):
   TID

If MAXREQUESTS remain uncompleted, a REQUEST is ignored by the kernel. It is the responsibility of the client to count the number of uncompleted REQUESTS. ARGUMENT is a single word supplied to the server handler upon handler invocation.

5. ACCEPT (REQUESTER_SIGNATURE, ARGUMENT, GET_BUFFER, PUT_BUFFER):
   STATUS

STATUS is any of (SUCCESS, CANCELLED, CRASHED).
ARGUMENT is a single word supplied to the requester handler upon REQUEST completion.

6. CANCEL (REQUESTER_SIGNATURE) : STATUS

STATUS is any of (SUCCESS, FAIL).

3.7.5. Handler Control Primitives

7. OPEN()
8. CLOSE()
9. ENDHANDLER()

The handler must be OPEN and IDLE in order to be eligible for execution. ACCEPT completion interrupts will be queued by the requester's kernel if the requester's handler is unavailable when the ACCEPT is issued. REQUEST interrupts are not queued by the server's kernel. If the handler is BUSY when a REQUEST arrives, the REQUEST will be retried by the requester kernel. As long as queued completion interrupts are present, the handler is considered BUSY. If client C_1 issues an ACCEPT followed by a REQUEST to another client C_2 the ACCEPT will cause an invocation of C_2's handler before the REQUEST will. The client may execute any SODA primitive, including ACCEPT, within the handler.

3.7.6. Handler Arguments

In addition to an argument indicating the reason for the handler invocation (REQUEST_COMPLETE, REQUEST_ARRIVAL, BOOTING), the following arguments are provided to the handler:

(1) On incoming REQUESTS, the REQUESTER SIGNATURE as well as information supplied with the REQUEST: the PATTERN part of the SERVER SIGNATURE, the REQUEST ARGUMENT, and the sizes of the two buffers.

(2) On request completion, the nature of the completion (CRASHED, SUCCESS, UNADVERTISED), the ACCEPT ARGUMENT, the REQUESTER SIGNATURE of the REQUEST being completed and the amount of data transferred in each direction by the corresponding ACCEPT.

(3) On booting, the handler is provided with the MID of the parent client and is in the OPEN state.

3.7.7. Process Control Primitives

10. DIE()

3.7.7.1. Reserved Patterns Interpreted by the Kernel

1. KILL_PATTERN
2. BOOT_PATTERN(S)
3. LOAD_PATTERN

The KILL_PATTERN and BOOT_PATTERNS are bound at SODA creation time; the LOAD_PATTERN is bound at boot time.

4. EXAMPLES

In this chapter, we present examples of SODA applications. We show how SODA is used to implement typical operating system utilities and distributed programming language constructs. This chapter begins with a discussion of a programming language which makes SODA more convenient to use. We next consider how SODA can be used to construct higher level communication facilities; present some scenarios for SODA use in a distributed operating system; and illustrate uses of SODA with programmed examples.

4.1. SODAL: a Programming Language for SODA

SODAL borrows from familiar programming languages such as MODULA [28], C [53], and Pascal [2]. A SODAL program is divided into three parts: INITIALIZATION, HANDLER and TASK procedures. The INITIALIZATION procedure is actually the handler invoked during booting. The HANDLER procedure is the client-provided interrupt handler invoked on REQUEST arrival or completion. An ENDHANDLER call is implicit at the end of the INITIALIZATION and HANDLER parts. A DIE call is implicit at the end of the TASK procedure. The client may supply other procedures as well. Variables are either local to a unique procedure or global to all procedures.

Skeleton SODAL Program

```
-- Predefined types:

-- MACHINE_ID, TRANSACTION_ID, PATTERN are defined by the kernel
-- implementation.
-- type STATUS = (REQUEST_COMPLETION, REQUEST_CRASHED,
--                REQUEST_UNADVERTISED, REQUEST_ARRIVAL, REJECTED)
-- REJECTED is discussed in §4.1.2.
-- (BOOTING is also a STATUS value, but the Initialization section
-- is provided for the handler invocation that has a Status of BOOTING)
-- The constant OK is used in REQUESTS and ACCEPTS for a default
-- argument when the argument is not specified by the client.

type REQUESTER_SIGNATURE =
   record
      Mid : MACHINE_ID;
      Tid : TRANSACTION_ID;
   end;
type SERVER_SIGNATURE =
   record
      Mid : MACHINE_ID;
      Patt : PATTERN;
   end;

<global declarations>


Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);

<local declarations>
   begin
      <handler code>
   end;

Initialization (Parent_MachineId: MACHINE_ID)

<local declarations>
   begin
      <initialization code>
   end;

Task()
<local declarations>
   begin
      <task code>
   end;
```

4.1.1. Blocking and Non-Blocking REQUESTS

SODAL provides the type BUFFER which is a record consisting of the address of a buffer and its size.

```
type BUFFER =
        record
          Address : integer;
          Size : integer;
        end;
```

Objects are coerced into BUFFERS as necessary. SIGNAL, PUT, GET and EXCHANGE are variants of REQUEST supplying the necessary buffers (as defined in §3.3.2).

```
SIGNAL (srv : SERVER_SIGNATURE; Arg : integer):
   TRANSACTION_ID;
PUT (srv : SERVER_SIGNATURE; Arg : integer; outbuf: BUFFER):
   TRANSACTION_ID;
GET (srv : SERVER_SIGNATURE; Arg : integer; inbuf: BUFFER):
   TRANSACTION_ID;
EXCHANGE (srv : SERVER_SIGNATURE; Arg : integer;
        inbuf,outbuf: BUFFER): TRANSACTION_ID;
```

A *blocking* REQUEST is a REQUEST which does not allow the requester to proceed until the REQUEST has been completed by the server (with data transfer unless the REQUEST is a SIGNAL). To provide blocking request primitives the SODAL compiler generates code as shown in the following example. The resulting blocking primitives are called B_SIGNAL, B_PUT, B_GET, and B_EXCHANGE. When a blocking REQUEST completes, the Status (REQUEST_COMPLETED, REQUEST_CRASHED, REQUEST_UNADVERTISED) associated with the request completion is returned. When a blocking primitive is issued from within the handler, SODAL requires the ability to modify the saved program counter (PC) that is normally installed upon handler exit. This allows SODAL to return the client to an alternate point in the

task which will idle until the blocking REQUEST completes, at which time the client PC is reset to point to the code just after the blocking REQUEST. If the client is not permitted access to the PC, SODAL will not allow blocking REQUESTS to be issued from within the handler because such REQUESTS must necessarily deadlock (there is no way to receive a request completion while BUSY in the handler).

Implementation of B_PUT

```
-- the following variables are hidden from the SODAL programmer:
var
   tid : TRANSACTION_ID;
   CompletionAwaited, done: Boolean;
   retpc, intpc: integer;
   status: STATUS;

Task()
begin
   B_PUT (..., OK, message); -- invocation of SODAL primitive
end;

B_PUT (server: SERVER_SIGNATURE; arg: integer; buf: BUFFER): STATUS
begin
   done := FALSE;
   tid := PUT(server, arg, buf);
   if inHandler then              -- blocking REQUEST within handler
      retpc := RETURNPC();        -- save point where B_PUT would return
                                  -- in current handler invocation
      cleanstack();               -- make stack look like it did before
                                  -- call of B_PUT
      intpc := END_INT_PC;        -- save point in task where handler normally
                                  -- returns to after ENDHANDLER is executed
      END_INT_PC := wait;         -- alter point of return after ENDHANDLER
                                  -- so it is possible to field completion
                                  -- interrupt
      inHandler := FALSE;
      CompletionAwaited := TRUE;
      ENDHANDLER();               -- Enter wait state; only completion
                                  -- interrupts can be handled now.
   fi;
 wait:
   while (not done) do
      idle(); -- idle is a procedure that does nothing. We discuss
              -- idle in §5.2.1.
   end;
   return (status);
end;
```

```
Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
begin
   InHandler := TRUE;
   if (Status <> REQUEST_ARRIVAL) and (Asker.Tid = tid) then
      done := TRUE;   – PUT was ACCEPTED
      status := Status;
      if CompletionAwaited then
         CompletionAwaited := FALSE;
         END_INT_PC := intpc;    –restore return point to where it
                                 –was at time of blocking REQUEST
         goto retpc;             –next statement after blocking REQUEST
      fi;
   fi;
   InHandler := FALSE;
end;
```

For the server, SODAL provides ACCEPT_SIGNAL, ACCEPT_PUT, ACCEPT_GET, and ACCEPT_EXCHANGE to complete SIGNAL, PUT, GET and EXCHANGE or their blocking relatives. These are implemented as ACCEPTS with NIL buffers (with size 0) filled in for the missing buffers (if any).

```
type ACCEPT_STATUS = (SUCCESS, CANCELLED, CRASHED);
ACCEPT_SIGNAL(requester: REQUESTER_SIGNATURE, Arg: integer):
   ACCEPT_STATUS;
ACCEPT_PUT(requester: REQUESTER_SIGNATURE; Arg: integer; outbuf: BUFFER):
   ACCEPT_STATUS;
ACCEPT_GET(requester: REQUESTER_SIGNATURE; Arg: integer; inbuf: BUFFER):
   ACCEPT_STATUS;
ACCEPT_EXCHANGE(requester: REQUESTER_SIGNATURE; Arg: integer;
   inbuf, outbuf : BUFFER): ACCEPT_STATUS;
```

4.1.2. Exception Handling

If the client uses the primitives incorrectly (e.g., issues more than MAXREQUESTS non-blocking REQUESTS) or gets back an unexpected result (e.g., CRASHED), a SODAL exception handler may be invoked. Typically, an exception handler will make an intelligent attempt at reexecuting the failed command: For example, a non-blocking REQUEST issued when MAXREQUESTS REQUESTS are already pending can be postponed until some pending request is completed. This strategy will work most of the time but will fail if a server which can complete a pending REQUEST must first receive the postponed REQUEST. The CRASHED exception may be handled by retrying after some delay or terminating the client.

For naive clients, library exception handlers might be suitable. In some applications, however, the client may require highly-specialized exception handling tools. A more detailed discussion of error recovery is beyond the scope of this thesis. The interested reader is referred to [15,25].

SODA permits the server to ACCEPT with a smaller buffer than REQUESTED which could cause invocation of an exception handler but is not necessarily an indication of an error. For example, in reading a file using GET, the requester may be provided with a partially-filled final chunk. To disambiguate errors from normal returns, the argument given with ACCEPT may be used as an error indicator. In the preceding example, if there is an error reading the file, the server should indicate that all is not well by supplying an appropriate argument. We will follow a convention that negative arguments denote error conditions. In SODAL, the REJECT statement is provided to handle typical error cases. REJECT is implemented as an ACCEPT with both buffers NIL and an argument of -1. REJECT will cause the requester handler to be invoked but no data will be transferred. REJECTED REQUESTS will be returned a value of REJECTED.

A frequent use of ACCEPT is to complete the REQUEST that has just arrived (i.e., caused the current handler invocation). For this purpose, SODAL provides ACCEPT_CURRENT which issues an ACCEPT for the present REQUEST. ACCEPT_CURRENT is implemented as an ACCEPT with the REQUESTER SIGNATURE of the present requester filled in. It is illegal to use ACCEPT_CURRENT outside of the handler (because a new REQUEST could arrive before ACCEPT_CURRENT is executed, leading to semantic ambiguity).

```
ACCEPT_CURRENT_SIGNAL(Arg: integer) : ACCEPT_STATUS;
ACCEPT_CURRENT_PUT(Arg: integer; inbuf: BUFFER) : ACCEPT_STATUS;
ACCEPT_CURRENT_GET(Arg : integer; outbuf: BUFFER) : ACCEPT_STATUS;
ACCEPT_CURRENT_EXCHANGE(Arg: integer; inbuf,outbuf : BUFFER) :
    ACCEPT_STATUS;
```

4.1.3. Naming Constructs

The broadcast facility of SODA is made more convenient through the DISCOVER primitive provided by SODAL.

DISCOVER (Patt: PATTERN) : SERVER_SIGNATURE

DISCOVER takes a pattern as an argument and returns a SERVER SIGNATURE. It is implemented by using the special BROADCAST identifier in a GET, with a buffer big enough to contain one response. DISCOVER will block until a response is obtained. More sophisticated clients who require a list of responses or who can take alternative action if no responses arrive may use the SODA primitives directly.

Casts are available to convert tid-mid pairs into REQUESTER SIGNATURES and mid-pattern pairs into SERVER SIGNATURES:

<mid, tid> — *cast the mid, tid pair into a REQUESTER SIGNATURE*
<mid, pattern> — *cast the mid, pattern pair into a SERVER SIGNATURE*

A literal pattern is represented as an octal number prefaced by a "%" mark:

```
const Well_Known_Pattern = %0345;
```

4.1.4. Queueing Constructs

Because SODA provides no queueing of messages, it is very common for the server client to do its own queueing. SODAL supplies queueing operations to facilitate writing server code.

SODAL provides the QUEUE type. The client specifies the length of the queue and the type of object it will contain. Other languages, such as SIMPAS [54] provide similar queueing constructs.

var Name_Queue : QUEUE [3] of REQUESTER_SIGNATURE

declares Name_Queue to be a queue of three elements, where each element is a REQUESTER SIGNATURE. There are six operations supported for objects of type QUEUE:

EnQueue(object, queue)	*– insert "object" at end of queue "queue"* *– raise exception if there is no room on the queue.*
DeQueue(queue): object	*– remove and return object at head of "queue"* *– raise exception if queue empty.*
IsEmpty(queue): Boolean	*– return TRUE if queue is empty, else FALSE*
IsFull(queue): Boolean	*– return TRUE if queue is full, else FALSE*
AlmostEmpty(queue): Boolean	*– return TRUE if queue has a single element left*
AlmostFull(queue): Boolean	*– return TRUE if queue can hold one more item*

4.1.4.1. Entries and Completions

The server signature serves as an *entry point* to the server. The data that are available to the server upon handler invocation (request signature, buffer sizes, etc.) are called the *tag*. The server makes a scheduling decision based on the tag information which determines when the ACCEPT of a REQUEST will take place. A *completion* occurs when the requester is informed that a REQUEST has been ACCEPTED.

SODAL provides a syntax for manipulating entries and completions. Inside the handler, the client may use ENTRY and COMPLETION inside case statements as follows:

```
case ENTRY of          -- process incoming request
  pattern_1: begin ... end;
  ...
  pattern_k: begin ... end;
esac;
case COMPLETION of  -- a request has been completed
  tid_1: begin ... end;
  ...
  tid_n: begin ... end;
esac;
```

where ENTRY and COMPLETION are bound to the Invoked_Pattern and Tid field of the incoming REQUESTER SIGNATURE respectively. The case labels may be variables or constants. If the handler is invoked for a request arrival, only the ENTRY case will be considered; if the handler is invoked for a request completion, only the COMPLETION case will be eligible for execution. Another possibility would be for SODAL to provide the illusion of separate handlers for ENTRY and COMPLETION interrupts.

4.2. Higher Level Communications Facilities

In this section we discuss how SODA can be used as a basis for constructing higher level IPC primitives.

