# The CRYSTAL Multicomputer: Design and Implementation Experience

by

David J. DeWitt
Raphael Finkel
Marvin Solomon

Computer Sciences Technical Report #553

September 1984

# The CRYSTAL Multicomputer: Design and Implementation Experience

David J. DeWitt
Raphael Finkel
Marvin Solomon

Computer Sciences Department
University of Wisconsin - Madison

This research was partially supported by the National Science Foundation under grant MCS81-05904, by the Defense Advanced Research Projects Agency (DoD) ARPA Order No. 4095 Monitored by Naval Research Laboratory Under Contract No. N00014-82-C-2087, and by a Digital Equipment Corporation External Research Grant.

## ABSTRACT

This paper presents an overview of the hardware and software components of the Crystal multicomputer project. The goal of the Crystal project is to design and implement a vehicle that serves a variety of research projects involving distributed computation. Crystal can be used simultaneously by multiple research projects by partitioning the available processors according to the requirements of each project. Users can employ the Crystal multicomputer in several ways. Projects such as operating systems and database machines that need direct control of processor resources (clock, memory management, communication devices) can be implemented using a reliable communication service (the ``nugget`` that resides on each node processor. Projects that prefer a higher-level interface can be implemented using the Charlotte distributed operating system. Finally, users interested in Crystal principally as a cycle server can run UNIX jobs on node machines using the ``remote``unix service. Development, debugging, and execution of projects can take place remotely under the control of any of several UNIX hosts. Acquiring a partition of machines, resetting each machine, and then loading an application onto each machine is performed by invoking a UNIX-resident program (the ``nuggetmaster'' ). Communication with node machines in a partition is facilitated by a virtual terminal and window mechanism. Crystal is fully operational and has been used to support a variety of research projects.

## 1. Introduction

The Crystal multicomputer project began in the fall of 1981 with funding from the National Science Foundation's Coordinated Experimental Research Program. The goal of the project was to design and implement an environment to support research in distributed computation. At that time, research in progress included programming language design for writing distributed systems, tools for debugging and evaluating the performance of distributed systems, distributed operating systems, multiprocessor database machines, and parallel algorithms for image analysis. The variety of ongoing projects requiring this type of test-bed and our experience with two earlier multicomputers led to the concept of a "software partitionable multicomputer." Crystal would support multiple parallel experiments, each running in a dedicated partition of the multicomputer, with establishment and termination of partitions controlled by software.

When the Crystal project was conceived, our research in distributed systems was (inadequately) supported by two earlier multiprocessors. The first consisted of five LSI-11/03 processors completely interconnected by point-to-point word-parallel interfaces. The second consisted of eight LSI-11/23 processors linked together with a 1 megabit/second local area network and a multiport memory locally designed and implemented. While these two multiprocessors provided an experimental test-bed for some of our research activities, both systems had several serious limitations that made expansion undesirable. While the addressing structure and limited performance of the 11/03 and 11/23 processors certainly limited the usefulness of these two systems, reliability and "ease-of-use" were much more significant limitations. The point-to-point connections were the source of numerous failures in the first system. Knowing exactly what boards to wiggle became an important part of getting research done. In the second system (which was designed to avoid the problems that plague point-to-point connections), we were constantly plagued by hardware problems with the processors. In the two years the system was operational there was only a period of about two months when all eight processors were working.

Despite all the hardware problems (which perhaps never disappear in a multiprocessor), the most serious impediment to conducting research was that neither system was easy to use. Consider, in contrast, a typical time-sharing system. In this environment, we write, debug, and execute programs with a minimum of effort and without ever leaving our desks. On the other hand, in both our early multiprocessor efforts, running an experiment required going to the machine room to physically reset the machines, typing magic numbers to start the boot process, and then watching the consoles of each processor to gather debugging information and the output of the experiment. The entire procedure was complicated and discouraged use of the facility except by a dedicated few.

A final problem we encountered with our earlier efforts was sharing. Neither system could be used by two research projects simultaneously even if each project only required a few processors. Consequently, development of software proceeded at a rate slower than if a resource-sharing mechanism had existed.

