THE ARACHNE DISTRIBUTED OPERATING SYSTEM

by

Raphael A. Finkel
Marvin H. Solomon

Computer Sciences Technical Report #439

July 1981

THE ARACHNE DISTRIBUTED OPERATING SYSTEM[1,2]

by

Raphael A. Finkel and Marvin H. Solomon

University of Wisconsin--Madison

ABSTRACT

Arachne is multicomputer operating system implemented at the University of Wisconsin that allows a network of microcomputers to cooperate to provide a general-purpose computing facility. After presenting an overview of the structure of Arachne, this paper reports on experience with Arachne, compares it to other multiprocessor and multicomputer operating systems, and indicates our future plans.

---

[1]This research was supported in part by the United States Army under contract #DAAG29-75-C-0024 and by the Defense Advanced Projects Research Agency under Navy contract N00014-81-C-2151.

[2]A preliminary version of this paper appeared as reference [1].

TABLE OF CONTENTS

1. INTRODUCTION

Recent trends in the economics of computer hardware have led many researchers to investigate the possibility of constructing both special-purpose and general-purpose computers out of many independent processors connected by a local communications network [2,3,4,5,6,7,8,9,10,11,12]. Numerous problems of software design must be solved in order to accomplish the goals of these architectures. Arachne[3] is an experimental operating system designed to investigate some of these problems. This paper describes Arachne, reports on our experiences with it, compares it to other operating systems, and indicates our future plans based on these experiences.

The principal goal of the Arachne project is to develop software techniques for organizing large numbers of independent processors into a network that appears to the casual user to be a single general-purpose computer. We call such a machine a multicomputer to distinguish it from a multiprocessor, in which processors share memory. Examples of multiprocessors are C.mmp [13] and Cm* [14]. Operating systems for multiprocessors have been developed [15,16,17]; we will discuss the relationship between Arachne and related research projects later.

The essential features of Arachne are:

1. The network appears to the casual user to be a single powerful machine. A process runs on one machine, but communicating processes have no need to know if they are on the same processor and no way of finding out. However, no attempt is made to transform an individual sequential program automatically into multiple processes.

---

[3]Arachne was formerly called Roscoe. The name (but not the substance) was changed because Roscoe is a registered trademark for an unrelated commercial software product.

- All processors are identical. Similarly, all processors run the same operating system kernel. However, they may differ in the peripheral units connected to them.
- No memory is shared between processors. All communication is achieved by messages between a processor and a small number of physically connected neighbors.
- No assumptions are made about the topology of interconnection except that the network is connected (that is, there is a path between each pair of processors). The communications hardware is assumed to be fast enough to allow concurrent processes to cooperate in performing tasks.
- Control is not centralized, either physically or logically. Resource allocation decisions result from negotiations among processes throughout the network. Each process involved maintains only local information.
- The network is constructed entirely from hardware components commercially available at the time of construction (January, 1978). The software is written in the C programming language [18] and compiled using an existing compiler.
- Everything described in this paper is functional. Although Arachne has undergone much revision, it has been working since the summer of 1978.

The remainder of this paper is organized as follows: Section 2 contains an overview of the structure of Arachne. In this section, we have resisted the temptation to describe what we intended to do or what we would have done if we had better hardware or more time or people to write better software; we attempt to describe Arachne as it actually is. Section 3 discusses some of the lessons we learned from the implementation effort. Section 4 compares Arachne to related multicomputer operating system projects. Section 5 describes current and planned activities in the Arachne project.

2. OVERVIEW

The goals and implementation of Arachne have been described in great detail elsewhere [1,19,20,21]. In this section we will summarize the most important elements of the Arachne architecture. The current Arachne implementation runs on five Digital Equipment Corporation LSI-11 computers.[4] Each has 28K words of memory, a programmable clock, extended instruction set, a serial asynchronous line (intended for a terminal), and 16-bit parallel synchronous lines (DRV-11) to one or more other LSI-11 machines. In addition, each LSI-11 has a parallel synchronous line to a PDP-11/40 running the Unix operating system [22]. All Arachne software is written in the C programming language [18] with the exception of a small amount of code written in assembler

---

[4]This equipment was purchased with funds from National Science Foundation Research Grant #MCS77-08968.

language. When Arachne is running, the PDP-11/40 is not part of the network; its function is to assist in program development, initial loading, and as a peripheral that provides file-system functions.

2.1 The Kernel

The structure of Arachne revolves about the kernel, which is a module that resides on all the machines of the network. Its primary purpose is to provide the process abstraction. In particular, it governs creation and destruction of processes, allocates their memory, schedules them, and transmits messages among them. Its services are requested by means of service calls, which appear to the caller as procedure invocations. The service calls that create and destroy processes are only issued by the resource manager utility process described below.

The fundamental concepts managed by the kernel are core images, processes, links, and messages. The first two components are roughly equivalent to similar concepts in other operating systems; links and messages are idiomatic to Arachne.

2.2 Links

The link concept is central to Arachne. It is inspired and heavily influenced by the concept of the same name in the Demos operating system for the Cray-1 computer [23]. Although the Cray-1 is a conventional uniprocessor, Baskett et al. suggest that links might also be used in a multiprocessor environment. Links have proved to be a great success in insulating the writer

of Arachne processes from the peculiarities of a multiprocessor architecture. Arachne has demonstrated the utility of the link concept for multicomputer architectures.

A link combines the concepts of a communications path and a capability [24]. It represents a one-way logical connection between two processes, and may be considered to be a non-revocable capability to an "input port" in Cashin's terminology [25]. Links should not be confused with lines, which are a physical connections between two processors. Each link connects two processes: the holder, which may send messages over the link, and the owner, which receives them. The holder may duplicate the link or give it to another process, subject to restrictions associated with the link itself. The owner of a link, on the other hand, never changes.