4.2.1. Ports and Priority Queues

An *input port* is a queueing point (and a name to refer to it) for incoming messages. A number of different processes may write data to the port; a single process reads data from the port. *Priority Queues* are input ports that allow for priority scheduling of incoming messages. The implementation of input ports and priority queues is straightforward in SODAL. The server providing the port advertises the name of the port and enters a polling loop in the task. Port customers issue B_PUTs to write on a port. When a request arrives which uses that name,

the server handler enqueues the requester's signature. The polling loop tests if the queue is non-empty. If so, the first requester from the queue is removed and ACCEPTED. If the queue for holding request signatures becomes full, the server handler will CLOSE its handler until the queue is no longer full.

In SODA, ports are useful to provide fair access to the server. The faster port requests can be enqueued, the closer a true FIFO ordering of incoming requests is approached (this point is discussed further in §6.13). The entries on a port queue are quite small, so a long queue will not require excessive storage space. Data sent to the port is not buffered so clients writing on the port cannot do work in parallel with the port reader.

Implementation of Ports

```
-- Example of an input port implementation
-- We bind the operation Port_Op to the port;
-- this operation is invoked when data is read from the port.

var
   port: PATTERN;
   portq: Queue[...] of REQUESTER_SIGNATURE;

Initialization (Parent_MachineID : MACHINE_ID)
begin
      port := %...;
      ADVERTISE (port);
   end;
end;


Task()
var buffer: BUFFER;
begin
   loop
      if not IsEmpty(portq) then
         OPEN;  -- if closed before, must be room in portq now
         ACCEPT_PUT (DeQueue(portq), OK, buffer);
         Port_Op (buffer);
      fi;
   forever;
end;


Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
var I: integer;
begin
   case ENTRY of:
      OTHERWISE: begin
         EnQueue(portq, Asker);
         if IsFull(portq) then CLOSE; fi;   -- no room in portq
      end;
   esac;
   case COMPLETION of:
      -- handle completion of B_PUT here as described in
      -- §4.1.1
   esac;
end;
```

Priority queues are implemented by adding a priority field to each queue entry. The argument provided with the REQUEST is used as a priority. The polling loop in the task then selects the entry with the highest priority for completion. The queue entries may be stored in a sorted data structure, such as a heap, for efficient access to the highest priority entry.

4.2.2. Remote Procedure Call

Remote procedure call (RPC) is a very common style of communication in distributed systems. A complete discussion of the issues involved in RPC may be found in [55]. RPC is similar to a normal procedure call except that the caller and subroutine are on different machines. Should the machine executing the remote subroutine crash, the caller should be informed so that the call may be repeated using a different machine. We do not present details of how the exception handling mechanism should be designed for managing crashes.

We present here a possible implementation of RPC in SODA. The caller issues a PUT to pass parameters to the remote subroutine and then a blocking GET to retrieve results from the remote call. The server will invoke the remote subroutine when both PUT and GET have been received, ACCEPTING the PUT to obtain the *in* parameters. When the remote routine completes, the *out* parameters are returned by ACCEPTING the GET, which also unblocks the caller. The pattern used in the PUT and GET is bound to a particular subroutine in the server.

Implementation of RPC

```
-- Code for issuing the remote procedure call
--ignores type checking
const
   PROC_PATT = %0123;
   remote_procedure = <..., PROC_PATT>;
Task()
var out_params, in_params: BUFFER;
begin
   B_PUT(remote_procedure, OK, out_params);    -- invoke remote proc
   B_GET(remote_procedure, OK, in_params);     -- get results
end;


-- Code for the remote procedure.
-- Only shows how to proceed when server provides a single procedure "proc"
--for remote invocation.

const PROC_PATT = %0123;
var in_params, out_params: BUFFER;
   call_ready, got_inparams: Boolean;
   caller: REQUESTER_SIGNATURE;
Task()
begin
   ADVERTISE (PROC_PATT);
   got_inparams := FALSE;
   call_ready := FALSE;
   while (not call_ready) do idle(); end; --wait for PUT and GET to arrive
   proc (in_params, out_params);          --invoke procedure; out_params are
                                          --passed by reference
                                          --and filled with the return value
   ACCEPT_GET (caller, OK, out_params);
end;
```

```
Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer;
      Status: STATUS; Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
begin
   case ENTRY of
      PROC_PATT: begin
         if not got_inparams then
            ACCEPT_CURRENT_PUT(OK, in_params);
            got_inparams := TRUE;
         else
            caller := Asker;
            call_ready := TRUE;
         fi;
       end;
   esac;
end;
```

4.2.3. Remote Memory Reference

A Remote Memory Reference (RMR) is the access by one processor to the memory of another. At least two primitives are needed for RMR:

PEEK(dest, address, size, value)
 – *return contents of "size" words starting at location*
 – *"address" on processor "dest" and place in the reference*
 – *parameter "value"*
POKE(dest, address, size, value)
 – *install "value" which has "size" words in*
 – *location "address" on processor "dest"*

In addition, some synchronization mechanism (e.g. semaphores or test-and-set primitives) should be provided.

PEEK and POKE may be directly implemented in SODA. The server establishes a well-known RMR entry point. PEEK is implemented by a GET; POKE is implemented by a PUT. The SERVER SIGNATURE MID is the destination processor, the argument indicates the location in remote memory being referenced, and the buffer size shows the number of words being stored or fetched. Synchronization to protect critical sections is provided by OPEN and CLOSE or by scheduling ACCEPTS appropriately. We consider other RMR issues in §6.16.3.

4.2.4. Virtual Circuits

A *virtual circuit* is a logical communications channel between two processes. It is sometimes called a *link* when it is possible to change the binding of an end of the virtual circuit to a process dynamically (i.e., after the circuit is established).

A *link end* is represented by a table entry maintained by each client that contains a REQUESTER SIGNATURE indexed by a small integer *(link id)*. When a client wishes to issue a REQUEST, it uses a link id instead of a SERVER

SIGNATURE. SODAL replaces the link id with the SERVER SIGNATURE stored in the link table. Once a link is established, the SERVER SIGNATURE bound to the link id may change.

One possible link-based protocol is now described. To use links, various message passing primitives must be provided. These primitives will require specialized message formats: For example, some messages may contain enclosed links. When a child process is created it will possess a distinguished link to its parent by which the child may obtain other links. A process that possesses two links may INTRODUCE the two associated processes. As a result, the two processes have a link between themselves. A link end may be DESTROYED when the process owning it does not require the link any longer. A process attempting to send a message along a link that has a destroyed end will be informed of the destruction. Finally, a process may decide to MOVE an end of a link to another process. The MOVE should be transparent to the process at other end of the link. We present a SODAL algorithm for link moving here.

Implementation of Link Moving

```
– When two processes are INTRODUCED, one holds the MASTER
– end of the link and the other holds the SLAVE end.
– The SLAVE must become MASTER in order to move its end of a link.
– When a link is moving, REQUESTS issued over it are REJECTED and
– must be reissued when the link has completed its move.

type
  LinkEntry =
            record
            Machine: MACHINE_ID;
            Patt: PATTERN;
            State: (MASTER, SLAVE);
            Installed: (INSTALLED, BEING_INSTALLED);
          end;
          – illegal to issue REQUESTS on a link with
          – Installed <> INSTALLED
const
  LINK_SERVICE = %...;    – well-known entry for managing links
  LINKMAX = ...;          – maximum size of link table
var
  Link_Table: array [LINKMAX] of LinkEntry;
  curslot : integer;      – where a free link table entry is
  Moving: array [LINKMAX] of Boolean;
  RequestTable : array[MAXREQUESTS] of Transaction_Id;
  – holds uncompleted client requests (which use links).


Handler (Asker : REQUESTER_SIGNATURE;
     Arg : integer; Status: STATUS;
     Invoked_Pattern: PATTERN;
     PutSize, GetSize: integer);

var
  OtherEnd: MACHINE_ID;
  NewLink:
          record
            new_master: MACHINE_ID;
            new_pattern: PATTERN;
          end;
  patt: PATTERN;
  CurLink: integer;
```

```
begin
  case ENTRY of
    LINK_SERVICE: begin      – Install new MASTER link end of moving link
      patt := GETUNIQUEID(); – create new signature for link
      ADVERTISE (patt);
      linkid := curslot;
      curslot := NextSlot();  – find free slot in link table
      ACCEPT_CURRENT_EXCHANGE (OK, OtherEnd, patt);
      Link_Table[linkid] := {OtherEnd, patt, MASTER, BEING_INSTALLED};
      – ok to receive on this link but wait until previous
      – master tells me that the link is completely installed
      – before issuing requests on it
    end;
    Otherwise: begin – a regular client request has arrived
                     – search LinkTable for linkid
      CurLink := Find (Invoked_Pattern);
      if Moving[CurLink] then REJECT;
                     – the REQUEST must be reissued when the link
                     – end has completed the current move
      elsif Arg < 0 then  – special use of link
        case Arg of
          -1: begin      – Request to become MASTER of the link
            if not Moving[CurLink] then
              ACCEPT_CURRENT_GET (OK, SUCCESS);
              LinkTable[CurLink].State := SLAVE;
            else
              WantToMove[CurLink] := Asker;
              – we will ACCEPT after the move completes
              – and let the SLAVE ask again to be MASTER
            fi;
          end;
          -2: begin  – link has moved; update link table
            ACCEPT_CURRENT_PUT (OK, NewLink);
            LinkTable[CurLink].Machine := NewLink.new_master;
            LinkTable[CurLink].Patt := NewLink.new_pattern;
            FlushRejectedList(CurLink);
            – retry any rejected REQUESTS (which may fail again)
          end;
          -3: begin  – ok to use newly-moved link end
            ACCEPTCURRENT_SIGNAL (OK);
            LinkTable[CurLink].Installed := INSTALLED;
          end;
        esac;
      fi;
    esac;
  case COMPLETION of:
    – Store any REJECTED requests on RejectedList; these must be reissued
    – because they were issued to a link in transit
  esac;
end;
```

```
Initialization (Parent_MachineId: MACHINE_ID)
var i: integer;
begin
   for i := 1 to LINKMAX do Moving[i] := FALSE;
   curslot := 1;
   ADVERTISE (LINK_SERVICE);
end;

Task()
begin
   -- client code
end;


LINKMOVE (NewPartner: integer; CurrentPartner: integer);
-- move link bound to CurrentPartner to NewPartner
var newpattern, oldpattern: PATTERN;
   slave_end: MACHINE_ID;
begin
   Moving[CurrentPartner] := TRUE;
   BecomeMaster (CurrentPartner);    -- if MASTER, go ahead; if SLAVE,
                                     -- try to become MASTER
   oldpattern := LinkTable[CurrentPartner].Patt;
   oldmachine := LinkTable[CurrentPartner].Machine;
   newmachine := LinkTable[NewPartner].Machine;
   -- move link to new machine; get pattern bound to new link end
   B_EXCHANGE (<newmachine, LINK_SERVICE>, OK, oldmachine, newpattern);
   -- tell CurrentPartner how to change its tables and to
   -- reissue REJECTED requests
   B_PUT (<oldmachine, oldpattern>, -2, {newmachine, newpattern});
   -- tell new MASTER it's ok to REQUEST on the link
   -- (indicating that SLAVE changes are installed)
   -- we could save one message by instructing the SLAVE to do the
   -- move for us. Then the final synchronizing SIGNAL is not needed.
   B_SIGNAL (<newmachine, newpattern>, -3);
   Moving[CurrentPartner] := FALSE;
   if WantToMove[CurrentPartner] <> NIL then
      ACCEPT_GET (WantToMove[CurrentPartner], OK, FAILED);
      -- ok for CurrentPartner to try to become MASTER again
      WantToMove[CurrentPartner] := NIL;
   fi;
end;
```

```
BecomeMaster (linkid: integer)
var stat: (SUCCESS, FAILED);
begin
   while LinkTable[linkid].State = SLAVE do begin
      – ask to become MASTER
      B_GET (<LinkTable[linkid].Machine, LinkTable[linkid].Patt>, -1, stat)
      if (stat = SUCCESS) then
         LinkTable[linkid].State := MASTER;
      fi; – else, try again; MASTER end has moved
   end;
end;
```

4.2.5. Rendezvous Applications

Some distributed algorithms call for a *symmetric rendezvous* to take place between two processes. The desire is that if a process attempts to exchange messages with some set of processes, it will perform the exchange with *exactly one* other process from the set. Each process involved will know of the pairing. Once paired, the processes exchange information and then disengage from the rendezvous. Symmetric rendezvous is required in certain situations; for example, to include output guards in Communicating Sequential Processes [21,22].

Deadlock can occur if for example the rendezvous attempt is blocking and process P_1 attempts to rendezvous with process P_2 when P_2 is trying to rendezvous with P_1 at the same time. *Livelock* could result in the above scenario if both P_1 and P_2 discover that there is a possible deadlock and each aborts and restarts its rendezvous attempt. If the timing is synchronized, the same potential deadlock is detected and the backoff is repeated. A careful implementation must ensure that deadlock does not occur and allow one of the conflicting rendezvous requests to succeed.

Deadlock Danger in Symmetric Rendezvous

A requests rendezvous with B; B requests rendezvous with A
A can accept B's request; B can accept A's request

Acceptable Solutions:
1. A initiates and B accepts, or
2. B initiates and A accepts

Unacceptable Solutions:
1. A initiates and B initiates and no progress made, or
2. A initiates and B initiates, both accept

A less strict mechanism called *asymmetric* rendezvous splits processes into two groups: the *initiators* and the *acceptors* of a rendezvous. An initiator may attempt to rendezvous with only one other process. An acceptor however may rendezvous with any one of a number of initiators. This form of rendezvous is relatively easy to implement because the asymmetry obviates the unacceptable symmetric rendezvous situations.

CSP without output guards and ADA [25] are languages that require asymmetric rendezvous.

4.2.5.1. Communicating Sequential Processes (CSP)

We describe the asymmetric rendezvous mechanism available in CSP and discuss how it can be implemented in SODAL. Then we present Bernstein's algorithm [21] for adding output guards to CSP in SODAL. The following description of CSP is due to Bernstein [21].

> Central to the language are the input and output constructs and the use of Dijkstra's guarded commands [17]. The former are used by a process to control the flow of information from or to a device or another process.

> They have the form <process name> ? <target variable> and <process name> ! <expression>, respectively. The process name identifies the process with which the communication is to take place, and the value of the expression in the output [!] command must match the [type of the] target variable in the input [?] command in order for information to be transferred. Matching commands are executed simultaneously in the named processes and thus either process may be forced to wait for the other.
>
> A guarded command has the form G→C, where G is a guard and C a command list. A guard consists of a (possibly empty) list of Boolean expressions and declarations that may be followed by an input command. Output expressions may not appear in guards. The guard fails if any of the Boolean expressions have value "false" or if the process named in the input command has terminated. The command list is executed if all Boolean expressions are "true" and the input statement is executed. Guarded commands may be combined into an alternative command having the form:
>
> $$[\, G_1 \rightarrow C_1 \,[]\, G_2 \rightarrow C_2 \cdots []\, G_n \rightarrow C_n \,]$$
>
> which specifies the execution of exactly one of its constituent guarded commands. Consequently, if all guards fail, the alternative command fails. If more than one guard is executable, an arbitrary one is selected.

To implement CSP guarded commands in SODA, we require that all output commands cause a REQUEST to be issued. Each client maintains a list of arriving output command REQUESTS. When an input guard is evaluated, the list of available output commands is scanned for a matching entry. If an alternative command cannot proceed because it is awaiting a matching output command, the client enters a polling loop that terminates when the output command list is updated to include a matching output command.