Crystal was designed in an attempt to build a relatively large multiprocessor that would alleviate the limitations and problems of our earlier systems. The first issue to be resolved was whether to use shared memory as the basis for the design of the system. While there is no question that certain classes of parallel algorithms benefit from the availability of shared memory to communicate intermediate results, we saw several drawbacks to such a system. First, a shared-memory system would have required constructing a great deal of custom hardware: at a minimum, a memory subsystem and some sort of interconnection device for connecting processors to the memory. While we felt we had the necessary expertise, we were wary about putting all our energy into constructing the hardware only to "run out of gas" before finishing the software necessary to make the system usable. The second problem we saw with shared memory organizations was scalability. While we never anticipated constructing a system with thousands of processors, we wanted to insure that if the system were successful, it could be expanded with a minimal amount of effort.

Given the problems we saw with shared memory, we decided that Crystal should be implemented using off-the-shelf processors ("node machines") interconnected using very high speed local-network interfaces (100 megabits/second). Processes cooperating on a task would use messages to communicate with one another. While we understood that messages can be expensive, the work of Spector [SPEC81] and LeBlanc [LEBL82] made a high-performance system based on the message paradigm seem plausible.

To facilitate sharing the Crystal node machines, the notion of *software partitioning* was included as a fundamental component of the Crystal design. Software partitioning involves the capability of dividing the available processors into disjoint subsets of varying size. Partitioning is controlled by software. For example, at a given time there might be three active partitions: one with 3 machines, one with 20 machines, and one with 5 machines. In addition to providing resources commensurate with the current needs of an experiment, software partitioning provides a mapping mechanism from virtual machine addresses (1, 2, 3, ... ) to physical machine addresses (22, 19, 1, ... ). Thus, algorithms can be developed independently of the physical machines actually allocated. As discussed in Section 2, we had hoped to acquire network interfaces that would enforce the established partitions. Since these interfaces never became available, this mapping is performed in software.

Two factors make the Crystal multicomputer easy to use. First, we have designed and implemented software and hardware that permits a user to load programs onto the node machines in a partition with a single command on a host machine (any of several VAXes running Berkeley Unix[1]). Furthermore, through the use of windowing software, debugging and output information is delivered to the terminal in the user's office.

The Crystal multicomputer is a tool and *not* an end in itself. In many aspects it is similar to a particle accelerator. In both cases a significant research and development effort is required to build the tool before the truly interesting research can begin. At this stage, we have finished constructing the tool. We have begun to use Crystal as the basis for our research efforts.

While Crystal and Locus [POPE81] may appear similar they are quite different. The principal objective of the Locus project was to produce a distributed operating system that hides the multimachine nature of the underlying hardware as much as possible. The goal of Crystal, on the other hand, is to make the parallelism visible but as easy to use as possible. Running an operating system such as Locus in a Crystal partition would be a natural application of the Crystal multicomputer.

In Section 2 we introduce the hardware components of the Crystal multicomputer. Section 3 provides a detailed description of the key software components. In Section 4, we describe several applications we have constructed using the Crystal facility. Our conclusions and plans for the future are presented in Section 5.

[1] Unix is a trademark of Bell Laboratories

## 2. Crystal Hardware

Our proposal for the Crystal multicomputer was funded in June 1981 by a grant from NSF's Coordinated Experimental Computer Science Research Program. The first phase of the project involved evaluating hardware alternatives, designing system software, and implementing a prototype using the existing LSI-11/23-based multicomputer. We had expected the task of selecting a node processor and an interconnection technology to be completed in about 6 months. In fact, it took almost 18 months. In this section we discuss our selection of hardware components. The software components of the Crystal multicomputer are described in the following section.

### 2.1. Design Alternatives

The first task was to choose a general design for the multicomputer. Given a fixed equipment budget we identified three alternative designs. The first was to purchase a large number (100 to 10,000) of very simple processors (e.g. Intel 8088s). ZMOB [RIEG81] and Non-Von [SHAW82] are two examples of this approach. The obvious advantage of the "army of ants" approach is the amount of parallelism available. Despite this advantage, we saw several disadvantages with this approach. First, our experience with the multicomputer built using the LSI-11/23 indicated that a 16-bit address space complicates software development and limits potential application programs. A 24-bit address space was deemed to be the minimum acceptable. A second problem with this class of machines is the limited (or non-existent) support for floating-point operations. Third, research by LeBlanc [LEBL82] indicated that the time required to transmit a message between two processors connected via a local network is much more sensitive to CPU performance than to the bandwidth of the communications medium. While this result might not apply to tightly-coupled, message-passing architectures such as ZMOB or Non-Von, our desire to use "off the shelf" hardware dictated that the Crystal node processors would be relatively loosely coupled.