Links are created by their owners. When a link is created, the creator specifies a code and a channel. The kernel tags each incoming message with the code and channel of the link over which it was sent. Channels are used by a process to partition the links it owns into subsets; when a process wants to receive a message, it specifies an explicit set of channels. The process is blocked until a message arrives over a link corresponding to one of the specified channels. The code is used by the owner to associate incoming messages with the links that it has created. Links created for different clients are usually distinguished by the code.

A process names each link it holds by a small positive integer called a link number. This number is used by the kernel as

an index into the table of currently-held links for that process. All information about a link is stored in this table; no information about a link is stored in the tables of its owner.

The creator may also specify certain properties of the link, for example, that it may not be copied, that it may be used only once, or that its destruction causes a notification to be sent to the owner.

2.3 Messages

A message may be sent by the holder to the owner of a link. In addition, certain messages, called notifications, are manufactured by the kernel to inform the owner of a link of changes in its status. For example, the creator of a link may ask to be informed when the link is destroyed or copied. Notifications are identified to the recipient by an unforgable field.

A message may contain, in addition to arbitrary text, an enclosed link, which was previously held by the sender of the message. The kernel adds an entry to the link table of the destination process and gives its link number to the recipient of the message. In this way, the recipient becomes the holder of the enclosed link. If the sender does not destroy its copy, then both processes will hold identical copies of the link.

The holder of a link has no particular information about the owner of the link except indirectly through the manner the link was acquired and through the responses that the owner makes to messages sent along the link. In particular, the holder does not know whether the owner is on the same or a different machine.

Each kernel maintains a pool of message buffers allocated among all local processes. Incoming messages that have not yet been received by their destination processes are queued in these buffers, as are outgoing messages that have not yet been delivered to a neighboring machine. A simple priority algorithm is used to reduce the chance of buffer deadlock.

2.4 Utility Processes

Arachne has been designed so that as many as possible of the traditional operating system functions are provided not by the kernel, but by ordinary processes called utility processes. Utility processes reside on some machines, but not necessarily all. They provide such functions as terminal control, file services, and command interpretation.

The terminal driver is an example. One terminal driver resides on each processor that has a terminal. All terminal input/output by other processes is performed through messages to this process. It understands and responds to all commands accepted by a file (see below), as well as a few extra ones, such as "set modes" (for example echo/noecho, hard-copy/soft-copy).

A file manager process resides on each processor that has mass storage. Since only one floppy disk is directly accessible to Arachne, the file system is actually maintained by the PDP-11/40, which acts as a very intelligent disk controller from the point of view of the file manager process. Arachne can be configured with file managers on any number of machines; they share the same intelligent disk. A process wishing to open a

file sends a message to any file manager, which creates a link to represent the open file. The client process closes the file by destroying the link.

The most complex utility process is the resource manager (RM). Resource managers reside on all processors and are connected by a network of links. Any process may request its local RM to create a new process. The local RM may create the process on its own machine or relay the request to another RM, based on local considerations such as availability of free memory and the possibility that the required program is already in memory. The next RM decides in a similar way whether to load the process or relay the request.

A new process is started with only a link to its local RM. It can use this link to request links to the process that requested its creation, to a file manager process, to a terminal driver, or to other resources. The RM can kill the process, or it can give a special link to another process (usually a terminal driver) that may be used to kill it.

Processes form a hierarchy in two senses. First, every process is started by another. However, since the RM usually performs the starting, this hierarchy is quite flat. Second, every process is started at the request of another to the RM. The RM maintains this hierarchy in order to satisfy requests to terminate processes.

2.5 Library Routines

Functions provided by service calls are rather primitive, and communication with utility processes can involve complicated protocols. An extensive library of routines has been provided to simplify writing of programs that use service calls and utility processes. For example, general message passing is simplified by "call", which creates a reply link, sends the message, waits for the reply, and decodes the returned message.

Other routines may be used to correspond with the file manager to request directory information or to open, create, or delete a file. For file or terminal I/O, one routine separates strings of text into message-sized chunks, and another accepts text in chunks and assembles it into a buffer.

3. DESIGN CHOICES AND EXPERIENCES

The first version of the Arachne kernel was put into service in June 1978 on five Digital LSI-11 microcomputers. Since that time, additions and enhancements have been added, and many parts of Arachne have been modified or completely rewritten. Some of our initial decisions have turned out quite well; others have been revised. Some difficult problems that arose during implementation were temporarily resolved by ad hoc solutions, awaiting fuller study.

Arachne has reached a level of development sufficient to demonstrate its potential as a production operating system and

also to determine its weaknesses. For example, the following have been implemented: a command interpreter that accepts Unix-like pipe commands [22], the Unix text editor, and a sophisticated parallel checkers-playing program.

3.1 Choice of Hardware

Many of the features of Arachne were influenced by our desire to develop a design appropriate to a multicomputer with a large number of component processors (perhaps several thousand), even though the original Arachne implementation has only five processors. We chose to investigate multicomputers instead of multiprocessors and to pursue decentralized control. Shared memory requires complicated switches or multiport memories, whose cost and complexity would be prohibitive for such large networks. Centralized control would place an unacceptable load at the controlling site, create communications bottlenecks, and make the system much more vulnerable to hardware failures, which become more likely as the number of components increases.

Other decisions were based on constraints due to the limited resources (in time and money) available to the project. Lacking the facilities to build custom hardware, we were forced to use components that are less than ideally suited to the application. The decision to use entirely off-the-shelf components allowed us to get the system working quickly, but it meant we had to settle for hardware not ideally suited to our needs.