The requirement that an input command fail when the named process terminates complicates matters slightly. We cannot have input commands cause REQUESTS (which automatically fail upon termination of the destination client) to be issued because there is no way to ensure that exactly one REQUEST will be ACCEPTED, even if all remaining REQUESTS are CANCELLED immediately when the first REQUEST completes. One way to detect terminated processes is to issue a special REQUEST for each input guard in an alternative command list. This

REQUEST is never ACCEPTED by the destination. However, SODA guarantees (via the probing mechanism) that a REQUEST will complete with failure should the destination terminate. When the evaluation of an alternative command completes, all of these special REQUESTS may be CANCELLED.

Bernstein presents an algorithm that allows output guards in CSP. We have already discussed the possibility for deadlock when symmetric rendezvous is implemented; Bernstein's algorithm avoids this danger. We detail below an implementation of Bernstein's algorithm in SODAL.

Bernstein's Algorithm

```
-- Bernstein's algorithm for CSP with output guards.

var
   State: (ACTIVE, QUERYING, WAITING);
   Matched: Boolean;
   QueryPending: Boolean;
   Delayed: Queue[...] of REQUESTER_SIGNATURE;

Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
begin
   case ENTRY of
      MY_NAME: begin  -- got a query
         if State = WAITING and <Arg indicates a valid type> then
            ACCEPT_CURRENT_PUT (OK, variable)
            Matched := TRUE;
            State := ACTIVE;
         else if ((State = QUERYING) and <Arg indicates a valid type> and
               QueryPending and (MY_MID > Asker.Mid)) then
            Enqueue(Delayed, Asker); -- we will delay the query
         else
            REJECT;     -- ACTIVE or no match or QUERYING and
                        -- MY_MID < Asker.Mid.
         -- If ACTIVE, we may eventually issue a REQUEST to the
         -- REJECTED client when we enter an alternative command
         -- and the REJECTED client is WAITING.
         fi;
      end;
   esac;
end;
```

```
Initialization (Parent_MachineId: MACHINE_ID)
begin
   Matched := FALSE;
   ADVERTISE (MY_NAME);
   QueryPending := FALSE;
end;

Task()
begin
   State := ACTIVE;
   loop
      if <Alternative Command> then
         EvalAltCmd();
      else
         <evaluate next statement>;
      fi;
      <get next statement>;
   until <no more statements>;
end;
```

```
EvalAltCmd()
var
   I: integer;
   result: STATUS;
begin
   State := QUERYING;
   <select guards in some order for evaluation. WLOG let this order
   be G1, G2, ..., Gn; set GuardList to contain the guards>;
   for I := 1 to n do begin
      CurrentGuard := GI;
      <evaluate all Boolean expressions in CurrentGuard>;
      if <not all Boolean expressions are TRUE> then
         <remove GI from GuardList>;
      else
         if <CurrentGuard has no input for output commands> then
            State := ACTIVE;
            <Execute command list I>;
            return;
         else
            P := <SERVER SIGNATURE bound to process named in guard>;
            QueryPending := TRUE;
            result := B_PUT (P, <type of variable>, <value of variable>);
            -- try to rendezvous; succeed if other guy is WAITING and
            -- can match the variable type. May be delayed here
            -- if other guy is also QUERYING.
            QueryPending := FALSE;
            case result of:
            REQUEST_FAILED: begin  -- P died
                  <remove GI from GuardList>;
               end;
            REJECTED: begin -- P didn't match or was not in WAIT state
                  if not IsEmpty(Delayed) then
                     ACCEPT_PUT (Dequeue(Delayed), <variable>);
                  fi;
                  -- else do nothing; P may match eventually or
                  -- may attempt to query us
               end;
            REQUEST_COMPLETED: begin  -- Success
                  State := ACTIVE;
                  <Execute command list I>;
                  return;
               end;
            esac;
         fi;
      fi;
   end; -- for loop
```

```
    State := WAITING;
    while not Matched do Idle(); end;
    Matched := FALSE;
    -- State set to ACTIVE (atomically) in handler
end;
```

-- Example: Suppose P_1 is querying P_2 who is querying P_3 who is querying P_1.
-- P_2 delays P_1 ; P_3 delays P_2 ; P_1 REJECTS P_3 unblocking P_3's query.
-- P_3 ACCEPTS P_2 unblocking P_2's query; P_2 ACCEPTS P_1 .

4.3. Scenarios for SODA Use

In this section we discuss how SODA can address two important issues in a distributed operating system: How processes locate each other; and how timeouts for IPC primitives can be provided.

4.3.1. Connection Methods

There is an obvious bootstrapping problem with entries: How does a client obtain an entry point for another? There are three basic ways. In the first method *(compile-time interconnection)* a pattern that is known by the client at compile time is used and the associated MID is discovered by issuing broadcast REQUESTS. In the second *(load-time interconnection)* a group of related processes is loaded by a *connector* process that modifies each process' core image to use specific REQUESTER SIGNATURES. Alternatively, the connector may provide specific REQUESTER SIGNATURES at client initialization time by sending REQUESTS containing signatures to the clients. In the third method *(run-time interconnection)* a *switchboard* process may be interrogated to obtain an entry point while a process is running.

A connector is used to load processes on different machines and establish communications paths between processes. The connector will boot the number of SODA machines necessary for the application. In so doing, it obtains the MIDS of each machine. Each client specifies which modules it will communicate with. For each connection, the connector creates a REQUESTER SIGNATURE by concatenating the result from a GETUNIQUEID call to the MID of the client's machine. The connector will then modify the client core image to use the new REQUESTER SIGNATURE.

```
– Connector has loaded client C1 on machine M1 and client C2
– on machine M2. In a specification file, the user states that
– C1 and C2 should have a communication path between them.
Initialization(...) – C1's initialization section
begin
  ADVERTISE(?p);
end;
Task() – C1's task
begin
  PUT (<?m,?p>, OK, buffer); – talk to C2
  ...
end;
Handler(...) – C2 handler
begin
  case ENTRY of
    ?p: ACCEPT (buffer, OK);
  esac;
end;
– 1. connector gets a new pattern P from GETUNIQUEID
– 2. connector obtains machines M1 and M2 for booting
– 3. connector changes "?p" in the core images of C1
– and C2 to be P and ?m in C1 to be M2
– 4. connector loads C1 on M1 and C2 on M2
```

A connector can be thought of as a linkage editor which, instead of tightly linking separate modules together, links them loosely together by establishing entry points used for intermodule communication. A connector establishes communication paths in a manner quite similar to the way the MULTICS [56] linkage editor makes segments "known". A connector in a distributed computing environment is used in the Charlotte [38] and Arachne [35] operating systems.

Connectors may be used to establish policy on node allocation. If a client wishes to boot a set of machines, it may be required to use a connector/manager process to obtain the required free nodes. Kernel pattern screening prevents clients from outside the set from inadvertently communicating with members within the set.

4.3.2. Timeouts

SODA does not provide timeouts with any of its communications primitives. Therefore, it must be possible to abort a communication attempt in order to allow an impatient client to continue (or even break a potential deadlock) after trying to contact another client.

One way to implement timeouts is to register a wakeup REQUEST with a *timeserver utility* (which possesses a hardware clock) prior to initiating a REQUEST to a potentially slow server. Such a REQUEST could be a simple (non-blocking) SIGNAL with the delay specified in the REQUEST argument. The timeserver notes the delay and REQUESTER SIGNATURE. When the delay has expired, the REQUEST is ACCEPTED, thus notifying the requester that the alarm has expired. The requester may then CANCEL outstanding requests to other clients and attempt alternative action.

4.4. Programmed Examples

We now present five detailed examples using SODAL. The examples were chosen primarily to illustrate the completeness of the SODA primitives. The examples we present are:

1) two-way bounded buffer
2) four-way bounded buffer
3) readers-writers
4) dining philosophers
5) file server

4.4.1. Two-Way Bounded Buffer

In this problem, producer processes (such as teletype drivers) are producing data and delivering it to a consumer process (such as a file server) that performs buffering to better match speeds with the producer. In the event the producer is

too fast, however, the consumer must provide backpressure in order that its buffers are not overrun.

In our solution, a producer uses a double-buffering scheme in order that it may work while the consumer is processing the producer's last request. The consumer performs buffering on two resources. First, the consumer saves REQUESTER SIGNATURES in a queue when it cannot immediately buffer a request, which helps ensure that incoming requests will be be processed in the order of arrival. Because the SODA kernel does no request queueing of its own, fair access to the handler is achieved by making the handler code as short as possible. Second, the consumer buffers data from producers so that producers may work in parallel with the consumer. Flow control on REQUESTER SIGNATURES is achieved by CLOSING the handler when the request signature queue fills. Flow control on data is managed because a producer will not issue a new request until its most recent request has been ACCEPTED.

Two-Way Bounded Buffer

```
-- Producer Process
const
   CONSUMER_PATTERN = %...;
var
   ready : Boolean;
   Item1, Item2: BUFFER;
   current : BUFFER;
   consumer : SERVER_SIGNATURE;

Initialization (Parent_MachineId: MACHINE_ID)
begin
   Item1.Size := ITEMSIZE; Item2.Size := ITEMSIZE;
   Item1.Address := getspace(ITEMSIZE); Item2.Address := getspace(ITEMSIZE);
   current := Item1;
   consumer := DISCOVER (CONSUMER_PATTERN); -- locate consumer
   ready := TRUE;
end;

Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer)
begin
   if Status = REQUEST_COMPLETION then ready := TRUE; fi;
end;

Task()
var tid: TRANSACTION_ID;
begin
loop
   current -> Size := ...;
   current -> Address := ...;          -- produce some data
   while not ready do idle(); end;     -- wait for last request to complete
   ready := FALSE;
   PUT (consumer, OK, current);        -- send data to consumer
   if (current = Item1) then           -- swap buffers so we can produce
                                       -- as other buffer is being consumed
      current := Item2;
   else
      current := Item1;
   fi;
forever;
end;
```

```
-- Consumer process
-- Can consume data from a set of producer processes

const
   MAXQSIZE = ...;
   MAXPORTSIZE = ...;
   CONSUMER = %...;
var
   Produced, FreePool : Queue[MAXQSIZE] of BUFFER;
   Pending : Queue[MAXPORTSIZE] of REQUESTER_SIGNATURE;
   work, tmp : BUFFER;


Task()
var
   work, tmp: BUFFER;
begin
loop
   CLOSE();
   if not Queue_Empty(Produced) then
      work := DeQueue(Produced);                    -- obtain data to consume
   else
      work := NIL;
   fi;
   if not Queue_Empty(Pending) then                 -- Produced no longer full
      tmp := DeQueue(FreePool);                     -- get free buffer
      ACCEPT_PUT (DeQueue(Pending), OK, tmp);       -- obtain data
      EnQueue(Produced, tmp);    -- Buffer data for eventual consumption
                                 -- Pending no longer full
   fi;
   OPEN();
   if not (work = NIL) then
      process_data (work);       -- consume data
      EnQueue (FreePool, work);  -- return buffer to free pool
   fi;
forever;
end;
```

```
Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
var tmp: BUFFER;
begin
   if Status = REQUEST_ARRIVAL then
      if Queue_Full(Produced) then                    -- no more buffers
         EnQueue (Pending, Asker);                    -- save this request
         if Queue_Full(Pending) then CLOSE(); fi;     -- no room for new requests
      else
         tmp := DeQueue(FreePool);                    -- get a free buffer
         ACCEPT_CURRENT_PUT (OK, tmp);                -- fill it with data from producer
         EnQueue(Produced, tmp);                      -- queue it for eventual consumption
      fi;
   fi;
end;

Initialization (Parent_MachineId: MACHINE_ID)
begin
   Pending := NIL;
   <Initialize FreePool by enqueueing MAXQSIZE empty buffers>;
   ADVERTISE (CONSUMER);
end;
```

4.4.2. Four-Way Bounded Buffer

This example illustrates how SODA can be used to manage a complex scheduling situation. Two clients are each attached to similar devices. The devices maintain internal buffers and follow a CNTRL-S/CNTRL-Q flow-control protocol to start and stop the device. Each client reads from its device and sends the data to the other client. Further, incoming data from each client is buffered. Thus, each client is both a producer and a consumer.

The task in each client polls the device input-ready and output-ready registers. When the device has produced some data, it is shipped off to the remote client which then buffers this data in a FIFO queue. When the device is ready to accept some data, the client takes an item off of its queue and writes the device.

An interesting use of EXCHANGE appears in this algorithm. When the remote client buffer is written, a status is returned. The status indicates whether or not the remote buffer is full. Because the EXCHANGE used is blocking, the producing client is informed immediately of the remote buffer situation and its device is stopped.

Four-Way Bounded Buffer

```
-- This client is attached to a device and communicates with
-- another client attached to a similar device.
-- The device produces data (when input_ready) and accepts
-- data (when output_ready). The device will stop producing upon hearing
-- CNTRL-S and start again hearing CNTRL-Q.
-- Each client tries to read data from its device and
-- send the data to the remote device. Incoming data is buffered
-- by the client and must be flow-controlled.

const
   OTHER = ...;              -- machine id of other client
   RESTART = %...;           -- pattern used for restarting remote client
   BUFFER_DATA = %...;       -- pattern used for remote buffering entry
   RemoteRestart := <OTHER, RESTART>;
   RemoteBuffer := <OTHER, BUFFER_DATA>;
type STATE = (CONTINUE, FULL);
var
   DevBufFull, PartnerBufFull, PartnerBufEmpty,
   RemoteClientStopped: Boolean;
   status : STATE;
   data: BUFFER;
   Q : Queue [...] of BUFFER;

Initialization (Parent_MachineId: MACHINE_ID)
begin
   DevBufFull := FALSE;
   PartnerBufFull := FALSE;
   PartnerBufEmpty := FALSE;
   RemoteClientStopped := FALSE;
end;
```

```
Task()
begin
loop
   -- read data from device (READ loop)
   if not PartnerBufFull and (DEV_IN_STATUS = DATA_AVAIL) then
   -- device has produced some data
      data := DEV_IN_BUF; -- reading DEV_IN_BUF resets DEV_IN_STATUS
      case data of
         'CNTRL-S': DevBufFull := TRUE;    -- device's buffer is full
         'CNTRL-Q': DevBufFull := FALSE;   -- device's buffer is empty
         OTHERWISE: begin
            -- send data to other client
            B_EXCHANGE (RemoteBuffer, OK, data, status);
            if status = FULL then          -- other guy is now full
               PartnerBufFull := TRUE;     -- shuts down READ loop until
                                           -- this can take effect in
                                           -- WRITE loop

            fi;
         end;
      esac;
   fi;

   -- store data in device (WRITE loop)
   if not DevBufFull and (DEV_OUT_STATUS = READY) then
   -- device ready to accept data
      if PartnerBufFull then               -- other client's buffer is full
         PartnerBufFull := FALSE;
         DEV_OUT_BUF := 'CNTRL-S'; -- Stop device from producing more
                                   -- shuts down READ loop
                                   -- (DEV_OUT_STATUS
                                   -- will not be DATA_AVAIL)
      elsif PartnerBufEmpty then           -- other client's buffer is empty
         PartnerBufEmpty := FALSE;
         DEV_OUT_BUF := 'CNTRL-Q'; -- Restart device
      elsif not QueueEmpty(Q) then
         data := DeQueue(Q);
         DEV_OUT_BUF := data;              -- store data in device
         if IsEmpty(Q) and RemoteClientStopped then
            RemoteClientStopped := FALSE;
            B_SIGNAL (RemoteRestart, OK);  -- restart other client
         fi;
      fi;
   fi;
forever;
end;
```

```
Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer)
const SIZE = ...;
var
   free: BUFFER;
   ReturnStatus: STATE;
begin
   case ENTRY of
      BUFFER_DATA: begin  – Buffer data from other client
         free.Address := getspace (SIZE);
         free.Size := SIZE;
         ReturnStatus := CONTINUE;
         if AlmostFull(Q) then
            RemoteClientStopped := TRUE;    – stop remote producer now
            ReturnStatus := FULL;
         fi;
         ACCEPT_CURRENT_EXCHANGE (OK, free, ReturnStatus);
         EnQueue(Q, free);
      end;
      RESTART: begin – ok to produce again and restart device
         ACCEPT_CURRENT_SIGNAL (OK);
         PartnerBufEmpty := TRUE;
      end;
   esac;
end;
```

4.4.3. Dining Philosophers

This famous problem was first proposed by Dijkstra [57]. We assume that the reader is familiar with this problem. We propose a somewhat different solution than the one given by Dijkstra. There are five philosopher processes, each of which owns one fork. Additionally, there is a deadlock-detector process and a timeserver process. The philosophers think, try to grab two forks, eat, put back the forks and iterate. The deadlock detector process is periodically awakened by the timeserver process. Once awakened, the deadlock detector attempts to detect a deadlock by asking the philosophers what their state is. If a deadlock is found, one philosopher is asked to give up his fork so that the other philosophers can eat. To ensure fairness, a philosopher will not be asked to return his fork twice before all other philosophers have returned their fork once. To implement this fairness policy, a list (LIST_OF_NICE_PHILOS) of philosophers who have been asked to return a fork is maintained. In addition, we have taken care (see program, below) that once a philosopher releases its fork, it will be assured of obtaining that fork before its successor uses it twice.