The second alternative considered was a small number of very powerful processors. Acquiring machines in the VAX-11/780 class would have eliminated the address space and performance problems associated with the smaller processors. However, since we would have been able to only purchase a limited number of machines (about 8), the limited amount of available parallelism would have reduced the usefulness of the system.

## 2.2. Selection of Node Processors

During this phase of the project, the first of the "small" 32 bit machines (e.g. Motorola 68000 family) were becoming available. These machines solved the addressing and performance problems of the previous generation of small machines at a price that permitted acquisition of between 40 and 50 processors. Since this number of machines was in our desired range, we elected to pursue this approach.

The hardware we needed, however, was "just around the corner." By January of 1982, we had identified four likely candidates for node processors: the VAX 11/730, the HP 9000, and two packages based on Motorola 68000 processors: one from Sun Microsystems (the SUN-1) and one from IBM Instruments Division (the IBM CS/9000). At that time neither the VAX 11/730 nor the HP 9000 had been announced. By June 1982 we had benchmarked each of these systems, even though the HP 9000 still was unannounced. None of the systems was without its problems. The Vax 11/730 was slow, memory management was not yet available for the 68000, the reliability of the SUN and IBM Instruments hardware were unknown, and the announcement date for the HP 9000 kept slipping. Despite its production problems, the HP 9000 became our first choice, and we waited for its announcement (September 1982) to start the formal acquisition process. However, between our internal decision to use HP 9000's and the end of the bidding process, we began to have doubts regarding the 9000's architecture and software development environment. In addition, DEC offered us a substantial discount on VAX 11/750 CPU's (no disks and no consoles). This level of discount would permit the acquisition of 40 node machines during the course of the project. Except for the amount of floor space consumed by each CPU, the 750 satisfied our architectural requirements and provided an acceptable level of performance. Furthermore, the 750 was known to be a reliable machine - a prime consideration. Hence, we selected the 750 for node machines. As of July 1984, 20 machines, each with 2 megabytes of memory, have been installed. Four machines have disk drives attached (Fujistu Eagles).

## 2.3. Interconnection Technology

The original Crystal proposal suggested that node machines would be interconnected using frequency-agile, broadband interfaces. In many ways this technology is similar to that used in the cable-television industry. Standard coaxial cable can carry signals spread over a 300-megahertz frequency range. By dividing this spectrum into fifty channels of 6 megahertz each, a single piece of coax can effectively support fifty 2-megabit/second connections between pairs of processors. Our idea was to use the different channels to emulate alternative interconnection topologies among the processors and to help isolate processors in different partitions from one another.

In parallel with our efforts to select a node machine we tracked the development of the frequency-agile, broadband technology. In particular, soon after the start of our project, Sytek announced a product named System 40. System 40 was to support five channels of 2 megabits/second each. Unfortunately, shipment of this product was delayed, and it was recently discontinued. While many factors may have contributed to cancellation of the product, it appears that frequency agile modems are difficult to build.

Once we saw that it would probably not be feasible to use broadband technology, we began exploring alternatives. At that time our production VAXes were interconnected using a 10 megabit/second token ring from Proteon Associates [PROT83]. The ring is implemented with two cards per machine: a ring card and a host-specific board. The ring card is responsible for implementing the token ring (packet transmission and reception and token regeneration). Incoming and outgoing packets are buffered in the host-specific board, which implements both input and output DMA engines. The token ring card is implemented in standard TTL logic. To obtain the same total bandwidth and partitioning capability of broadband technology, we contracted with Proteon Associates to provide a version of the token ring card implemented in ECL. Using ECL technology increased the raw bandwidth of the ring to 100 megabits/second. The available bandwidth is reduced to 80 megabits/second through the use of an 8-out-of-10 encoding scheme. This encoding scheme permits the detection of double-bit errors and correction of single-bit errors on a per-byte basis.