We have discovered that the most vexing aspect of the inexpensive processor we use is not its rather slow speed, but its

lack of memory management facilities. This deficiency manifests itself in three ways. First, we have no way to prevent a runaway process from damaging other processes or the kernel on its machine. Second, a core image, once loaded, cannot be moved within a processor or to another processor except at the same address. This restriction has prevented us from experimenting with process migration. Third, the address space of 56K bytes must be shared by the kernel, utility processes and applications programs. A processor equipped with a kernel, resource manager, terminal driver, command interpreter, and file manager has almost no room for application programs. Fortunately, the Arachne design mitigates this problem; user programs running on other, more lightly equipped processors may use the services of a heavily equipped "service-only" processor.

In attempting to live with our space problems, we have investigated why programs cannot be made smaller. One reason is that the general (and primitive) send/receive paradigm for communication is significantly more complicated than the higher-level call/return. Library routines such as "call" hide this difficulty from the programmer, but separate copies must be loaded with each process, thus wasting space.

3.2 Choice of Programming Language

We decided to use the C programming language rather than design a language specifically suited to our application, since we felt that writing a new compiler would unacceptably delay the implementation of the operating system. Obviously, C is far superior to assembler language. On the other hand, a great deal of time was spent tracking down bugs that would have been caught at compile time in a more strongly typed language. An errant subscript or pointer reference in an applications program can subtly modify a kernel table, producing symptoms seemingly unrelated to the source of the error, but C provides no run-time assistance in checking for such errors. In retrospect, we would much rather have written the program in Pascal [26] or Modula [27], had an appropriate compiler been available. None of these languages provides explicit support for messages. A language that more closely reflects the communication primitives of the underlying operating system would be useful.

3.3 Experience with Links

The most gratifying result was the conceptual simplification due to the link concept. Most obviously, it isolates the programmer from details of communication. Sending a message to another process is the same whether or not the destination is on the same machine, even if the message needs to be relaid by an intermediate processor.

The fact that a process sending a message names a link, not a destination process, has several advantages. First, it allows flexibility in allocating resources. A client requesting a service may be given a link to any process that provides the service. The client has no way of knowing to which server it has been connected, only that messages sent over the new link are understood and answered properly. Second, like abstract data

types [28], links help to separate the behavior of a facility from its implementation. An example is provided by the file manager. A request to open a file is satisfied by returning a link that may be used for operations on a file, such as reading or writing. In the current implementation, this link is owned by the file manager, which honors such requests itself. However, we could switch to an implementation in which each open file is represented by a separate process, and the change would be invisible to client processes.

A related advantage to hiding the identity of the owner of a link from the holder is the ability to substitute a process that fulfills the expectations of the holder but does more. One possibility mentioned by Baskett et al. [23] is to replace a standard process by one that not only satisfies requests but also monitors them to gather performance statistics. Another version of this idea is used in Arachne to implement the Unix pipe facility. A "pipe" process is interposed between two processes. The pipe responds to one process as if it were an output file and to the other process as if it were an input file, buffering data between them. Similarly, a terminal driver is capable of behaving like a file, so a process needs no modification to read or write to a terminal rather than a file.

The link concept has also proved useful to solve other problems not so obviously related to communication. One example is synchronization. Since processes do not share address space, synchronization is not necessary to prevent interfering memory references. However, in situations where synchronization is required, the ability to accept messages only on a subset of all incoming links can be used to advantage. For example, the pipe process has two incoming links, one from the writing process and one from the reading process. When the pipe's buffer is empty, it stops accepting read requests; when it is full, it stops accepting write requests. Otherwise, it accepts whichever request comes first.

On the other hand, our experiences with writing application programs using links leads us to believe that for typical applications they are at too low a level. Research in high-level programming language constructs for distributed programming has begun to yield some results [25,29,30,31,32,33]. As these issues become better understood, they will suggest more appropriate low-level implementation primitives than links.

The shortcomings of links can be seen through a simple example. A common situation involves a client process that wants to send a single request to a server process and expects to receive a single message in reply. It must manufacture a use-once link to carry the reply, send the request with the reply link enclosed, wait for the reply for a specified maximum time, and institute recovery actions if the reply does not come back soon enough. We provide a library routine named "call" to perform most of these actions, and another named "recall" in case the desired recovery from timeout is to wait a little longer. However, this situation arises so frequently that it might be worthwhile to provide a "send-with-reply" primitive (sometimes called "remote invocation send" [31,29]) directly in the kernel.

Not only would building this facility into the kernel save space (since it would not need to be duplicated in the code of each process), it might allow the kernel to manage flow control and buffer allocation more intelligently, since it would supply the kernel with more information on the expected behavior of processes. We do not provide a facility comparable to "call" to simplify the writing of the server process. Perhaps a message-screening facility similar to the select/accept construct of Ada [32] would be useful for such purposes.

When a reply requires several messages, the situation is even more cumbersome. For example, the server might represent a file and the request might be to read a large amount of data. Currently, the client must break its request into several packets and create a reply link for each one. Duplex links, discussed in the next section, are a reasonable alternative.

3.4 Duplex links

Arachne links are strictly simplex. No information at all is stored in the kernel of the owner. The effect of this restriction is that holders cannot be found. Therefore, a link cannot be revoked (although messages may be ignored, or read and discarded), a sender cannot be informed of failure to deliver messages (although it may depend on higher-level acknowledgements from its correspondent), and a kernel moving a process cannot inform all potential senders that the old address is invalid.

One way to introduce a duplex ability is to make all process names known globally. This approach is followed in Plits [34],

in which processes are statically determined at compile-time. It does not permit the information-hiding facilities of links.

Another approach to duplex links is the concept of a "connection", as used in computer communications networks [35]. Once a connection is established between two processes, each may send and receive. Establishing the connection requires several low-level messages, usually performing the "three-way handshake". We believe that extensions to the current meanings of links can build semi-permanent duplex links that preserve the capability flavor of our current link mechanism.

3.5 Error Handling

Another problem that complicates writing programs and leads to many bugs is the continual need to check for error conditions. For example, to read data from a file, a process must ask the resource manager for a link to a file server, ask the file server for a link to the file, and ask the file for the desired data. Any one of these requests may fail (no file server is available; the file does not exist; an I/O error occurs), so the prudent client must continually check return codes. Normally, a process does not expect any of these errors and does not know how to cope with them if they occur, but would rather have the system provide some default action (usually killing the process).