Call a philosopher *needful* if he has obtained one fork and wants the other. Philosophers first obtain their left fork and then their right fork before eating. A needful philosopher's *successor* is the neighbor with whom he shares the allocated fork. The deadlock detector algorithm:

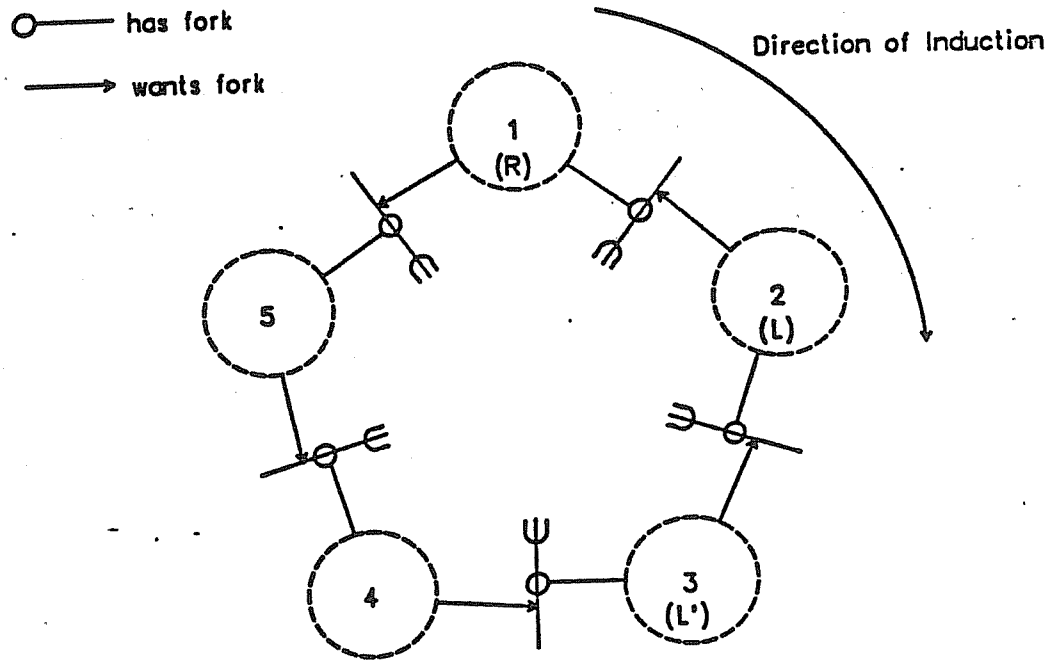
(1) Initially, set LIST_OF_NICE_PHILOS to contain all philosophers.
(2) On being awakened, select a random philosopher R from LIST_OF_NICE_PHILOS. Ask R if it is needful. If not, the deadlock detector goes back to sleep. If so, the philosopher reports the TID of its REQUEST for the first (left-hand) fork.
(3) If R has responded that it is NEEDFUL, the detector asks R's successor if it is needful. If not, the detector goes back to sleep. If so, it asks the successor of that philosopher and so on. Only the first philosopher R is required to report the TID of its first fork REQUEST; the others may simply report whether or not they are needful.

(4) If the detector gets back to the original philosopher R, it asks again if it is needful and if so, what the TID of its outstanding REQUEST is. If R is needful and the TID is the same as the one it reported the first time it was asked, R must not have changed state between probes by the deadlock detector and deadlock is announced.

(5) Remove R from LIST_OF_NICE_PHILOS. If this list is now empty, reinitialize it to contain all philosophers. R is now told to release the fork it owns to break the deadlock.

We show that a philosopher is only told to release a fork if deadlock exists by induction (see diagram, below). At the moment we ask R the second time and R is in the same state, we know that R's left-hand neighbor L must want the fork R has because we know that L was NEEDFUL. Now consider the left-hand neighbor of L, L'. We asked L' if it was NEEDFUL *after* we asked L. The NEEDFUL state of L is unchanged since we first asked it so L' must be in its same state as well. By continuing with around the ring in this fashion, we see that all philosophers must be in the same NEEDFUL state since the deadlock detector first inquired so deadlock must exist. Q.E.D.

Dining Philosophers Deadlock Detection



Dining Philosophers

```
-- Timeserver Process
const ALARM_CLOCK = %...;
var
   expire: integer;
   detector: REQUESTER_SIGNATURE;
Task()
begin
   loop
      <wait for clock tick on hardware clock>;
      case expire of:
         -1: skip;
         0:  ACCEPT(detector, OK);
         OTHERWISE: expire := expire - 1;
      esac;
   forever;
end;

Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
const INTERVAL = ...;
begin
   case ENTRY of
      ALARM_CLOCK: begin
         expire := INTERVAL;
         detector := Asker;
      end;
   esac;
end;

Initialization (Parent_MachineId: MACHINE_ID)
begin
   expire := -1;
   ADVERTISE (ALARM_CLOCK);
end;
```

```
-- Philosopher Process
const  -- well-known patterns
   GETFORK = %...;
   PUTFORK = %...;
   RETURN_FORK = %...;
   CHECK = %...;
   GIVE_BACK = %...;
   LEFTFORK = %...;
   HE_OWNS = 0;
   I_OWN = 1;
var
   forkstate : array[HE_OWNS..I_OWN] of
      (MINE, HIS, IDLE);
   -- I_OWN my "right" fork
   -- HE_OWN's (my left-hand neighbor) the left fork
   myrequest : TRANSACTION_ID;  -- TID of my request for LEFTFORK
   hisrequest : TRANSACTION_ID;  -- TID of his request for fork I control

Initialization()
begin
   forkstate[HE_OWNS] := IDLE;
   forkstate[I_OWN] := IDLE;
   myrequest := hisrequest := NIL;
end;


grab_my_fork() : Boolean;
-- attempt to obtain my right fork which I control
var result : Boolean;
begin
   CLOSE(); -- critical section since forkstate changed by handler
   if forkstate[I_OWN] = HIS
      then result := FALSE
   else
      result := TRUE;
      forkstate[I_OWN] = MINE;
   fi;
   OPEN();
   return (result);
end;
```

```
Task()
begin
loop
 think();
 myrequest :=
  SIGNAL (<LEFTFORK, GETFORK>, OK);
 while forkstate[HE_OWNS] <> MINE do idle();
 while (not grab_my_fork() or
  forkstate[HE_OWNS] <> MINE) do idle();
 -- need to retest forkstate in case we had to give the fork back
 eat();
 B_SIGNAL (<LEFTFORK, PUTFORK>, OK);
 forkstate[I_OWN] := IDLE;
 forkstate[HE_OWNS] := IDLE;
 if hisrequest <> NIL then
   forkstate[I_OWN] := HIS;
   ACCEPT_SIGNAL(hisrequest, OK)
   hisrequest := NIL
 fi;
forever;
end;

Handler(Asker : REQUESTER_SIGNATURE;
      Arg : integer;
      Which : PATTERN;
      PutSize, Getsize : integer);
begin
   case COMPLETION of
     myrequest: begin
        myrequest := NIL;
        forkstate[HE_OWNS] := MINE;
      end;
   esac;
```

```
case ENTRY of
   PUTFORK: begin
     ACCEPT_CURRENT_SIGNAL (OK);
     forkstate[I_OWN] := IDLE;
    end;
   GETFORK:
    if forkstate[I_OWN] = MINE then
     hisrequest := Asker;
    else
     forkstate[I_OWN] = HIS;
     ACCEPT_CURRENT_SIGNAL (OK);
    fi;
   CHECK:
    if forkstate[HE_OWNS] = MINE
    and forkstate[I_OWN] = HIS then
     ACCEPT_CURRENT_GET (OK, myrequest)
    else
     REJECT;
    fi;
    GIVE_BACK: begin
     myrequest :=
       SIGNAL (<LEFTFORK, RETURN_FORK>, OK);
       forkstate[HE_OWNS] := HIS;
    end;
    RETURN_FORK: begin
       forkstate[I_OWN] := MINE;
       hisrequest := Asker;
      end;
  esac;
end;
```

```
-- Deadlock Detector process

const
   CHECK = %...;
   GIVE_BACK = %...;
   ALARM_CLOCK = %...;

var
   AlarmServer : SERVER_SIGNATURE;
   AlarmTID : TRANSACTION_ID;
   Phil: array [1..5] of MACHINE_ID;
   PossibleVictims: set of 1..5;
   TimesUp : Boolean;
   NextVictim: integer;

Initialization()
begin
   <initialize Phil>;
   PossibleVictims := [1,2,3,4,5];
   NextVictim := Random (PossibleVictims);
   PossibleVictims :=
      PossibleVictims - [NextVictim];
   TimesUp := FALSE;
   AlarmServer := DISCOVER(ALARM_CLOCK);
   AlarmTID := SIGNAL(AlarmServer, OK);
end;
```

```
Task()
var
   Current : integer;
   CheckTID, FirstTID: TRANSACTION_ID;
begin
loop mainloop:
 if TimesUp then
   TimesUp := FALSE;
   if B_GET (<Phil[NextVictim], CHECK>,OK, FirstTID)
      = REJECTED then continue mainloop;
   fi;
   Current := NextVictim;
   repeat
     Current := Current mod 5 + 1;
     if B_GET (<Phil[Current], CHECK>, OK, CheckTID)
        = REJECTED then continue mainloop; fi;
   until Current = NextVictim;
   if CheckTID <> FirstTID then
      continue mainloop
   fi;
   B_SIGNAL (<Phil[NextVictim], GIVE_BACK>, OK);
   if PossibleVictims = [] then
     PossibleVictims := [1,2,3,4,5];
   fi;
   NextVictim := Random (PossibleVictims);
   PossibleVictims :=
      PossibleVictims - [NextVictim];
 fi;
forever;
end;

Handler(Asker : REQUESTER_SIGNATURE;
      Arg : integer;
      Which : PATTERN;
      PutSize, Getsize : integer);
begin
   case COMPLETION of
      AlarmTID: begin
        TimesUp := TRUE;
        AlarmTID := SIGNAL(AlarmServer, OK);
       end;
   esac;
end;
```

4.4.4. Concurrent Readers and Writers

This problem was initially posed by Courtois [58]. We wish to provide a concurrency control service for a database. This service is invoked by requesting permission to read or write (via START_READ or START_WRITE), performing the read or write, and releasing the request (via END_READ or END_WRITE). We wish to exclude readers from access to the data while a write is in progress and writers from access to the data when reads or writes are in progress. This exclusion must be fair in the sense that when a write request is pending, no new read requests will be honored and when a write is in progress, those read requests that have accumulated during the write are honored before any new writes may begin.

We will use a separate client process (distinct from the database itself) called the *moderator* to process the START_READ, START_WRITE, END_READ, END_WRITE requests. We assume that the clients accessing the database are correct in the sense that all reads are preceded by START_READ and followed by END_READ; and all writes are preceded by START_WRITE and followed by END_WRITE. This protocol can be enforced at compile-time given suitable programming language features (such as *MOD regions) [26].

Readers and Writers

```
-- Moderator Client
var
   ReadQueue, WriteQueue: Queue[...] of REQUESTER_SIGNATURE;
   readcount: integer;    -- number of active readers
   writecount: integer;   -- number of active writers (0 or 1)

Task()
begin
   loop
      Idle();
   forever;
end;

Initialization (Parent_MachineId: MACHINE_ID)
begin
   readcount := 0;
   writecount := 0;
end;

Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer; Status: STATUS;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
begin
   case ENTRY of
      START_READ: begin
         if (IsEmpty (WriteQueue) and (writecount = 0)) then
            ACCEPT_CURRENT_SIGNAL (OK);
            readcount := readcount + 1;
         else
            EnQueue (ReadQueue, Asker);
         fi;
      end;
      STARTWRITE: begin
         if ((readcount = 0) and (writecount = 0)) then
            ACCEPT_CURRENT_SIGNAL (OK);
            writecount := writecount + 1;
         else
            EnQueue (WriteQueue, Asker);
         fi;
      end;
```

```
    END_READ: begin
       ACCEPT_CURRENT_SIGNAL (OK);
       readcount := readcount - 1;
       if ((readcount = 0) and not IsEmpty(WriteQueue)) then
          ACCEPT_SIGNAL (DeQueue(WriteQueue), OK);
          writecount := writecount + 1;
       fi;
    end;
    END_WRITE: begin
       ACCEPT_CURRENT_SIGNAL (OK);
       writecount := writecount - 1;
       if not IsEmpty (ReadQueue) then
          readcount := 0;
          while not IsEmpty (ReadQueue) do -- handle waiting readers
             ACCEPT_SIGNAL (DeQueue(ReadQueue), OK);
             readcount := readcount + 1;
          end;
       elsif not IsEmpty (WriteQueue) then
          ACCEPT_SIGNAL (DeQueue(WriteQueue), OK);
          writecount := writecount + 1;
       fi;
    end;
  esac;
end;
```

4.4.5. File Service

We present an outline of how a file service might be constructed. A client wishing to open a file begins by locating the appropriate file server via DISCOVER. Then, the requester client issues a REQUEST using the well-known pattern OPEN. The file server creates a pattern via GETUNIQUEID, binds it to the file, and returns this pattern to the requester. This pattern is then used by the requester for future transactions concerning the file.