In addition to improved performance and error correction capabilities, the new ring interface card also provides a queue for incoming packets and implements a group-addressing scheme. This addressing scheme permits partitioning of nodes into 16 groups of 16 processors, 32 groups of 8 processors, and so on. We are examining two alternative ways of using this group-addressing capability. One is to use the group addresses to aid in isolating software partitions from one another. An alternative use of the group addressing capability is to assign each processor to two addresses: 2i and 2i+1 [BARA83]. When the application program (called the client) on a node has posted a receive, the device listens on address 2i+1. Otherwise, the device listens on address 2i. A message intended for the resident software (called the nugget), not the client, is sent to the group of both addresses. A message intended for the client is only sent to address 2i+1. Therefore, the nugget will always get messages bound for it, and messages for the client are only accepted (by the hardware) if the client is ready. Rejected messages can be detected by the sender. Therefore, software acknowledgements should be not be necessary for either client or nugget messages.

As discussed in Section 1, a principal goal of the project was to make it easy to use Crystal. Being able to reset and boot machines remotely is a critical component of this goal. To enable us to reset a machine remotely, we built a device known as the "break box". The break box is attached to one of our production UNIX machines via a parallel interface and by optically-isolated wires to the reset buttons of the node machines. When the address of a node machine is deposited in the output register of the parallel interface, the addressed machine is reset. We developed a boot ROM that will reboot the Crystal software (that is, the nugget) via the communications hardware whenever the machine is reset. This ROM when combined with the break box makes "hands-off" control of the Crystal multicomputer straightforward.

## 3. Crystal Software

### 3.1. Introduction

The Crystal multicomputer was designed to facilitate research in distributed systems. To achieve this objective, the software had to fulfill a number of requirements. First, it had to be flexible so that projects ranging in complexity from a parallel sorting algorithm to a distributed operating system could be accommodated. Second, it had to make the hardware as easy to use as a conventional timesharing system. Finally, the software had to permit the simultaneous use of the hardware by multiple experiments (ie. software partitioning). Two components, the **nugget** and the **nuggetmaster**, are the keys to achieving these objectives. Their location and relationship to the other software components of Crystal are illustrated in Figure 1.

To support software partitioning, two functions must be implemented, First, a user must be able to acquire a partition of idle node machines. Second, an experiment running in one partition must be prevented from sending messages to an experiment running in another partition. To obtain a partition of machines, a user (such as User 1 in Figure 1) invokes the **nuggetmaster** on a host machine and requests the desired number of machines. If available, these machines are assigned to the user. If the requested number of machines are not available, the user can place a reservation for a future time.

Once a user has obtained a partition of machines, the nuggetmaster is then instructed to load the software that comprises the experiment onto each node machine. This software generally consists of two components. Those pieces that run as processes under 4.2 Unix on a host machine are termed **application programs**. The software which is loaded onto a node machine is termed a **client**. Generally, the application program is used to provide input data to the clients and to gather the results from an experiment.

The remaining functionality necessary for software partitioning is implemented by the **nugget**. The nugget is a simple communications kernel that resides permanently on each node machine. The nugget provides message-passing primitives to the client and insures that a client does not send messages to nodes outside the user's partition.

Four features help make Crystal easy to use. First, the nugget virtualizes node numbers. Thus an experiment that uses five node machines can use numbers 1 to 5 as the addresses of the various machines regardless of which physical machines are allocated by the nuggetmaster. The nugget also simplifies implementing an experiment by providing high-performance communication services between clients on node machines and between a client on a node machine and an application program on a host machine. Third, a **virtual terminal package** aids in debugging experiments. Consider, for example, User 2 in Figure 1. This user has an experiment running on node machines #1 and #2. Rather than observing the experiment's output on the consoles of the two machines, the user has the virtual terminal package redirect the console output from each node to a window on his terminal. Finally, a **file server**, which provides data storage for users of the Crystal multicomputer, runs as a "permanent" client on one node machine.

The level of functionality provided by the nugget is the key to making Crystal a flexible research facility. From the viewpoint of someone wishing to implement an operating system, the services provided by the nugget do not overlap with those traditionally provided by an operating system (processes, virtual memory management, a file system). Thus, the nugget does not get in the way. In fact, to an operating system, the nugget can be viewed as an intelligent device driver for the communications interface. On the other hand, the reliable communication services provided by the nugget make it simple to directly implement a parallel algorithm as a client program.