To ease these problems, we added a facility for special error messages. When a process creates a link (to itself), it specifies whether the link may carry error messages. Rather than sending an ordinary message over such a link, a holder may cause

an error message to be sent. The owner (receiving end) of a link may associate with it a handler procedure that is automatically invoked whenever an error message is received. If no handler has been established, reception of an error message kills the receiving process. When a process is killed, all links it holds are destroyed. Any of these links may also be error links, in which case the kernel generates an error message announcing the demise of the process. When this message reaches its destination, it will kill that process unless it is explicitly handled. In this way, errors propagate backwards until some process is willing and able to handle them intelligently.

3.6 Interrupts

As mentioned above, one design principle we followed was to put as little "operating system" function as possible in the kernel. As an example, device drivers are not kernel programs, but ordinary processes. A service call is provided that establishes a process as the handler for hardware interrupts on a particular device. The arguments to the call are the interrupt vector address for the device, the address of a "handler" procedure in the address space of the process that should be invoked on interrupt, and a channel number for synchronizing the main process with the handler procedure.

To wait for an interrupt to occur, the process waits for a "message" to arrive on the synchronization channel. When the hardware interrupt occurs, the kernel invokes the designated handler in a special "interrupt-handler" state. In this state,

the procedure is not running as a true process, and cannot, for example, send or receive messages. However, since it runs in the same address space as the rest of the device-driver process, it can inspect and modify global variables in that address space. It can also invoke a special "awaken" service call that enqueues a synchronization message to the main process on the channel named in the "handler" call. When the handler procedure returns, whatever process was running when the hardware interrupt occurred is allowed to continue. However, if the handler invoked "awaken", the main process is no longer blocked waiting to receive from the synchronization link.

This mechanism is illustrated by the method the terminal driver uses to respond to a request for a line of input. It executes a "receive" service call on its synchronization link. Each time a character is typed at the terminal, the handler procedure is invoked. It echoes the character and places it into a buffer. If the character is a line terminator, the handler also executes an "awaken" service call. This call unblocks the terminal driver process, which responds to the synchronizing message by sending the line of input to the client. From the point of view of the terminal driver process, it appears that another process has filled the line buffer and sent a message to indicate that it is full.

3.7 Asynchronous Receipt

A related issue concerns software interrupts. In the original version of Arachne, a process could only receive a message by executing a blocking "receive" service call. The process is blocked until a message arrives on one of the designated links or the indicated time limit is exceeded. (A process can find out whether any messages are queued by specifying a time limit of zero; it can wait indefinitely by specifying an infinite time limit.)

However, we encountered an application in which it is necessary for a process to be able to receive a message "spontaneously". A checkers-playing program was coded for Arachne that uses multiple processes to search a lookahead tree of possible future game states [36]. If a process searching one branch of the tree finds a particularly advantageous move, it would like to inform those searching other branches, so that they need not consider ramifications of moves that are known to be worse. However, the other processes have no idea when (if ever) such a discovery might occur, so they must not block waiting for the good news. An alternative is to try to receive news periodically with a zero time limit, but this technique introduces the undue overhead of repeated fruitless calls. If Arachne allowed shared variables between processes, the value of the best move found so far could be put in a global location. However, lacking global memory, we arrived at a different solution, analogous to our handling of interrupts.

A process may associate a message-handler procedure with a set of incoming channels. If a message arrives on any of those channels while the process is running, it is interrupted and the handler procedure is immediately invoked. (If the process is not currently running, it is marked so that the handler will be invoked the next time the process is scheduled.) A message handler procedure is restricted in much the same way as an interrupt handler; it may not use most service calls. Since the handler shares the address space of the process proper, it can extract information from the message and place it in a global location accessible to the main process. The main process perceives a spontaneous change to the value in that location.

3.8 Process Control

An interactive operating system needs a way to stop a process that is running out of control. Once again links were pressed into service to solve the problem. When a process requests the kernel to create another process, it receives a special link, called a "lifeline", to the new child process. The "kill" service call may be executed by the holder to kill the child. Normally, only the resource manager uses this service call, but it does so in response to a request from another process, such as a command interpreter.

Let us consider what happens when a user at a terminal types a command to invoke a program. The command may specify that the program be executed either in the foreground or background. For a background request, the command interpreter displays a number

by which the user may identify the new process. At some later time, the user may type the command "kill nnn", where nnn is that number.

Here is what happens internally: The command interpreter sends a message to the local resource manager requesting creation of a background process. The resource manager creates the process and stores its lifeline in a table, replying to the command interpreter with the index of the table entry. The command interpreter displays this index to the user. When the user types a kill command, the command interpreter sends the argument of the command to the resource manager, which uses it to index into its table, find the lifeline, and kill the process.

Now let us consider foreground processes. Each terminal is always associated with a unique foreground process, typically a command interpreter. If the user requests another program to be started in the foreground, the current foreground process is conceptually pushed on a stack, and further terminal input is sent to the new foreground process. At any time, the user can kill the foreground process by typing an interrupt character. The previous foreground process is then reestablished.

As with background processes, the user's command to start a foreground process is accomplished by the command interpreter sending a request to the resource manager. This time, however, the resource manager saves one copy of the lifeline to the new process on a stack and gives another copy to the controlling terminal driver process. To provide prompt response to an interrupt character, the terminal driver uses its copy of the lifeline to

kill the process directly, without waiting for a message to propagate through several intermediaries. When a new foreground process is created, the resource manager sends a copy of the new process' lifeline to the terminal driver, telling it to discard the lifeline it currently holds. When the new process terminates, the resource manager sends the terminal driver a copy of the lifeline of the next process down the stack.