File Server

```
– client protocol for using file server

1. – locate file server
fs := DISCOVER (FILESERVER);
– Locate the MID of the appropriate file server.
– FILESERVER is a well-known name specific enough to
– locate the correct machine.
– For example, the pattern can indicate a subdirectory containing
– a particular file.

2. – open file
var fd: PATTERN;
B_EXCHANGE (<fs.Mid, OPEN>, OK, "foo", fd);
– Get file descriptor pattern "fd" to be used for all transactions
– concerning file "foo"
– File opening errors are detected upon the first use of the file.

3. – use file (kind = seek, read, etc.)
B_EXCHANGE(<fs.Mid, fd>, kind, wrbuf, rdbuf);
– ask for something to be done with file "foo"
– kind is CLOSE, SEEK, READ, or WRITE
```

```
const
  FILESERVER = %...;
  OPEN = %...;
type FILE_OPERATION =
  record
    client: REQUESTER_SIGNATURE;
    operation: integer;
    filedesc: integer;
  end;
var
  OpQueue: Queue[...] of FILE_OPERATION;
  action: FILE_OPERATION;
  FileTable: array[...] of PATTERN;

Initialization (Parent_MachineId: MACHINE_ID)
begin
  ADVERTISE (FILESERVER);
  ADVERTISE (OPEN);
end;

Task()
begin
  loop
    while IsEmpty (OpQueue) do Idle();
    action := DeQueue(OpQueue);
    perform (action);
    -- perform will ACCEPT action.client
    -- and execute requested file operation
  forever;
end;


Handler (Asker : REQUESTER_SIGNATURE;
      Arg : integer;
      Invoked_Pattern: PATTERN;
      PutSize, GetSize: integer);
var fname: string;
begin
  case ENTRY of
    OPEN: begin
      fd := GETUNIQUEID();
      ADVERTISE (fd);
      ACCEPT_CURRENT_EXCHANGE (OK, fname, fd);
      I := open (fname);
      FileTable[I] := fd;
    end;
    OTHERWISE: begin
      I := Find (Invoked_Pattern);
      Enqueue (OpQueue, {Asker, Arg, I});
    end;
  esac;
end;
```

4.5. Conclusions

A rudimentary language (SODAL) was outlined. This language can be used either to implement application programs directly or as an intermediate form for higher-level language primitives. We demonstrated that SODA is convenient for providing both higher-level protocols (such as link management and file service) and high-level language constructs (such as remote procedure call and CSP-like rendezvous) in a distributed environment.

SODA does not provide all possible primitives. For example, it lacks reliable broadcast and highly-efficient remote memory reference primitives. However SODA appears to be adequate for typical IPC needs. A SODA machine may join an existing network and communicate with and boot other nodes without recourse to manager processes. A more disciplined approach (such as a centralized connector service that manages node allocation as well), can also be accommodated.

It is difficult to judge the power of a system from a small number of examples. Nevertheless, the examples presented here cover a wide variety of typical distributed applications. The SODA primitives have proved thus far to be powerful and expressive.

5. IMPLEMENTATION

In this chapter we discuss our implementation of SODA on a network of PDP-11/23's connected by a 1 megabit broadcast bus: Computrol's Megalink [59]. Included in our discussion will be details of the kernel-client interface. We also present performance results. Our goal in this discussion is to show that SODA is implementable in an efficient and compact manner.

5.1. Development System

Our network configuration consists of eight "bare" PDP-11/23's and a VAX-11/780 running UNIX [60] which all have access to the Megalink. All software was developed on the VAX and then downloaded to the PDP-11's using the Megalink.

5.2. Kernel Simulation

We simulated a SODA processor by implementing it in software. The SODA kernel is a collection of routines running on a PDP-11/23. The implementation must multiplex a single processor to perform the tasks of both client and kernel. This additional overhead would not be present on an ideal two-processor SODA node.

The kernel accesses physical addresses directly, whereas the client memory is mapped so that it forms a logical address space extending from zero up to 65535, with some exceptions discussed below. Normally, the client process is the only activity engaging the CPU. The kernel is only active when the client executes a SODA primitive, an alarm expires, or a network interrupt arrives.

The Megalink always operates out of two preallocated kernel buffers (the input buffer and the output buffer). Messages are copied out of client space to the output buffer for transmission and are copied from the input buffer into client space upon receipt.

5.2.1. SODA-Client Interface

SODA and the client communicate by shared memory. A portion of client memory is mapped to coincide with kernel memory (the *interface segment*). Each of the ten SODA primitives is invoked via a TRAP. All arguments are obtained from the interface segment. If TRAPS are replaced with control line access, this setup corresponds well with the architecture illustrated in §3.2.

To invoke a SODA primitive which requires arguments (such as REQUEST or ACCEPT), the client first prepares a descriptor that contains the arguments and places the address of this descriptor in the interface segment. Then the client invokes the appropriate TRAP. The client should not alter the contents of the descriptor until the descriptor is no longer needed. Descriptors used for REQUESTS may safely be reused after the associated REQUEST completes. Other descriptors may be reused as soon as the primitive completes execution because all other SODA primitives block until completed. There is a problem of concurrency involved however. Descriptors are allocated (by the client) from a pool of free descriptors. The client, when executing within the task, must be careful to avoid inconsistent states of the free descriptor pool by using CLOSE and OPEN.

The client does not receive hardware interrupts or TRAPS but has direct access to the I/O registers which enables the client to perform busy-wait I/O. SODA itself only requires access to the clock and the network device interrupts. SODA will generate a software interrupt to invoke the client handler.

When the client is waiting for a handler interrupt and has nothing else to do, it may execute an IDLE instruction (analogous to the WAIT instruction on a PDP-11) if one is available. This instruction will prevent the client from accessing memory it shares with the SODA kernel until an interrupt occurs, thus avoiding unnecessary contention.

5.2.2. Communications Protocol

ACCEPT can never be prevented from executing by a requester client. This ensures that there is no deadlock when two clients issue ACCEPTS to each other and also ensures that ACCEPT will complete within a bounded amount of time. When a requester kernel is informed of an incoming ACCEPT, the buffers for data transfer are filled by the kernel and the completion interrupt is queued if the requester handler is either CLOSED or BUSY. When the client executes ENDHANDLER, SODA checks to see if any REQUEST completion indications are pending. If so, control passes immediately back to the handler. Similarly, if the client has CLOSED the handler, and completion indications have accumulated since the CLOSE, OPEN will cause immediate handler invocation.

If a REQUEST encounters a BUSY or CLOSED handler, the REQUEST will be periodically repeated until it succeeds or fails. Different REQUESTS to the same server will always be delivered in the order issued. However, REQUESTS to different servers will be issued even if some pending REQUESTS encounter BUSY or CLOSED handlers. The rate of REQUEST retransmission decreases with the number of retransmission attempts to avoid flooding the bus needlessly.

The Megalink computes a cyclic redundancy check (CRC) checksum to detect transmission errors. A message with an incorrect CRC is simply discarded. An alternating-bit stop-and-wait protocol as described in [61] is used by the ker-

nel to achieve reliability in the presence of lost or mutilated packets. If a packet is not acknowledged, SODA waits a random amount of time and tries again. SODA assumes that the destination client has crashed if no response is heard after retransmitting a packet to a destination a given number of times.

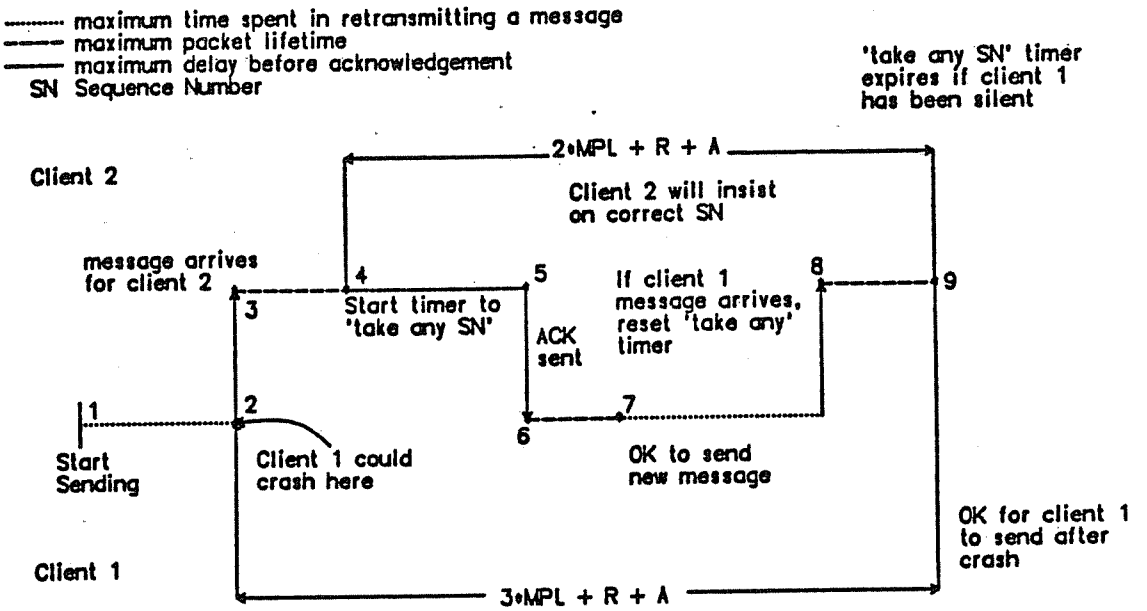
The retransmission rate to obtain an acknowledgement is not the same as the retransmission rate to find the server handler OPEN and IDLE. In our implementation, the former rate is faster. Also, the number of retransmissions used in an attempt to get an acknowledgement is bounded; the number of retransmissions to find the server handler OPEN and IDLE is potentially unbounded. One "retransmission" to find the server handler OPEN and IDLE may require several actual retransmissions to reliably deliver a REQUEST.

It is crucial to have a mechanism to establish initial sequence numbers for transactions between processors. One technique is the *three-way handshake* [62], in which a connection is explicitly established by exchanging packets. A simpler protocol is the Delta-t protocol [63,64], which requires no explicit connection establishment. Instead, Delta-t uses the fact that in a local area network, the total time a message is retransmitted (R), the time to convey a message from sender to receiver (maximum packet lifetime (MPL)), and the maximum delay in acknowledging a packet (A) can be kept within tight bounds. We outline the protocol here; more details are discussed in [63].

We define the interval Δt = MPL + R + A. If a processor P_1 hears from a remote processor P_2 and no connection record exists, *any* sequence number used by P_2 is accepted and used henceforth and a connection record is created. If P_1 hears nothing from P_2 in time MPL + Δt, P_1 destroys the connection record it has with P_2 and will again accept any sequence number from P_2. The interval MPL + Δt is sufficient for P_2 to complete all retransmissions (R), P_1 to receive the message

and acknowledge it (MPL + A) and P_2 to receive the acknowledgement (MPL). It ensures that P_2 has either crashed or has received an acknowledgement from P_1 for P_2's last retransmission.

Typical Delta-t Situations



When a message has arrived at a server handler but has not yet been accepted, the requester kernel will issue periodic probing messages to the server kernel. If no response is heard after retransmissions in the interval MPL + Δt, the requester kernel will report to its client that the server is dead. Similarly, when a REQUEST or an ACCEPT is issued and no response is heard after retransmitting during the interval MPL + Δt, the destination is reported dead.

If P_2 has indeed crashed, P_2 will wait 2*MPL + Δt before allowing any new communications to begin after recovering. This delay ensures that all message traffic and acknowledgements between P_1 and P_2 have died out and that the connection may safely be reestablished without introducing the possibility of duplicate packets.

If N is the number of nodes on the network, the number of connection records a node must allow space for is N - 1. This is due to the alternating bit stop-and-wait protocol which requires a packet to be acknowledged before a new one may be sent. Thus, at most one open connection for each node on the network may exist at one time. If N is quite large, it may be possible to tighten this bound further. If a node may only send and receive k messages within an interval of 2*MPL + Δt (due to limited processor speed or network bandwidth) then only space for k connection records need be maintained.

5.2.3. Acknowledgements

Acknowledgements (ACKS) are used to indicate successful receipt of a message. Negative acknowledgements (NACKS) are used for efficiency reasons and error conditions. All messages and acknowledgements carry the state of the sender's end of the connection which enables the receiver to discard duplicate packets. SODA requires different NACKS to indicate BUSY handlers (receipt of

such a NACK will cause adjustment of the retransmission interval) and for various errors (such as addressing a non-advertised PATTERN). All messages carry a bit that indicates whether the current Delta-t connection is open, which prevents a message from appearing to contain a piggybacked ACK when a connection is not currently active.

A REQUEST may only be CANCELLED if the state of the server is known. That is, a REQUEST must be acknowledged (but not necessarily ACCEPTED) before it is eligible for cancellation.

Our SODA implementation expends considerable effort to piggyback information on outgoing messages. As an example, when a REQUEST arrives at a server handler, the acknowledgement is delayed momentarily to give the server a chance to ACCEPT quickly. If the ACCEPT is issued soon after handler entry, the acknowledgement will be piggybacked on the ACCEPT.

In the best case, a PUT requires two messages to be sent: REQUEST+DATA by the requester and ACCEPT+ACK by the server. Because the ACK could be lost and because the ACK must carry information with it (the argument and the sizes of the buffers specified in the ACCEPT), the server kernel must maintain the ACCEPT arguments as part of the connection state. Then, if the ACK is lost, retransmissions by the requester will be acked with the appropriate information. For GETS and EXCHANGES, three messages are required: REQUEST+DATA by the requester; ACCEPT+DATA by the server; and ACK by the requester. A REQUEST is only sent with data (on a PUT or EXCHANGE) one time. Should the message get lost or arrive at a busy server handler, subsequent retransmissions do not include the data. This reduces the amount of network traffic while allowing for efficient communication between synchronized clients.

An EXCHANGE sent to a busy handler will require six (or more if the handler stays busy for a long time) messages:

1. REQUEST+DATA (by requester)
2. BUSY (by server) -- handler not available, try again
3. REQUEST (by requester)
4. ACCEPT+DATA (by server) -- made it to handler; accept is immediate
5. DATA+ACK (by requester)
6. ACK (by server)

If the ACCEPT was not issued immediately when the REQUEST arrived at the server handler, at least seven messages are required:

1. REQUEST+DATA (by requester)
2. BUSY (by server)
3. REQUEST (by requester)
4. ACK (by server) -- made it to handler, not accepted yet
5. DATA+ACCEPT (by server)
6. DATA+ACK (by requester)
7. ACK (by server)

When a BUSY handler is encountered, the retransmission rate is slowed slightly from its normal rate (by modifying the random backoff function). This adjustment saves network bandwidth by allowing some time for the handler to complete before wasting a retransmission.

We prepared two versions of SODA that approach piggybacking slightly differently. Normally, when an ACCEPT for a GET or EXCHANGE is issued soon after the REQUEST arrives, an ACK plus DATA is returned to the requester's kernel. The REQUESTER may have a new GET or EXCHANGE ready to send when notified of the REQUEST completion. The new REQUEST could then be piggybacked on the ACK for the data from the previous REQUEST. However, the server kernel is still awaiting an ACK for the data it sent so the piggybacked REQUEST will initially encounter a BUSY handler, causing a BUSY NACK to be sent to the requester ker-

nel. The ACK will be processed by the server kernel separately from the new REQUEST and the next retransmission by the requester kernel can then succeed if the server client exits the handler quickly.