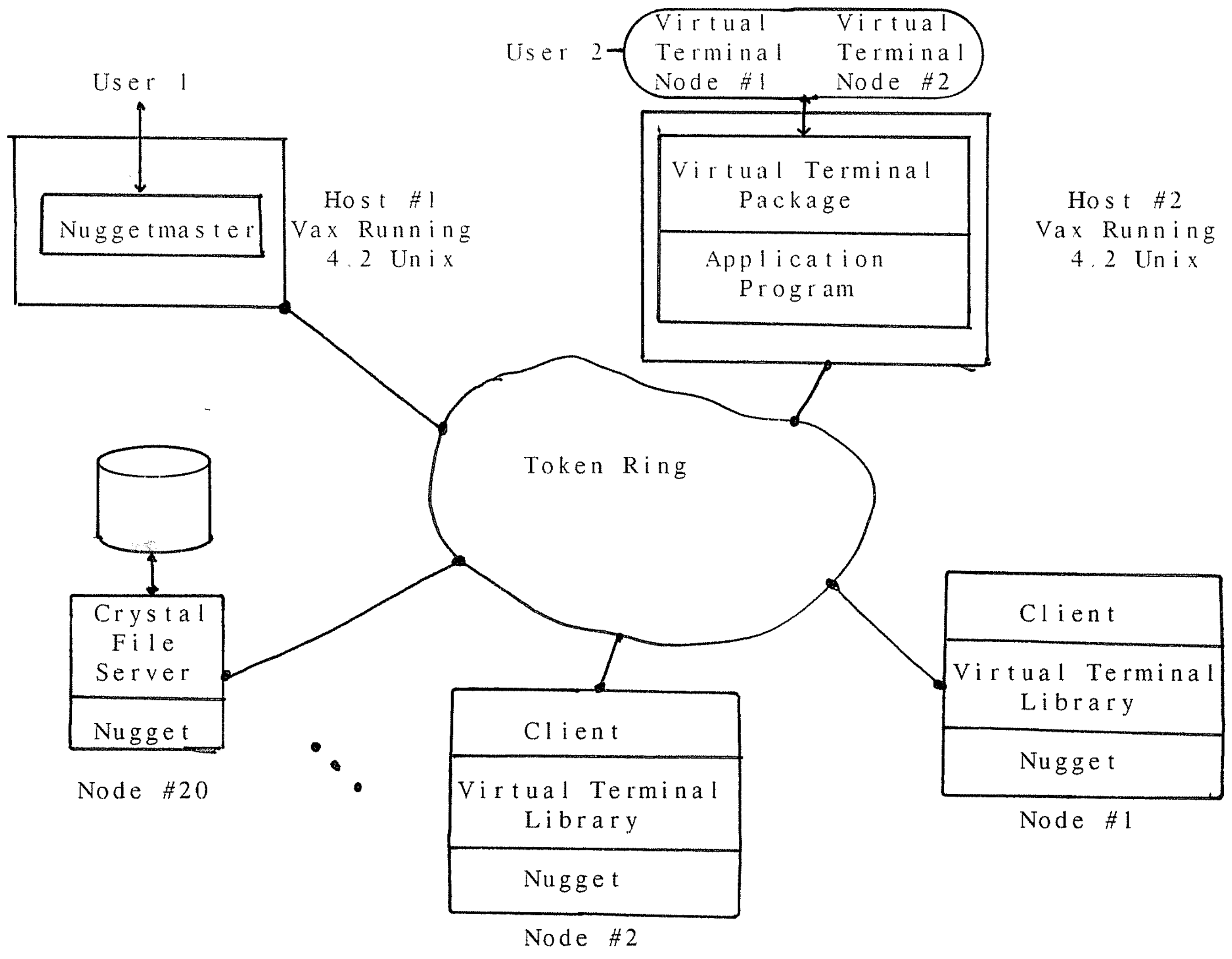

Figure 1

## 3.2. Nugget

One set of choices that faced the project at the outset was the set of features that the nugget should provide and how it should provide them. Since we did not want to preclude developing multiple operating systems for Crystal, we decided that the nugget should not provide processes, virtual memory management, or a file system. In deciding the "best" way of providing services, we were torn between a "virtual-device" approach, in which all nugget facilities are presented as virtual device registers, and a "package of subroutines" approach, in which services are presented through subroutine calls. Our design has tended to the former, although once registers are set up, events are initiated by subroutine call. Completion of events is very device-like, with asynchronous interrupts, since clients may be multi-programmed operating systems that can schedule useful work while waiting for a communications operation to complete. This design imposes the cost of a subroutine call and an interrupt for every send or receive; a pure approach of either variety might have been more efficient, although not as appropriate for many applications.