This scenario is complicated by the fact that the stack of foreground processes associated with a given terminal may be scattered through several physical processors. When a resource manager receives a request to start a new process, it may find that its own processor is too crowded and relay the request (together with the reply link) to a resource manager on another machine. Each resource manager only maintains lifelines for processes on its own machine, but if the predecessor of a foreground process is not on the same machine, the resource manager stores in its lifeline stack a link to the resource manager responsible for the predecessor.

3.9 Kernel knowledge of processes

One of our goals was to keep the Arachne kernel ignorant of the function of individual processes, so that it treats processes uniformly. However, in some circumstances, we were forced to relax this restriction. For example, there is a distinguished process on each machine called the "kernel job", which is used during initialization to load the first resource manager and later to ask another machine to create a new process. The kernel job

is not only able to perform service calls as an ordinary process, but is also able to manipulate kernel data structures directly. It takes advantage of this ability to construct links to the kernel job and to the resource manager of other machines in the network. No other process may invent a process name and build links based on that name.

Another example concerns loading core images. On one hand, the loader must act as an ordinary process in order to communicate with the file manager (by links and messages) to fetch the program text. On the other hand, the loader is invoked by the kernel to satisfy a service call, so the kernel needs to know its special function. In addition, the loader needs access to memory outside its own space in which to load the new process. Our solution was to include the loader module as a part of the kernel. It is free of direct calls on the rest of the kernel, using standard user service calls throughout (including a privileged reserved service call to request free space). However, the loader is not scheduled like an ordinary process, but is rather invoked from the kernel by subroutine call.

A third example concerns "privileged" service calls. Only the resource manager and kernel job should use process-creation calls. Other processes should request the resource manager's assistance to start new processes. Only the loader module of the kernel should request free storage. These "restrictions" are currently enforced only by convention. Proper enforcement would require that the kernel treat some utility processes in a special manner.

3.10 Inter-machine communication

For the sake of efficiency, communication between machines is governed by code in the kernel. An alternative is for the kernel to give the message to a special "network manager" process that controls the hardware line directly.

The advantage of separating this function from the kernel is that various strategies may be used without redesigning the kernel. In particular, the operating system would be easy to modify as different hardware is installed to connect machines or as different topologies and routing methods are used. On the other hand, this separation would have the disadvantage that the kernel would have to treat the network manager process in a very special way. The network manager process would be unique in that each machine must have exactly one, and the kernel must know which process it is. That process not only receives input from the physical line (as the teletype driver does); it also gets special messages from the kernel. These messages are different from typical messages received by normal processes in that they they have fields that are hidden from normal recipients (like the location of the recipient) but must be inspected to perform routing decisions and passed along intact to the corresponding network manager process on another machine. The network manager process would need a special form of send primitive to transform the message back into the ordinary format.

Another disadvantage of a separate network manager process is that it would introduce extra levels of buffering in remote

messages, leading to considerable expense. Sending messages is already fairly slow (about 10 milliseconds for a 40-byte message between machines).

3.11 Routing

Although most of Arachne is independent of the actual topology, one module in the kernel must know the best first step toward any given machine. Each process id contains the name of its host machine. This arrangement prevents migration, since migration would change the binding of process to host. We could change the id at the time we move the process, and have the old host keep track of processes that have migrated away. It could not only redirect errant mail to the moved process, it could also send a change-of-address form to the sender of errant mail. The table entry would have to be retained until either the process terminates or all holders of links to it have been informed.

3.12 Reliability

Multicomputers have both the potential to increase reliability and the necessity to recover from failures. The potential comes from the duplication of processors and routes between them. The necessity comes from the multiplicity of components, each of which has some failure rate.

Despite the importance of reliability, we decided that reliability would not be a major component of our research. We did feel that it was important to bring a node back into the network after failure, since otherwise we would have to start the entire

operating system over. Relinking a node into the web of resource managers is the hardest part of the task; it requires the kernel job to be able to send a message to a functioning resource manager on another machine in order to start the recovery. A new copy of the local resource manager is loaded directly from the PDP-11/40 (without using the file manager at all) and given a link to a foreign resource manager. Then the new resource manager can use foreign resources to bring back working copies of any other utility process desired. However, processes that were on the machine when it went down are lost, and any other processes trying to communicate with them become confused.

The initial versions of Arachne employed positive acknowledgement on the lines; the current version does not. We found the physical lines to be quite reliable, but slow, so we aimed at speed instead of reliability. Nonetheless, entire messages do occasionally get lost, leading to unrecoverable situations. As we acquire DMA devices, we intend to invest in much more robust communication protocols.

4. COMPARISON WITH OTHER OPERATING SYSTEMS

The past few years have seen several research projects developing distributed operating systems. This section briefly discusses those that we feel are closest to Arachne in goals, implementation, or spirit.

4.1 Hydra, Medusa, and StarOS

Hydra [15], Medusa [17], and StarOS [16] are all object-based operating systems for multicomputers. Object-oriented approaches treat all entities accessible by a process in a uniform way. Among the entities so treated are local data, shared data, and other processes.

Hydra was designed for the C.mmp multiprocessor [13], and the others were designed for Cm* [14]. Like Arachne, these systems all use message-passing for interprocess communication, but unlike Arachne, they all take advantage, to some extent, of the ability of processes to share data. Both StarOS and Medusa use the concept of a task force, which is a set of processes that together provide some service. In StarOS, members of task forces are often small, share little code, and rely on each other to perform subroutine-like actions. StarOs uses mailboxes as destinations for messages. A process must possess the capability for a mailbox in order to send messages to it. These capabilities are passed in messages much like Arachne links may be enclosed in messages. However, the implementation language also allows the implementor of a task force to define static mailboxes, which are then known throughout the task force.