In the other version (the *pipelined* version), when a REQUEST arrives at a busy server handler, a BUSY NACK is *not* issued immediately. Instead, the REQUEST (and data associated with it if it is a PUT or EXCHANGE) remains active in the input buffer for a short while. If the server handler becomes available quickly, the input buffer may still contain a valid REQUEST which can be immediately processed. In the non-pipelined implementation, a BUSY NACK followed by a retransmission of the REQUEST would be required. For handling steady streams of REQUESTS, the slight overhead associated with this method (e.g., the ENDHANDLER routine must check the input buffer for an active REQUEST) is well-justified (§5.5).

5.3. Broadcast REQUESTS

When a broadcast REQUEST is issued, a single packet is broadcast (using a special machine identifier recognized by all Megalink interfaces) to each node. Any node that recognizes the pattern will send an acknowledgement containing the MID of the matching node. It is important that these ACKS be staggered as otherwise several ACKS responding to the same broadcast REQUEST will collide. We implement this "staggering" by delaying an ACK to a broadcast REQUEST for an interval proportional to the (unique) machine id of each node.

5.4. Pattern and Transaction Id Generation

Our kernel implementation provides 48-bit PATTERNS. Because we lack associative hardware and because patterns are always checked on incoming

REQUESTS, we placed a restriction on the use of PATTERNS which allows pattern lookup in a single operation. The first eight bits of a PATTERN are used as the index into an array of 256 PATTERNS. The client is free to select any value for these eight bits, but if two patterns are advertised that are identical in the first eight bits, the second pattern overwrites the first.

GETUNIQUEID produces network-wide unique 40-bit patterns by concatenating an eight-bit serial number (unique to each machine) to a 32-bit *counter*. The counter is incremented each time a new unique pattern is created. The initial value of this integer is derived from a monotonic clock function maintained by the host VAX used for development. Each time a SODA kernel is rebooted a new initial value is set.

Some SODA systems may wish to make it difficult for clients to guess valid PATTERNS. For such an implementation, PATTERNS should be random, yet still guaranteed to be unique. One possible implementation method is to concatenate a number produced by a random number generator to the serial number/counter pair (§6.15).

SODA semantics require that an ACCEPT issued to a requester client which has crashed return an error. Transaction Id's in our implementation are generated from the same counter used to generate unique patterns. When a new client is booted, the current value of the counter is recorded. When an ACCEPT is issued, it is checked to ensure that it lies between the present (maximum) value of the counter and the (minimum) value of the counter recorded upon booting.

5.5. Performance Results

Our implementation requires about 8k bytes of text and data exclusive of 4k bytes used for the input and output buffers. Of this, about 4k bytes comprise the

Delta-t protocol. No particular attempt was made to optimize our code; we believe it could be made even smaller. The SODA kernel consists of about 1400 lines of C and 400 lines of assembler code of which the Delta-t protocol comprises about 750 lines of C code. The SODA kernel was rewritten several times over a period of approximately six months while the SODA design was changing. The final version took about six weeks to implement and debug.

Leblanc [9] implemented message-passing primitives in *MOD using identical hardware. We present some of his results for comparison with our SODA implementation. A B_SIGNAL in SODAL (§4.1.1) requires 8.5 ms (exclusive of client overhead) when the ACCEPT for the REQUEST is done in the server handler. If the B_SIGNAL REQUEST is queued by the server handler and ACCEPTED by the server task that is polling the queue of REQUESTS, the time increases to 10.0 ms (which includes 0.7 ms for the queueing overhead). This latter situation is semantically similar to a synchronous remote port call (with a single-integer argument) in *MOD, which requires 20.7 ms.

SODA SIGNALS (non-blocking) are more efficient (4.9 ms) because client REQUESTS may be queued by the kernel while the current REQUEST is being delivered. When non-blocking SIGNALS are used in conjunction with queueing at the server end, 5.8 ms are required. This is semantically similar to the *MOD asynchronous port call which requires 11.1 ms.

The first table (below) shows the speed of PUTS, GETS and EXCHANGES using both pipelined and non-pipelined versions of the kernel. All measurements were made with MAXREQUESTS set to three, and with the ACCEPTS being done immediately by the server handler. MAXREQUESTS values other than one produced the same results. With MAXREQUESTS set to one, all REQUESTS become blocking so no advantage due to double buffering accrues. Results in the tables

below are significant to the nearest millisecond. The timings for our comparisons to *MOD were made in a separate, more accurate test significant to the nearest 0.1 millisecond.

We present in the second table (below) a rough breakdown of the time spent to issue a SIGNAL. The context switch time includes both REQUEST arrival and REQUEST completion interrupts. The client overhead includes the overhead of maintaining the descriptor pool (which is locked and unlocked with CLOSE and OPEN) as well as the overhead for invoking the transmission primitives (REQUEST and ACCEPT). We derived the 4.9 ms time mentioned previously for SIGNALS by subtracting this client overhead from the total time to perform the SIGNAL.

5.5.1. Simulation Overhead

We expect that performance would be greatly enhanced by an actual SODA processor. We list some of the bottlenecks in the present implementation:

(1) The PDP-11 is a slow processor (about 170,000 instructions/second). A specialized processor should be able to perform the SODA functions much more efficiently.

(2) The Megalink is a slow network (1 Megabit). Its speed especially affects the performance of sending long messages. Some current proposals [65] promise dramatically increased bandwidth, extending into the 10 gigabit range.

(3) Associative pattern lookup hardware would not speed up our implementation because of the restrictions placed on patterns (§5.4). However, such hardware would allow for an efficient implementation which conforms to the exact SODA semantics as defined in §3.4.

(4) The client-kernel interface requires software interrupts for handler invocations and traps for SODA primitive invocations. In addition, the client must maintain a descriptor pool for SODA commands. Because the number of required descriptors is bounded (by MAXREQUESTS), the descriptor handling mechanism might well be placed in the kernel. Hardware interrupts and hardware control signals to invoke SODA primitives would eliminate much of the overhead in our present simulation.

(5) A network interface which copied messages into a high-speed interface buffer prior to copying into client memory could reduce the frequency of retransmissions due to BUSY handlers. In the extreme case, if one buffer were available for each node on the network, we could eliminate retransmissions due to BUSY handlers entirely. As discussed in §5.2.3, even a simulated interface buffer can greatly improve performance for streams of requests.

(6) It would be desirable to be able to disassemble or assemble packets (i.e., stripping or adding headers from/to data to form a packet) without having to first copy headers and data from/into a single contiguous region. This *scatter-gather* capability, already present on some available network devices such as the Pronet interface [66], improves performance by eliminating redundant copies. For example, on a request completion which carries data, the requester kernel could read the header into private kernel memory, and then store the data directly into client memory. It is best if the interface can specify the whole gather at once (which the Intel Israel 82586 [49] can do, but the Pronet can't). In our simulation, the entire packet must be copied into a kernel buffer before storing the data because there is no way to read the header (necessary to screen messages) beforehand.

SODA Performance

Milliseconds Per PUT (non-pipelined)
2 packets per PUT

words	0	1	100	200	300	400	500	600	700	800	900	1000
ms	7	8	11	16	19	23	27	31	35	39	43	47

Milliseconds Per PUT (pipelined)
2 packets per PUT

words	0	1	100	200	300	400	500	600	700	800	900	1000
ms	8	8	12	15	19	23	28	31	35	39	43	46

Milliseconds Per GET (non-pipelined)
4 packets per GET

words	0	1	100	200	300	400	500	600	700	800	900	1000
ms	7	16	20	23	28	32	35	39	43	48	52	55

Milliseconds Per GET (pipelined)
2 packets per GET

words	0	1	100	200	300	400	500	600	700	800	900	1000
ms	8	11	15	19	23	27	31	34	39	42	47	50

Milliseconds Per EXCHANGE (non-pipelined)
6 packets per EXCHANGE

words	0	1	100	200	300	400	500	600	700	800	900	1000
ms	7	22	32	44	57	65	75	86	96	107	117	128

Milliseconds Per EXCHANGE (pipelined)
2 packets per EXCHANGE

words	0	1	100	200	300	400	500	600	700	800	900	1000
ms	8	12	20	27	35	43	50	58	67	75	82	90

Breakdown of Communications Overhead

Breakdown of Protocol Time (ms) 2 packets per SIGNAL		
Connection Timers	1.0	
Retransmit Timers	0.7	
Context Switch	0.8	
Transmission Time	0.4	
Client Overhead	2.2	
Protocol Time	2.0	
Total Time	7.1	

5.6. Summary

We have shown that SODA is implementable in a space-efficient manner and can perform well. We also pointed out that SODA was implemented with a minimum of effort; a fact that speaks well for the simplicity of the design. Careful attention to piggybacking acknowledgements led to significant performance improvements. Under certain conditions, GETS and EXCHANGES can perform almost as well as PUTS, which opens up the possibility of actively writing to *or* reading from (actively obtaining data as opposed to passively receiving it) a resource using remote requests at about the same cost per transaction. Thus new styles of communication are possible when contrasted with a system which provides only active, unidirectional SEND and passive RECEIVE primitives. Black [87] has investigated the potential of providing passive and active I/O primitives in an object-based system. Finally, we illustrated ways in which a hardware SODA implementation could offer improved performance over our present simulation.

6. RATIONALE

In this chapter we discuss our reasons for the SODA design. Any good design always balances two competing goals: simplicity and expressive power. We wanted to make SODA simple so that an inexpensive and efficient hardware realization would be possible. At the same time, we wanted SODA to be usable by a large clientele. Successful designs such as the RISC [3] machine or the Pascal [2] language have concentrated on *excluding* features rather than adding new ones. Thus, this chapter focuses attention on why certain capabilities were *not* included in the SODA design.

6.1. Messages

We selected a message-based system as opposed to a shared memory-based one. Although the two forms are equivalent in some sense [68] and one can be implemented using the other, message-based primitives are higher-level in that they unify synchronization and data exchange [11]. Further, message-passing lends itself well to available local network technology which currently is less expensive and more expandable than existing shared-memory architectures which require costly switching networks.

6.2. Uniprogramming

We feel justified in calling a SODA client uniprogrammed because each processor executes one process and the handler does not maintain state between invocations. The handler is seen as a temporary suspension of the task activity, not an independent process. If multiple processes or coroutines are required as an organizational tool, they may be provided at the programming language level (as opposed to the operating systems level) as in SIMULA [69] or Modula [28].

There are many advantages to providing only one process per processor. If processors are cheap, it is economically possible to dedicate processors for real-time activities. Without the requirement to multiplex the CPU, there is no need for an operating systems kernel to:

(1) Provide a complex operating system to schedule processes.

(2) Multiplex and demultiplex messages among different processes.

(3) Manage process contexts and buffers which must be locked into memory during IPC requests.

(4) Provide separate local and remote message passing mechanisms.

We can also finesse some complexity by making primitives, such as ACCEPT and CANCEL, blocking. In a multiprogrammed system, it would be unacceptable to delay all users while the operating system was servicing one users's I/O request, because the goal of the operating system is to provide the illusion of a dedicated machine to each user. In a uniprogrammed system, no such illusions are necessary. Whether or not these blocking primitives cause a loss of efficiency for some applications is a separate issue. The important issue is that primitives can be made blocking only if their blocking nature cannot lead to uncorrectable deadlocks. Because ACCEPT and CANCEL are guaranteed to complete in a bounded amount of time, they cannot cause any deadlocks which are impossible to break.

We should point out that process migration is still useful, although its use as a process load balancing technique is obviated. For example, a program may be compiled on a machine attached to a disk containing the program text, then move to a high-speed processor to perform numerical tasks, and ultimately migrate to a processor attached to a printer to produce output.

6.3. Process Creation and Termination

The same basic mechanism used to send messages is used to create processes: A client DISCOVERS a free processor with the desired attributes and issues REQUESTS to it which contain the core image. No special process creation primitives are required. The booting mechanism thus conforms well to our desire for uniformity.

It is necessary to provide special mechanisms to terminate a process because otherwise a malicious process could CLOSE its handler and loop forever in its task, effectively removing that processor from the system. We provide two KILL patterns associated with a client termination action. One of these (the KILL PATTERN) is intended for use by a system manager process to reclaim processors from application processes which do not respond to system REQUESTS. The other (LOAD PATTERN) is intended for use by the parent of a process to terminate a runaway child.

6.4. Connectionless Protocols

There are two reasons for establishing connections: To maintain state (i.e., sequence numbers) for reliable transport, and to represent the logical endpoints of a conversation.

6.4.1. Maintaining State

SODA uses an underlying connectionless protocol (the Delta-t protocol discussed in §5.2.2) to achieve reliable communication as contrasted with connection-based protocols such as TCP (§2.4.6) that require connections to be *explicitly* established. Because many transactions in a local distributed environment tend to be single-shot exchanges [70] that do not justify the cost of

establishing a virtual circuit, we have opted for a connectionless implementation.

There are three main disadvantages to using the Delta-t connectionless protocol as compared to a three-way handshake [63]. One is that space for connection state cannot be reclaimed as quickly. Connection closing in Delta-t can require more time than in the three-way handshake by an amount proportional to MPL. In a network where packets are routed between nodes, MPL can become quite significant. In a local area network however, MPL is typically quite small. Further, there is no difficulty if space for connection records can be preallocated. In our SODA implementation, each node preallocates one connection record for each machine on the network. This approach may be feasible for small networks (§6.2.2). The second is that the Delta-t protocol requires a delay proportional to MPL for crash recovery before the network can be rejoined. Again, if MPL is small this is not an important consideration. The third is that MPL must be bounded in order for the protocol to work properly. It may be difficult to establish upper bounds on MPL in a store-and-forward network.

6.4.2. Logical Connections

As pointed out in §4.2.4, SODA can be used to implement a connection-based link protocol. In an environment where broadcast is expensive due to hardware limitations, it may make sense to supply point-to-point connections. In such an environment, *switchboards* will be used to obtain logical connections. However, when broadcast is available, it provides an extremely valuable connection-making tool because logical connections can be established by making inquiries of all clients on the network in parallel. No centralized switchboards are then needed.

The DCS system [47] provides a completely connectionless (dynamic) process-binding mechanism in that all addressing is done by associative pattern

lookup. This sort of addressing may be impractical for very high speed interfaces because the interface must make an associative comparison for screening (§6.12). The Arachne system [34] provides a completely connection-based (static) process-binding mechanism (links) which requires connections to be established by the ancestors of a process. SODA takes an intermediate view. By using DISCOVER, a connection may be dynamically determined. However, all message exchange requires a specific machine identifier to be used. In this way we obtain the benefits of reliable message exchange and the flexibility of dynamic interconnection.

6.5. Failure Handling

In the presence of processor failures, SODA ensures only that a processor that fails detectably will not keep an uncompleted transaction waiting. Higher-level protocols such as the *two-phase commit* [71] may be employed to maintain data integrity in the presence of crashes.

Because clients can provide their own crash detection protocols, we did not build in a crash detection scheme. If two clients have been communicating and one of them crashes and recovers, the living client will not be aware of the crash unless the living client attempts to communicate with the recovering client before the recovering client can readvertise its patterns. If it is important to inform the living client of the recovering status of client which crashed, the latter may delay advertising its patterns until it has agreed with the former about its status. All clients could advertise a pattern which is bound to a crash recovery action; this pattern would be the only one available immediately upon crash recovery. If a client recovers from a crash, it sends a REQUEST, using the crash recovery pattern, to its partner. When the REQUEST is ACCEPTED, the recovering partner may

readvertise its service patterns. This scheme will also work if both partners crash and recover.