The nugget provides the following services to the client running on its node:

**Virtual node numbers**

Virtual node numbers run from one to the size of the partition. Node zero always refers to whatever Unix host machine the user employed to load the experiment. Since all communications are directed through the nugget to virtual node addresses, it is impossible for an experiment to interfere with the correct functioning of work in progress in any other partition. Experiments can be easily coded to make use of arbitrary numbers of nodes, since each client can discover the size of the partition and its own virtual node number.

**Communication**

The nugget abstracts the underlying Proteon token ring into a device that can gather up to three regions of data, queue them for sending in a single packet to any (virtual) destination, and interrupt when the send is finished. Up to 2K bytes of data may be sent in a packet. Similarly, there is a device that accepts packets (from any source) and scatters them into up to three regions of store, identifies the sender, and interrupts when the packet is received. Messages may be sent reliably, if the client desires. An alternating-bit positive-acknowledgement protocol (with piggybacked acknowledgements) is used for reliable packets. The delta-T mechanism [WATS81] is used to re-establish sequence numbers after crashes. Datagram service is also available; datagrams are not acknowledged and are discarded if they arrive at a machine whose "receive" communication device has not been enabled.

**Clock**

Since, the nugget uses the clock for its own purposes (ie. message timeouts), it provides a virtual clock for the client. This clock can be set to interrupt after any number of ticks. (A tick occurs every 10 ms.)

The nugget is implemented in Modula using a locally developed Modula compiler. The nugget has been designed three times. The initial design was demonstrated on the network of PDP-11 machines, the revised design is in use on the VAX network, and a third revision is under development. We have measured the speed of the nugget using the 10 megabit/second Proteon ring by sending 10,000 messages between a dedicated sender and a dedicated receiver. The results are shown in Table 1. We estimate that the raw rate for datagram messages is 4.0 Mbps: the overhead for sending a datagram message (exclusive of DMA time) is on the order of 2.0 milliseconds.

| Message Length | Transmission Time | |
|---|---|---|
| In Bytes | Reliable | Datagram |
| 200 | 5.7 ms. | 2.2 ms. |
| 500 | 6.6 ms. | 2.8 ms. |
| 800 | 7.7 ms. | 3.5 ms. |
| 1100 | 8.8 ms. | 4.1 ms. |
| 1400 | 9.7 ms. | 4.7 ms. |
| 2000 | 11.8 ms. | 6.0 ms. |

Table 1

## 3.3. Nuggetmaster

The nuggetmaster is a program employed by users for allocating and deallocating partitions, controlling the computation within a partition, and controlling network accesses from a host. The nuggetmaster defines a user's interactions with the Crystal multicomputer. As shown in Figure 2, the nuggetmaster is composed of two logical parts, a *resource monitor* and a *user interface.*

On each host, a nuggetmaster daemon process functions as a local resource monitor. One local resource monitor is designated to also function as the global resource monitor for Crystal resources. Current allocation tables are stored in this resource monitor. These tables indicate the status of each node machine and the partitions that are currently in force. Specific characteristics of node machines (such as the amount of main store and the presence of a disk) are also recorded there.

The *user interface* consists of a *library package* and a *command interpreter* program. The library package is available on each host for linking with other Unix programs. It communicates with the local resource monitor and the global resource monitor through inter-process communication facilities of Berkeley Unix 4.2. The library package provides routines for acquiring and releasing partitions, linking the nugget with object files, downloading programs into a partition, and debugging running programs with operations like *halt, run, peek,* and *poke.*

The interactive command interpreter uses the library package and may be run on any host machine. It provides a convenient interface to manipulate Crystal resources. It also maintains a log file for the various hardware and software components of Crystal, automatically sending mail to maintenance personnel when complaints are registered.