Medusa task forces usually share data but not code. Each process in a task force has access to private objects as well as public objects shared by the entire task force. Objects are of several predefined types, including pages, semaphores, and pipes. Clients automatically possess capabilities to task forces that

provide system utilities. Members of the task force select individual processors that they are willing to service. All processes on those processors then have the capability that points to the member of the task force. As the task force grows or shrinks, the selections are updated.

4.2 Distributed Computing System

The Distributed Computing System [5] was a project that connected a small number of inhomogeneous processors together. Some applications have to live on one processor, while others are flexible. As a new program is created, a bidding scheme determines which machine it would run best on, based on its needs and the current loading of the various machines. A ring is used to connect the processors together, so broadcast is a natural form of communication. It is possible to send a message to a process without knowing where it is, since its interface to the ring will pick up any messages destined for it.

Arachne is based on point-to-point communication. We allow neither broadcast nor "generic invocation", both of which are easier on a ring. On the other hand, we do not need to deal with contention for the physical lines.

4.3 Distributed Computer Network

The Distributed Computer Network [8] connects about 6 different PDP-11 machines by various-speed point-to-point interconnection hardware. The operating system loaded a process wherever the user directed, and then arranged for communication between it

and other processes. Arachne is more automatic in its placement of processes. DCN also took mapped files into user space when possible; Arachne uses message passing as a way to implement file access.

4.4 SODS/OS

The SODS operating system [37] is very similar to Arachne. The following similarities are particularly striking:

1. The goal is to produce a decentralized operating system with distributed control.
2. Processes run without knowledge of their location.
3. 95% of the code written in C.
4. Each process is associated with a capability table that is not in user space.
5. Exception and interrupt-class messages are defined.
6. There are use-once "response" capabilities.
7. Most operating system functions are provided by utility processes.

Despite these similarities and the fact that SODS is more recent than Arachne, communication with the SODS implementors has shown that the efforts are independent and represent convergent evolution of ideas.

Arachne and SODS differ in some important ways:

1. Messages in SODS are directed to and received from exchanges, which are "free ports" in Cashin's terminology [25] and exist independently of processes. In this way, a given message may be received by any process with the necessary capability over its exchange. An exchange is released when no process retains a capability to it. Each exchange may have multiple receivers as well as multiple senders, but full-duplex communication between a pair of processes still requires a separate exchange for each direction.
2. SODS provides control messages for process control such as killing, or suspending a process. These messages are a generalization of the Arachne lifelines and the "kill" service call.
3. The "send" service call can specify timeout.
4. SODS employs utility processes for some functions that Arachne performs in the kernel, including the exchange manager, user memory manager, agent (which handles service calls), and network manager.
5. Global information is kept about all processes and all exchanges.

SODS only runs on a two-machine network (IBM Series/1), although, like Arachne, it was designed to control much larger collections of processors. A successor, using more machines and a communications ring, is planned.

One interesting difference between Arachne and SODS concerns the decision of how many facilities are provided to a process by default and how many must be explicity requested.

When a message arrives that contains a new link (typically a return link), Arachne installs that link in the recipient's link table as a side effect of receiving the message. SODS, on the

other hand, informs the recipient that a link arrived, but requires that the recipient ask for the link explicitly before it is installed.

On the other hand, when a child process is created, Arachne initializes it holding only one link, which was presented to the kernel by the parent (typically, the resource manager). The child uses the link to ask for other resources it may need (such as standard input, standard output, a link to its real parent (considering the resource manager as a foster parent), and a link to a file manager). SODS, on the other hand, initializes each child with a "starter package" that contains all the communications tools ordinary processes are expected to need. These are all presented to the kernel by the parent at the time the parent asks the kernel to create the child. The Demos operating system follows the SODS approach.

4.5 Technec

The purpose of Technec [6] is to investigate applications such as distributed simulation of distributed processes, and distributed compilation. The entire network (about 8 LSI-11's) is intended to be used for one purpose at a time. Each node executes only one process. Resource allocation is performed statically. A central node maintains globally known names to allow interactive control. For these reasons, although Technec is superficially similar to Arachne, it is quite different both in purpose and implementation.

4.6 Micros

The Micros/Micronet project [11,38] was started at the same time as Arachne, has many of the same goals, and uses the same kind of processors. Nonetheless, Arachne and Micros differ in their approach. Whereas Arachne was careful to use only "off-the-shelf" hardware and programming languages, Micros began by designing special-purpose communications interfaces and adapting Concurrent Pascal [39]. Both of these decisions caused considerable delay in actually implementing an operating system. The use of Concurrent Pascal gives Micros a much more static flavor, since the number of processes and their relationship to each other must be specified at compile-time.

4.7 MuNet

MuNet [40] is a multicomputer designed to investigate parallel implementations of programs built from actors. MuNet is quite like Arachne in that it is intended to provide a general-purpose computing facility. The intented benefits are primarily economic: MuNet, like Arachne, is an attempt to make use of many small, cheap microprocessors to provide the functions of one large, expensive uniprocessor. The problems of interest to the researchers are techniques for structuring message-based distributed operating systems; in particular, the creation, maintenance, distribution, and destruction of inter-object capabilities ("references" in MuNet; "links" in Arachne). MuNet seems to be strongly influenced by the "Actor" model of processes pioneered

by Carl Hewitt [41].

Like Arachne, MuNet attempts to distribute the functions of an operating system into a set of loosely-coupled processes that respond to requests for service. Both processes and messages may contain references to other processes.

MuNet differs from Arachne in two interesting ways. First, MuNet intends that any object (that is, any process) may be stored on backing store along with its non-local references. When the object is recovered from backing store, it may proceed. All data are localized into objects, and all operations on data are messages that may be sent to objects, so many objects (for example, name servers) are inactive for long periods of time and may as well be swapped out. A similar approach is taken in the XOS operating system for the X-Tree project [42]. We have not yet considered this possibility in Arachne. In fact, because the LSI-11 provides no memory-management support, Arachne performs no swapping.