We provided CANCEL primarily to provide transactions that have finite patience. Timeouts were not included in the SODA message passing primitives because the majority of clients cannot know what realistic waiting intervals are. Further, it is easy to add timeout services in SODA (§4.4.3, §4.3.2).

6.6. Asynchronous Receipt

Asynchronous receipt was provided primarily because we wanted to provide a flexible scheduling capability. In addition, by polling a variable that is set in the handler it is easy to simulate blocking receives. With blocking receive it is impossible to be informed of exceptional conditions (such as timeouts or deadlocks or high-priority requests) that necessitate the termination of a receive. Finally, the ability to receive asynchronously prevents the deadlock situation which arises when two clients attempt to issue blocking sends to each other at the same time.

Although the ability to inquire about available messages (timeouts on receive give the same capability) in a system with blocking receive alleviates some scheduling problems, this kind of polling can be very inefficient. Some programs such as the checkers program described in [72] enjoy improved performance because the overhead needed to poll for an event is not present. The situation in the checkers program is this: A process P uses a variable v. If a "better" value of v is discovered by a process Q, P would like to start using the new value as soon as possible, but P doesn't want to waste time polling for a message from Q that could arrive at any time (or never). In SODA, the handler can simply update v when update messages arrive.

6.7. Flexible Scheduling

Communication Ports [29] permit the server to *schedule* the replies to requests; that is, other requests may be processed by a server before it replies to the first request. SODA uses a similar idea: The notification given the server of an incoming transaction (REQUEST arrival at the server handler) and the notification given the requester of a completed transaction (ACCEPT arrival at the requester handler) are separate events with the latter action under control of the server. A good example of the power of flexible scheduling is seen in our implementation of Bernstein's algorithm [21] (§4.2.5.1). Here it is easy to delay ACCEPTING a REQUEST when a possible deadlock exists until the REQUESTS have been ordered by machine id. Another example is the readers-writers solution given in §4.4.4. When a write finishes, the readers queue is flushed. All the while, new read and write requests may arrive. It is then a simple matter to schedule an incoming write request ahead of *new* read requests.

Systems that provide built-in queueing of incoming messages (ports) usually give a single FIFO queue, a (fixed, finite) set of FIFO queues, or at best a priority queue or queues. By using a two-phase scheme (first the REQUEST, later the ACCEPT) we allow the user to tailor the queueing discipline to the application while maintaining the simplicity of *not* buffering messages in the kernel.

6.8. Two-way data transfer

We believe that a transaction should be capable of causing data to be sent in either direction (from requester to server or vice versa) because it is natural to be able to either read or write from a server. We have shown that two-way transfer can be efficiently implemented and that for streams, GETS can be about as efficient as PUTS. Simultaneous two-way transfer (EXCHANGE) is quite useful

for short two-way transactions such as file opening (§4.4.5).

6.9. Non-blocking Send

Non-blocking send is important to provide in a uniprogrammed environment because the cost of implementing non-blocking send from blocking send would be high. To implement non-blocking send with blocking send, a blocking send is issued to an auxiliary process which immediately accepts the message (thus unblocking the sender). This auxiliary process then tries to deliver the message on behalf of the original sender. In a uniprogramming environment, the cost of providing the auxiliary process is the same as the cost of another processor. We feel this is too dear a price to pay for a simple and possibly frequent operation. Further, blocking send is easily and cheaply implemented using non-blocking send (§4.1.1).

There are many potential uses for non-blocking send. With non-blocking send, a client may issue a request for data before the data is actually required. Such anticipatory requests could be used, for example, to implement pre-paging from a utility disk server. Non-blocking send is also important for programs which require a high degree of parallelism. The double-buffering scheme employed in §4.4.1 is an example. Finally, utility servers should be able to issue requests without fear that a malicious client can refuse to receive. With non-blocking send, a server may always take an alternate action with respect to any uncompleted send.

6.10. Synchronous ACCEPT and CANCEL

We did not make ACCEPT asynchronous because we can guarantee an upper bound on the time ACCEPT takes to complete: Buffers have already been allo-

cated; data will be transferred in an amount of time bounded by the algorithm used by the network to ensure fair access to the line in the presence of contention such as binary exponential backoff or token passing [73]. To make ACCEPT asynchronous would add a third kind of handler invocation as well as slowing down ACCEPT because of the extra interrupt. The kernel would also be complicated by the necessity of queueing ACCEPT completion interrupts. CANCEL will also complete in bounded time and is synchronous for the same reasons ACCEPT is. Making ACCEPT asynchronous is attractive, however, because message transfer could then take place in parallel with server activity. Further research is needed to determine how important an asynchronous ACCEPT would be to client efficiency; or, more to the point, what client programs could profitably do if they could execute in parallel with ACCEPT. Our results show that the overhead of an ACCEPT can be quite significant, especially when the ACCEPT is issued well after a REQUEST has arrived and data from a PUT or EXCHANGE could not be piggybacked with the REQUEST. If SODA were converted to a multiprogrammed system, there is little doubt that ACCEPT would have to be made asynchronous.

6.11. Selective Receipt

When a REQUEST arrives at the client handler, the client is provided with the sizes of the buffers involved, the pattern the handler was invoked with, the MACHINE ID and TID of the sender (REQUESTER SIGNATURE), and an argument. We call this information the *tag* of a REQUEST. The buffer sizes allow the server to perform flow control: A REQUEST need not be ACCEPTED until a buffer of the correct size is available. The invoked pattern allows the server to select an operation relevant to the REQUEST. The REQUESTER SIGNATURE provides ACCEPT with a return address and allows some security (messages from unauthorized senders may be REJECTED). The argument may be used to discriminate different

uses of the pattern, provide a priority on a message, give simple type information about the messages being sent; it may even be used to send a short message (e.g., a character from a terminal). Because there are so many uses of the single integer argument we elected to provide it in SODA. For the sake of symmetry, and to allow error returns (such as REJECT) from an ACCEPT, ACCEPT also includes an argument seen by the requester's handler.

Some systems require that an entire message be read before a scheduling decision is made. When messages are selected on the basis of complex data types (i.e., more type information than can readily be associated with the REQUEST argument) or arithmetic expressions over the contents of the message as in Synchronizing Resources [74] the information supplied to the handler in the tag is not sufficient to make a scheduling decision. Higher level protocols such as the remote procedure call discussed in §4.2.2 must then be employed by SODA clients in order to read the message before releasing the sender.

There are two reasons for providing a short tag. The first is that the tag will be retransmitted frequently when the server handler is BUSY or CLOSED and we do not wish to consume excessive network bandwidth. The second is that the server must guarantee that space exists to buffer the tag. We did not want to require that the client preallocate a lot of memory simply to meet the SODA specifications. LEO [42,43] allows data to be transferred before releasing the requester. However, in LEO, the tag is 150 bytes and each server must preallocate one tag buffer for each machine on the network. In a large network, this could represent a considerable outlay of client space. Further, if only a single tag were buffered, 150 bytes would likely consume a lot of network bandwidth due to the necessity of retransmissions when the buffer is full.

6.12. Screening

SODA provides two screening mechanisms. First, all messages specify a specific processor (by MID) which allows rapid processing by the network interface (i.e., a single comparison) so that most spurious message traffic can readily be rejected. This type of screening is important when very high speed networks are employed. Second, SODA checks the PATTERN in an address. This form of screening is done after the message has already been delivered to the kernel (and therefore can be done by less expensive hardware) because improper patterns require an error indication to be returned to the requester. Improper MID'S in an address are simply ignored.

Pattern screening supports once-only service (a server that accepts a single message from one of a set of requesters can UNADVERTISE its pattern when the first REQUEST arrives) as well as *load control*. If a server becomes swamped with requests and other servers are available that use the same pattern as an entry, UNADVERTISING the pattern will force requesters to DISCOVER other servers using the same pattern. Another use for pattern screening is the support of process groups. Clients within a group can be assigned a set of unique ids by a resource manager which are then ADVERTISED by the individual clients. Attempts from clients outside the group to send messages to clients inside the group will be screened out.

It is important to have the pattern screening performed by the kernel for DISCOVER to work without necessitating client intervention. Broadcast in SODA is not guaranteed to be reliable but the kernel should make an effort to make it as reliable as possible. By providing the DISCOVER mechanism in the kernel, we at least remove current client state from further affecting reliability. We also can take advantage of associative kernel hardware which clients can not be

expected to provide.

6.13. Bufferless Kernel

In keeping with our desire to make the kernel small, the SODA kernel does no buffering of messages. As a result, fairness is the responsibility of the client. If the client does not service interrupts quickly, access to the handler becomes more random. Because SODA provides flexible scheduling of ACCEPTS, it is feasible to implement fast handlers that merely enqueue an incoming REQUEST for later service by the task. By not putting buffering in the kernel we leave the issue of flow control up to the client. Thus, it is important that the size of an incoming message be available as a parameter to a handler invocation.

6.14. Patterns

We chose an extremely simple pattern matching facility: The ability to advertise a set of fixed-length patterns. From this, a client has the power of regular expressions on a fixed-length string. The SODA hardware will require fast pattern comparison machinery, and very simple associative pattern-matching hardware can be used to find exact matches on fixed-length bit strings. Another reason for not providing powerful regular expressions is to maintain security. Powerful pattern-matching primitives would permit easy access to valid signatures with

DISCOVER(" * ") — "*" matches all patterns

More complex naming strategies (such as name hierarchies or name retrieval within a given environment) can be provided by a name server client. DISCOVER gives enough power to locate plausible servers which can support higher level name binding facilities.

The ability to generate unique patterns is important to provide local process groups and server entry points. Because GETUNIQUEID returns fewer than PATTERNSIZE bits, well-known names in which semantics are assigned to particular fields within a pattern can be established by reserving a bit in the pattern to indicate the presence of a well-known name. Well-known names can therefore be used without any fear of conflicting with random patterns if all clients follow this protocol.

6.15. Security

SODA was not designed to provide a high level of protection. Nonetheless, there are several mechanisms which can be used to restrict unauthorized client access:

(1) An ACCEPT from a server other than the one specified in the associated REQUEST will fail.

(2) The client may use its own random number generator to create patterns. If PATTERNSIZE is large enough that GETUNIQUEID returns a pattern containing far less than PATTERNSIZE bits, the client can use its own random number generator to provide part of a pattern and still ensure that the pattern is unique with respect to other clients using GETUNIQUEID.

(3) The MID of the REQUESTER is unforgable and given to the server on handler invocation. The server may elect to REJECT those REQUESTS issued from unauthorized nodes.

(4) RESERVED PATTERNS (§3.4.3) cannot be UNADVERTISED (allowing a client to permanently reserve a node) or ADVERTISED (allowing a client to masquerade as a free node in order to accept boot requests from another client).

(5) The node with MID 0 can alter RESERVED PATTERNS. Therefore, the ability to boot and kill processes can be restricted by a system manager who has access to the privileged processor.

6.16. DISCOVER returns a list

DISCOVER requests return a *list* of clients with matching patterns. The implementation will broadcast the DISCOVER query to all sites and wait an interval sufficient for many responses to arrive. If only a single response was allowed, it might be impossible, due to line characteristics, to locate SERVER SIGNATURES for several clients advertising a given signature (e.g., the client processor physically closest to the requester is always discovered first). Because a set of clients providing similar yet different services could ADVERTISE the same pattern, an unfair DISCOVER scheme could hide desired services from a client. Location-dependent DISCOVER responses could also hinder attempts by clients to distribute requests evenly among a set of resources, either to balance load or to broadcast information.

6.17. Primitives Not Provided

In this section we discuss some primitives that are potentially useful but which are not included in SODA. SODA includes those primitives that we feel are the most generally useful and the ones we present here can be implemented as library routines on top of SODA or could easily be added to the SODA kernel if desirable for efficiency reasons.

6.17.1. Multicast

Some proposals, such as Leblanc's [9], provide a single *multicast* primitive which will reliably send a single message to each member of a process group. There also exist reliable broadcast techniques [75] that can provide specialized services (such as maintaining backups of transactions or updating databases that support multiple copies). Such primitives take advantage of the particular broadcast capabilities of a local area network to reduce the number of transmissions required and may offer significant performance advantages over a group of individual, reliable transmissions.

In SODA, if a client wishes to send a message reliably to several sites in a group, it must issue a separate REQUEST to each site. Thus far, we have not seen enough practical uses for efficient reliable broadcast in a loosely-coupled network to justify complicating SODA by providing a special reliable broadcast primitive.

6.17.2. Remote Memory Reference

Spector [7] has advocated implementation of specialized primitives to perform remote memory reference (RMR) in a message-based distributed environment. However, his results show that even a highly optimized implementation provides remote memory references which are two to three orders of magnitude slower than local memory references. Shared-memory distributed architectures are capable of speeds within an order of magnitude of local memory references [76]. We feel that applications which rely on a remote memory reference paradigm (i.e., those that require fine-grained parallelism) can not execute efficiently on a message-based architecture.

A more efficient RMR service could be added to SODA by providing a kernel handler that services PEEK and POKE REQUESTS. This handler would be associated with a RESERVED PATTERN (§3.4.3) in the same way that the KILL PATTERN is. OPEN and CLOSE would affect the state of the kernel handler to provide synchronization. More highly optimized PEEK and POKE primitives could be provided: These would block until the PEEK or POKE returned, thus avoiding the overhead of a completion interrupt.

6.17.3. Datagrams

SODA does not support unreliable messages (datagrams). While any sort of protocol can be implemented on top of a datagram service, the additional layering causes performance degradation. As an example, the PUP protocol [77] provides a wide variety of services layered on top of a datagram service. However, each layer in the PUP protocol causes a noticeable drop in performance from the preceding layer. We also feel that most applications will require reliable message service. Nonetheless, there are applications (such as voice transmission or disk-to-disk copy) which can be implemented very efficiently using a datagram service.

6.17.4. Multipackets

Arbitrarily long transmissions are supportable by higher-level protocols that packetize and reassemble large blocks of data. Our experience with typical transactions shows that most transmissions tend to be either small chunks or large streams (e.g., a whole file). Although it may be possible to achieve better performance (e.g., by using windowing techniques) if the kernel permits arbitrarily large messages, our experience with sending streams of data indicates that perfor-

mance can be quite good when done by the client (§6.5).

6.17.5. Bidding Support

A client which has advertised a service may be only one of a community of servers that provide the same service. Such a client can be DISCOVERED and then used. DISCOVER returns a list of potential servers however and there is no way to discriminate among the members of the list. By allowing the client to ADVERTISE values which are returned as part of a broadcast REQUEST along with MIDS, a server could indicate how busy it is. Then a requester should select the least heavily loaded server from the list.

6.17.6. REQUEST Forwarding

A potentially useful service would be to allow a server other then the one that received the REQUEST to ACCEPT it. This would be similar to the forwarding capability of Thoth [37]. We have already shown (§4.2.4) how link protocols can be implemented in SODA which allow ends of a logical connection to move transparently. Therefore providing a similar (but less powerful) ability in SODA does not seem justified.