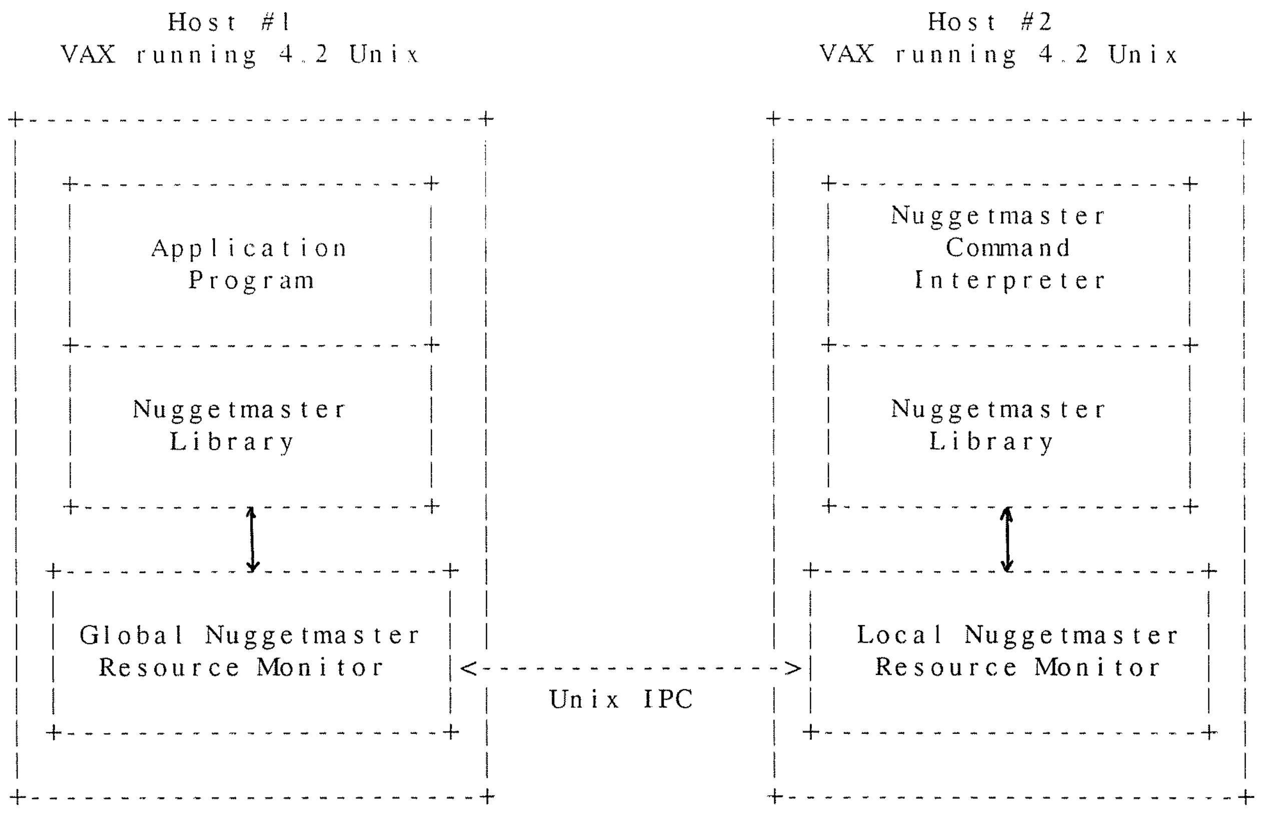

Figure 2

## 3.4. Communications Protocol

Communications among node machines and between a node machine and a host machine is done using a token ring from Proteon Associates. An alternating-bit positive-acknowledgement protocol (with piggybacked acknowledgements) is used for reliable messages. A delta-T mechanism [WATS81] is used to re-establish sequence numbers after crashes. This protocol is implemented in the nugget and as a device driver in the Unix kernel.

On a Unix host, access to the device is through *sockets*[2]. A socket can be opened by only one process at a time, but a subprocess may inherit an open socket. Each socket can handle at most one read or write request at a time. If more than one process tries to use the socket simultaneously, the first request is initiated and the others are queued.

A read or write request on the host uses a packet supplied by the application program. In the case of a write request, the packet header supplies the driver with addressing information (node number and socket number). On a read request, the packet header is filled with the same information by the driver, so the user can determine which node machine sent the packet.