Second, the route to any process A from any process B that has a reference to A is determined by a reference tree that joins A to B. This technique avoids general-purpose routing based on processor names. One cost of the technique is that a complicated protocol is required to enforce the reference tree as processes gain and lose references (which happens quite frequently in Arachne). This "membership protocol" requires 7 different message types and 13 states. Another cost is that shorter routes might be available that reference-tree routing will not find.

The MuNet approach has some strong advantages. It decouples the routing question from the naming issue, and thereby makes migration fairly easy. On the other hand, it is very sensitive to node failures that disconnect spanning trees, and it can easily create routes that are significantly longer than a direct path. In addition, it requires significant overhead the first time a host learns about a new owner. The host's table space for an owner may not be reclaimed when all local references to that owner have disappeared, since by then it may need to relay messages from other hosts.

4.8 Champ

The Champ operating system [9], is designed for two goals: reliability and expandability. It is built of nodes connected by point-to-point hardware and insensitive to the interconnection topology. For the sake of reliability, each node is composed of not only a task processor, but also a supervisor processor whose job is to perform diagnostic tests that will detect when the task processor fails. For the sake of efficiency, communications are directed through an I/O processor at each node.

CHAMP seems to be designed with military decision support applications in mind. Multiple processors are viewed as a way to provide reliability directly (one failing processor will not bring down the entire system) and as a way to scale the hardware configuration to a problem to provide excess capacity that can be "burned" to enhance reliability.

The architecture of CHAMP is strongly influenced by the "M-module" concept, which, in turn, seems to have its roots in goal-directed problem-solving languages such as Planner [43]. M-Modules are not general processes, but are rather tailored to particular sorts of applications. These applications have the property that they tend to increase the local fund of knowledge across time; resumption from a checkpoint suffers only from a small loss of time. It is much harder to resume, say, a file server that fails while client processes are still active.

4.9 Thoth

Thoth [3] is intended to be a portable microcomputer operating system. It is not explicitly meant for a multicomputer, but since the principal communication between processes is by messages, a distributed form of Thoth is consistent with its design. Thoth tries to make individual process creation and destruction very efficient, so that algorithms may make profligate use of processes. For example, a text editor may be designed in two processes, the command interpreter and the file manager. If the former receives an illegal or ill-formed command, it will terminate and the latter will start a fresh interpreter.

Thoth differs from Arachne in the semantics of message passing. Send blocks the user process until the message has been received. The recipient may either respond with another send (in which case it is blocked in turn) or by a "reply", which does not block. The receipt of messages may be restricted to those coming from a particular process. We have found that selectivity is

quite important, but provide a more general mechanism in two respects: First, Arachne messages are received by link (through the channel number mechanism), not by sender, so if the holder of a link should distribute it, all messages on the link are received by the same selection. The recipient need not (in fact, can not) have any information about the identity of the sender. Second, a process has more flexibility in screening message, since it may specify any subset of the set of channels of incomming links.

5. CURRENT AND FUTURE WORK

Arachne has reached a stable state and is no longer under active development. It has demonstrated that multicomputer operating systems can be built out of processes that communicate with each other and taught us much about the detailed form that communication should have. It has also shown that a capability-like naming construct provides exactly the right sort of information hiding and dynamic flexibility.

We have recently replaced the LSI-11 processors with eight PDP-11/23 processors, which are software-compatible with the LSI-11 but have memory-management hardware. We have also acquired interfaces that allow communication at one megabit/second on a shared coaxial cable. At the time of this writing (May 1981), we are modifying the Arachne kernel to take advantage of the new hardware. We plan to experiment with process migration. Simulations have shown [44] that process migration can be expect-

ed to increase the throughput of a multicomputer significantly.

Simultaneously, we are designing a successor to Arachne, tentatively called Charlotte. A pilot implementation will use the eight PDP11-23's, but we are planning to construct a multicomputer with between 50 and 100 nodes, each with a significant amount of memory and computing power, interconnected by a cable using multichannel FM broadcast. We plan to build a full production operating system for this multicomputer. In particular, we expect that text editors, compilers, and a general-purpose file system will be available on the multicomputer. Part of this research will involve developing a programming language in which to represent distributed algorithms; the design of the language and the operating system will depend on each other.

The experience gained from Arachne will form the foundation for design of this new operating system. The new system will have significant differences from Arachne because of lessons we learned from Arachne and because of the different communications facilities. In particular, all pairs of nodes will be, in effect, directly connected. Broadcast and "generic invocation" techniques will be explored. We expect that a more general notion of capability will replace the links of Arachne, and that connections between processes will be more symmetrical.

6. ACKNOWLEDGMENTS

The authors are pleased to acknowledge the assistance of Prof. Sun Zhongxiu of Nanking University and the following graduate students who have been involved in the Arachne project: Jonathan Dreyer, Jack Fishburn, Michael Horowitz, Will Leland, Paul Pierce, and Ronald Tischler. Their hard work has helped Arachne become a reality.

7. REFERENCES

[1] M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," Proc. 7th Symposium on Operating Systems Principles, pp. 108-114 (December 1979).

[2] J. E. Ball, J. A. Feldman, J. R. Low, R. F. Rashid, and P. D. Rovner, "RIG, Rochester's intelligent gateway: system overview," IEEE Trans. Software Eng. 2, 4, pp. 321-328 (December 1976).

[3] 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).

[4] W. R. Cyre et al., "WISPAC: A parallel array computer for large-scale system simulation," Simulation, pp. 165-172 (November 1977).

[5] D. J. Farber, J. Feldman, F. R. Heinrich, M. D. Hopwood, K. C. Larson, D. C. Loomis, and L. A. Rowee, "The Distributed Computing System," Proc. 7th Annual IEEE Computer Society International Conference, pp. 31-34 (February 1973).