6.17.7. Fine-grained Synchronization

CLOSE prevents all access to the handler. Allowing the handler to remain OPEN for specific SERVER SIGNATURES would allow for more finely-grained synchronization control. CLOSE could be parameterized to include a pattern that would leave the handler open to requests not using this pattern. This effect would not be the same as UNADVERTISING the pattern because requesters using an unadvertised pattern are returned a failure indication, whereas requests

attempting access to a closed handler are kept on "hold" until the handler reopens. So far, we have not found this feature necessary for writing programs in SODA.

6.18. Conclusions

SODA, besides providing the basis for a distributed programming language (SODAL), can also be used as an operating systems kernel in a distributed environment. Clients are expected to provide their own device drivers and interrupt handlers. Virtual memory, if available, is the concern of the client. No multiprogramming is provided, and typical operating systems functions (such as file service) are supplied by utility processes. Finally, incoming messages are delivered to the client without regard to any kernel-defined queueing discipline. SODA only provides process management and interprocess communication facilities; that is SODA provides *mechanisms* for allowing processes to cooperate in a distributed environment, leaving *policies* to the clients [78].

We have rigorously excluded features that we believe do not add significantly to the power of SODA. As a result, SODA presents a very small number of primitives that can be implemented efficiently. We feel that SODA has succeeded in the effort to provide only those services which are useful to the majority of clients. We do not provide services that will be used infrequently and that non-users of the services would be required to pay for in added cost or decreased performance.

7. SUMMARY

Line-level protocols such as X.25 [79] are no longer sufficient for the needs of sophisticated applications programs and the requirements of high-speed local networks. The goal of our work was to show that a communications adaptor (SODA) which incorporates some of the power of an operating systems kernel and which is simple to use and implement is a feasibility. SODA thus represents one of the next generation of communication adaptors.

SODA combines ten primitives into a unique design which has exceptional expressive power. In chapter 4 we demonstrated the power of SODA by illustrating its application to real problems in local networking. We showed that an efficient and compact implementation was possible in chapter 5. Our implementation experience strongly suggests that a SODA processor could be constructed inexpensively. Finally, we illustrated the rationale behind the SODA design in chapter 6. The rationale is important because it shows that operating systems design can be guided by a logical, rather than an *ad-hoc*, process.

7.1. Contributions to Computer Science

Our work makes three major contributions to the field of distributed operating systems research:

(1) We showed that an inexpensive communications interface can be built which supplies sufficient power for processes to cooperate in a distributed environment without additional kernel support. Because one less layer of support is required, distributed systems can be made to perform better and at a reduced cost.

(2) We explored the kinds of communications primitives that are necessary to support high-level applications in a system consisting of networked *unipro-grammed* processors.

(3) Our implementation shows that active RECEIVE and EXCHANGE primitives can be implemented with approximately the same performance as active SEND. As a result, new styles of interprocess communication may become feasible.

In addition, we presented a novel solution to Dijkstra's "dining philosopher's" problem.

7.2. Directions for Future Work

We expect that SODA is sufficiently simple to readily implement in VLSI. Therefore, a major next step would be a VLSI SODA implementation which would make it possible for even inexpensive processors to possess a powerful communications adaptor.

SODA may be thought of as a proposal for a local network protocol standard. Many more applications programs need to be written using the SODA primitives to give us more confidence that SODA indeed suffices for most needs and that extensions, such as those proposed in chapter 6, are unnecessary. Our initial experience shows us that SODA is more than adequate for typical high-level distributed applications.

Because SODA makes so few requirements of the client processor and because many client processors should be available in a SODA network, it may prove advantageous to design extremely simple client processors for use in a SODA system. A client processor with no memory management and only a SODA processor for communication could be constructed cheaply. A SODA network containing a very large number of inexpensive processors would then be feasible and

present a significant amount of computing power to the network user community.

The employment of highly-tailored processors in a network environment will soon be a reality because of advances in VLSI technology. Specialized server processors optimized for a particular application might well take advantage of a completely distributed network architecture, if a powerful communications adaptor were available to provide a uniform network interface. The communications adaptor would play a crucial role in providing efficient access to these servers.

Finally, we would be interested to explore how useful the SODA primitives are in other environments. It should be possible to implement SODA on a shared-memory architecture. By doing this, it may be possible to take advantage of the higher communication speed for those applications which can afford the expense of shared-memory, and still provide the high-level SODA primitives to the client. It should also prove possible to implement a kernel for a multiprogrammed machine where each process appears to have its own logical SODA interface. Thus, SODA may have utility as a general model for interprocess communication independent of architectural or multiprogramming constraints.

8. BIBLIOGRAPHY

[1] B. W. Lampson, M. Paul, and H. J. Siegert, "Distributed Systems - Architecture and Implementation," *Lecture Notes in Computer Science*, Springer Verlag Lecture Notes in Computer Science, (1981).

[2] K. Jensen and N. Wirth, "Pascal: User Manual and Report," *Lecture Notes in Computer Science 18*, Berlin, New York; Springer-Verlag, (1974).

[3] D.A. Patterson and C.H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *SIGARCH Newsletter 9*, 3, pp. 443-458 (May 1981).

[4] Per Brinch Hansen, *Operating Systems Principles*, Prentice Hall (1973).

[5] Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice Hall (1977).

[6] F. E. Heart, R. E. Kahn, S. M. Ornstein, W. R. Crowther, and D. C. Walden, "The interface message processor for the ARPA computer network," *Proc. AFIPS Spring Joint Conf. 36*, pp. 551-567 (1970).

[7] A. Spector, *Multiprocessing Architectures for Local Computer Networks*, Stanford Tech. Report STAN-CS-81-874 (1981).

[8] P.V. Mockapetris, "Communication Environments for Local Networks," *USC/Information Sciences Institute Research Report ISI/RR-82-103*, (1982).

[9] T. J. Leblanc, "The Design and Performance of High-Level Language Primitives for Distributed Computing," University Of Wisconsin Technical Report 492, Ph.d. Thesis (1982).

[10] P. Cashin, "Inter Process Communication," *Internal Memorandum*, Bell-Northern Research, (1980).

[11] Andrews, G. R. and F. B. Schneider, "Concepts and Notations for Concurrent Programming," TR 82-520, Department of Computer Science, Cornell University (September 1982).

[12] J. Bacon, "An Approach to Distributed Software Systems," *Operating Systems Review 15*, 4, (1981).

[13] R.A. Finkel, "Tools for Parallel Programming," *Internal Memorandum*, University of Wisconsin, (1981).

[14] M.J. Plasmeijer, "Input Tools: a language model for interaction and process communication," *Ph.D thesis*, Katholieke Universiteit Nijmegen, (1981).

[15] J.B. Goodenough, "Exception Handling: Issues and a Proposed Notation," *CACM 18*, 2, (1975).

[16] L.M. Casey, "On Kernels for Distributed Computer Systems," *U. of Edinburgh Tech. Rept. CSR-27-78*, (1978).

[17] E.W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," *CACM 18*, 8, (1975).

[18] A.N. Habermann, "Path Expressions," *Computer Science Tech. Report*, Carnegie-Mellon University, (1975).

[19] Fabry, R. S., "Capability Based Addressing," *CACM 17*, 7, pp. 403-412 (July 1974).

[20] C.A.R. Hoare, "Communicating Sequential Processes," *CACM 21*, 8, (1978).

[21] A. Bernstein, "Output Guards and Nondeterminism in 'Communicating Sequential Processes'," *ACM Transactions on Programming Languages and Systems 2*, 2, (1980).

[22] Buckley, G.N. and Silberschatz, A., "An Efficient Implementation for the Generalized Input-Output Construct of CSP," *Transactions on Programming Languages and Systems 5*, 2, (1983).

[23] Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," *Communications of the ACM 21*, 11, (November, 1978).

[24] Jan van den Bos, M.J. Plasmeijer, and J. Stroet, "Process Communication based on Input Specifications," *Transactions on Programming Languages and Systems 3*, 3, (1981).

[25] J. D. Ichbiah and O. Roubine, "Rationale for the Design of the ADA Programming Language," *Sigplan Notices 14*, 6, (1979).

[26] R.P. Cook, "*MOD--A language for Distributed Programming," *Proc. 1st Intl. Conf. on Distributed Computing Systems*, (1979).

[27] R.P. Cook, "Schedulers as Abstractions," *Computer Sciences Tech. Report 393*, University of Wisconsin, (1980).

[28] Niklaus Wirth, "A Language for Modular Multiprogramming," *Software-Practice and Experience 7*, (1977).

[29] Mao T.W. and Yeh R.T., "Communication port: a language concept for concurrent programming," *IEEE Transactions on Software Engineering 6*, 2, pp. 194-204 (1980).

[30] B. Liskov, "Linguistic Support for Distributed Programs: A Status Report," Computation Structures Group Memo 201, MIT Laboratory for Computer Science (October 1980).

[31] B. Liskov, "Primitives for Distributed Computing," *Proceedings of the Seventh Symposium on Operating Systems Principles*, (1979).

[32] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS 5*, 3, (July 1983).

[33] J.A. Feldman, "High level Programming for Distributed Computing," *CACM 22*, 6, (1979).

[34] R. A. Finkel and M. H. Solomon, "The Arachne Distributed Operating System," Technical Report 439, University of Wisconsin--Madison Computer Sciences (July 1981).

[35] Anthony S. Bolmarcich , "Arachne Makeroutes Utility Project ," *Class Project for CS 736*, (1982).

[36] H.P. Katseff, T.B. London, and C.S. Roberts, "Gaggle-A multi-Processor Operating system Based on Communication Channels," *Bell Laboratories Internal Memorandum 81-11353-8*, (1981).

[37] D. R. Cheritan, M. A. Malcolm, L. S. Melen, and G. R. Sager, "Thoth, a portable real-time operating system," *CACM 22*, 2, pp. 105-115 (February 1979).

[38] R.A. Finkel and M.H. Solomon, "Part IV of the First Report on the Crystal Project," *University of Wisconsin Internal Document*, (1983).

[39] R. Cook and R. Finkel, "Part I of the First Report on the Crystal Project," *University of Wisconsin Technical Report*, 499, (1983).

[40] R. Swan, S. Fuller, and P. Siewiorek, "Cm* a Modular, Multi-microprocessor," *Proc. NCC, AFIPS*, (1977).

[41] A. K. Jones et. al., "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proc. of the Seventh Symposium on Operating Systems Principles*, (Dec. 1979).

[42] Peter Bittmann, "The Architecture of the Personal Computerized Workstation LEO," *Internal memorandum LEO G-7.7.3*, University of Munich, (1982).

[43] G. Seegmueller, "A Short Description of LEO Hardware and Software Details," *Internal memorandum LEO G-7.7.2*, University of Munich, (1981).

[44] Heigert, Personal Communication. (1982).

[45] G. Cox and W. Corwin, "A Unified Model and Implementation for Interprocess Communication in a Multiprcessor Environment," *Proc. of Eighth Symposium on Operating Systems Principles 15*, 5, (Dec. 1981).

[46] Baskett, F., J.H. Howard, and J.T. Montague, "Task Communication in Demos," *Proc. of the Sixth Symposium on Operating Systems Principles*, pp. 23-31 (Nov. 1977).

[47] D.J. Farber and K. Larson, "The Distributed Computing System," *Internal Technical Report*, University of California, Irvine, (1972).

[48] P.V. Mockapetris, M.R. Lyle, and D.J. Farber, "On The Design of Local Network Interfaces," *IFIP 77*, (1977).

[49] M. Stark, A. Kornhauser, and D. Van-Mierop, "A High Functionality VLSI LAN Controller for CSMA/CD Networks," Compcon, Intel Israel LTD. (1983).

[50] J. Postel (ed.), "Transmission Control Protocol," ARPA Network RFC 793, USC/Information Sciences Institute (1981).

[51] J. Postel (ed.), "Internet Protocol," ARPA Network RFC 791, USC/Information Sciences Institute (1981).

[52] Quanta Microtique, "Preliminary Report on the QM10 Advanced Communications Controller," *Washington, D.C.*, (1983).

[53] B. W. Kernigan and D. M. Ritchie, "The C Programming Language," *Prentice-Hall*, (1978).

[54] R. M. Bryant, "SIMPAS User Manual," Technical Report 391, University of Wisconsin--Madison Computer Sciences (June 1980).

[55] B. J. Nelson, "Remote Procedure Call," CMU-CS-81-119, Carnegie-Mellon Technical report (1981).

[56] E.I. Organick, "The Multics System: An Examination of its Structure," Mit Press (1972).

[57] E.W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica 1*, Springer Verlag, (1971).

[58] P.J. Courtois, F. Heymans, and D.L. Parnas, "Concurrent Control with Readers and Writers," *CACM 14*, 10, (1971).

[59] Computrol Inc., Ridgefield CT., Megalink Manual.

[60] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *CACM 17*, 7, pp. 365-375 (July 1974).

[61] A. Tanenbaum, "Computer Networks," *Prentice-Hall, Inc.*, (1981).

[62] C. A. Sunshine and Y. K. Dalal, "Connection Management in Transport Protocols," *Computer Networks 2*, 6, (1978).

[63] J. G. Fletcher and R. W. Watson, "Mechanisms for a Reliable Timer-Based Protocol," *Computer Networks 2*, (1978).

[64] R. W. Watson, "Timer-Based Mechanisms in Reliable Transport Protocol Connection Management," *Computer Networks*, 5, (1981).

[65] H. F. Taylor, "Multi-processor bus architecture," ERC41015.4FR, Rockwell International (June 1980).

[66] Proteon Associates, Operation and Maintenance Manual for the Pronet (TM) Local Area Communications Network. (1982).

[67] Andrew Black, "An Asymetric Stream Communication System," *Proceedings of the Ninth Symposium on Operating Systems Principles*, (1983).

[68] H.C. Lauer and R.M. Needham, "On the Duality of Operating System Structures," *Operating Systems Review 13*, 2, (1979).

[69] G.M. Birtwistle, O. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*, Petrocelli/Charter (1973).

[70] Willy Zwaenepoel and Keith A. Lantz, "Perseus: Retrospective on a Portable Operating System," *Software: Practice and Experience (submitted)*, (December, 1982).

[71] B. W. Lampson and H. K. Sturgis, "Crash Recovery in a Distributed System," *Xerox PARC Memorandum*, (April, 1979).

[72] J. P. Fishburn and R. A. Finkel, "Parallel Alpha-Beta Search on Arachne," Technical Report 394, University of Wisconsin--Madison Computer Sciences (July 1980).

[73] F. A. Tobagi, "Multiaccess protocols in packet communication systems," *IEEE Transactions on Communications COM-28*, 4, pp. 468-488 (April 1980).

[74] G. R. Andrews, "Synchronizing Resources," *ACM Transactions on Programming Languages and Systems 3*, 4, (1981).

[75] J. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols," *Bell Laboratories Internal Report*, (1982).

[76] R. Rettberg, "Development of a voice funnel system," Report No. 5284, Bolt, Baranek, and Newman, Cambridge, Mass. (April, 1983).

[77] D. R. Boggs, J. F. Schoch, E. A. Taft, and R. M. Metcalfe, "Pup: An Internetwork Architecture," *IEEE Transactions on Communications 28*, 4, (1980).

[78] R. Levin et. al., "Policy/Mechanism Separation in Hydra," *Proceedings of the Sixth Symposium on Operating Systems Principles*, (1977).

[79] A. Rybczynski, "X.25 Interface and End-to-End Virtual Circuit Service Characteristics," *IEEE Trans. Commun. 28*, (April, 1980).