[6] W. Huen, P. Greene, R. Hochsprung, and D. El-Dessouki, "TECHNEC, a network computer for distributed task control," Proc. 1st Rocky Mountain Symposium on Microcomputers, (August 1977).

[7] J. Livesey and E. Manning, "Protection in a transaction processing system," Proceedings of the 7th Texas Conference on Computer Systems, (1978).

[8] D. Mills, "An overview of the Distributed Computer Network," Proc. National Computer Conference 45, AFIPS Press, pp. 523-531 (1976).

[9] C. A. Monson, P. R. Monson, and M. C. Pease, "A cooperative highly-available multi-processor architecture: CHAMP," Proceedings of Compcon Spring 1979, pp. 349-356 (February 1979).

[10] S. A. Ward, "The MuNet: A multiprocessor message-passing system architecture," Proceedings of the 7th Texas Conference on Computer Systems, (1978).

[11] L. D. Wittie and A. M. van Tilborg, "MICROS, a distributed operating system for MICRONET, a reconfigurable network computer," IEEE Transactions on Computers C-29, 12, pp. 1133-44 (December 1980).

[12] Y. Takahashi, N. Wakabayashi, and Y. Nobutomo, "A binary tree multiprocessor: CORAL," Journal of Information Processing 3, 4, pp. 230-237 (February 1981).

[13] W. A. Wulf and C. G. Bell, "C.mmp -- a multi-mini-processor," Proc. AFIPS 1972 Fall Joint Computer Conference 41, Part II, pp. 765-777 (1972).

[14] A. K. Jones, R. J. Jr. Chansler, I. Durham, P. Feiler, and K. Schwans, "Software management of Cm* -- a distributed multiprocessor," Proc. National Computer Conference 46, AFIPS Press, pp. 657-663 (1977).

[15] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The kernel of a multiprocessor operating system," CACM 17, 6, pp. 337-345 (June 1974).

[16] A. K. Jones, R. J. Jr. Chansler, I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, a multiprocessor operating system for the support of task forces," Proc. 7th Symposium on Operating Systems Principles, pp. 117-127 (December 1979).

[17] J. K. Ousterhout, D. A. Scelza, and S. S. Pradeep, "Medusa: An experiment in distributed operating structure," CACM 23, 2, pp. 92-105 (February 1980).

[18] B. W. Kernighan and D. M. Ritchie, The C Programming Language, Prentice-Hall (1978).

[19] R. A. Finkel, M. H. Solomon, and R. Tischler, "Arachne User Guide, Version 1.2," Technical Summary Report 2066, University of Wisconsin Mathematics Research Center (April 1980).

[20] R. A. Finkel and M. H. Solomon, "The Arachne Kernel, Version 1.2," Technical Report 380, University of Wisconsin--Madison Computer Sciences (April 1980).

[21] R. A. Finkel, M. H. Solomon, and R. L. Tischler, "Roscoe Utility Processes," Technical Report 338, University of Wisconsin--Madison Computer Sciences (February 1979).

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

[23] F. Baskett, J. H. Howard, and J. T. Montague, "Task communication in Demos," Proc. 6th Symposium on Operating Systems Principles, pp. 23-31 (November 1977).

[24] R. S. Fabry, "Capability-based addressing," CACM 17, 7, pp. 403-412 (July 1974).

[25] P. M. Cashin, "Inter-process communication," Technical Report 8005014, Bell-Northern Research (June 1980).

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

[27] N. Wirth, "Modula: A language for modular multiprogramming," Software Practice and Experience 7, 1, pp. 3-35 (1977).

[28] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, New York (1972).

[29] P. Brinch Hansen, "Distributed processes: A concurrent programming concept," CACM 21, 11, pp. 934-941 (November 1978).

[30] C. A. R. Hoare, "Communicating sequential processes," CACM 21, 8, pp. 666-677 (August 1978).

[31] B. Liskov, "Primitives for distributed computing," 7th Symposium on Operating Systems Principles, pp. 33-42 (December 1979).

[32] J. D. Ichbiah et al., "Preliminary Ada reference manual," Sigplan Notices 14, 6, (June 1979).

[33] C. Mohan, "A perspective of distributed computing models, languages, issues, and applications," Working paper DSG-8001, Distributed Systems Group, Department of Computer Sciences, Austin, TX (February 1980).

[34] J. A. Feldman, "High level programming for distributed computing," CACM 22, 6, pp. 353-368 (June 1979).

[35] A. S. Tanenbaum, Computer Networks, Prentice-Hall (1981).

[36] J. P. Fishburn, R. A. Finkel, and S. A. Lawless, "Parallel alpha-beta search on Arachne," Proc. 1980 International Conference on Parallel Processing, pp. 235-243 (August 1980).

[37] W. D. Sincoskie and D. J. Farber, "SODS/OS: A distributed operating system for the IBM Series/1," Operating Systems Review 14, 3, pp. 46-54 (July 1980).

[38] L. D. Wittie, "A distributed operating system for a reconfigurable network computer," Proc. 1st International Conference on Distributed Computers, pp. 669-678 (October 1979).

[39] P. Brinch Hansen, The Architecture of Concurrent Programs, Prentice-Hall (1977).

[40] R. H. Halstead, Jr., "Object management on Distributed Systems," Proceedings of the 7th Texas Conference on Computer Systems, (1978).

[41] C. Hewitt, "A universal modular ACTOR formalism," Proc. 3rd International Conference on Artificial Intelligence, (1973).

[42] B. Miller and D. Presotto, "XOS: An operating system for the X-Tree architecture," Operating Systems Review 15, 2, pp. 21-32 (April 1981).

[43] D. G. Bobrow and B. Raphael, "New programming languages for artificial intelligence research," Computing Surveys 6, 3, pp. 153-174 (September 1974).

[44] R. M. Bryant and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," Proc. Second International Conference on Distributed Computing Systems, pp. 314-323 (April 1981).