Unix calls are provided to manipulate permissions that individual sockets have to communicate with individual node machines. A library of protocol routines provides simplified access to the device. This library includes routines for setting access permissions, calculating cyclic redundancy check values, printing the current permissions map, and opening a particular socket (or finding an available socket to open).

A library of programs that uses these protocol routines is available for debugging the network. These programs can print statistics of network behavior, send files between host machines along the network, and generate particular patterns of traffic on the network.

Typically, an application program on a host machine will open a new socket to use for communicating with clients in the nodes of a given partition. The program will then send and receive messages through that socket. The first message it sends will contain its socket number, so the client program will be able to direct

[2] The standard device names are "/dev/dt/1" for socket one, "/dev/dt/2" for socket two, and so forth. Socket zero is reserved for any super-user process.

responses back to the program.

## 3.5. Virtual terminal

Although some of the node machines have terminals, a typical experiment does not require that the user be present in the machine room. Instead, output can be redirected to a process on the Unix host machine and displayed there. Likewise, the Unix terminal can be used for data entry. This feature is known as the *virtual terminal interface*.

The virtual terminal interface is composed of two parts. One is a library of routines linked with the client program that resides on node machines. The other is a Unix program that communicates with those routines.

The Unix program makes use of the nuggetmaster library package to establish communication with the node machines. It also uses a library of output routines [TORE83] to create one window on the screen for each node in the partition. We currently use character-only CRT devices (of several brands), so the windows are somewhat constrained. Each window is labelled with its node number and window name.

Only one window may be opened for input at a time. Output from nodes is put in the appropriate window whether or not the window is currently open, but only the current window's display is kept up to date on the screen. Optionally, update can be restricted to times when a key is typed, to prevent programs that produce voluminous output from locking the server in "refresh mode".

The virtual terminal can be switched between input and command mode. In command mode, windows can be manipulated, and terminal characteristics (such as echoing) can be adjusted. The virtual-terminal program may be suspended and resumed. When it is stopped, nodes that try to output to the virtual terminal will wait for the host program to resume. When the virtual terminal terminates, nodes direct their output to the real console terminal.

The virtual-terminal library provides the client program with raw, blocking input. No buffering is done; each character typed at the host terminal is sent in its own packet to the node. Output from a node to its virtual terminal is collected in buffers, whose size (limited to 2000 bytes maximum) is at the discretion of the client. A buffer is flushed (sent to the host) when it fills or is explicitly flushed.

The client may switch back and forth between the physical and virtual terminal. When the virtual terminal is in use, calls to character input and output routines are transformed into messages to the host program. Although these messages are directed through the nugget, the client never sees completion interrupts from the nugget caused when such a message is sent by the virtual terminal, since the nugget is given an interrupt address inside the virtual terminal package for these messages. On the other hand, messages arriving for the virtual terminal do generate interrupts that the client must handle. These packets (and the interrupts arising from them) can be distinguished from other packets since they arrive on a pre-defined socket reserved for the virtual terminal. The virtual terminal library includes a routine that the client should call when such interrupts occur.

## 3.6. Simple Applications

Crystal users who want the support of a general-purpose multiprogrammed operating system will use the Charlotte operating system (described below). However, applications that need only one process per Crystal node machine may not wish to suffer the additional overhead due to Charlotte. For such applications, we have created the **simple-application package**, which is a set of subroutines that allow applications programmers to use the nugget for communication without having to write interrupt-handling code (see Figure 3). Versions of the simple-application package are available for clients written in Fortran, Modula, Pascal, and C.

The simple-application package supports buffered communication using two queues, one for input and the other for output. As soon as initialization is complete, incoming messages are accepted and queued in the input buffer queue. Each message is identified by the sending node's virtual node number, and a flag that indicates that the buffer is full; the client program may busy-wait on these flags. The client calls another routine to release an input buffer. To send a message, the client busy-waits until a send buffer is available, fills it with data, indicates its destination, and then invokes a non-blocking "send" library routine. The client can discover when the message has been safely sent by inspecting the busy flag of the buffer.

Communication with the UNIX host is like communications with any other node. Its node number is 0. A **simple-application library** is also available for application programs running on a host computer. This library includes a blocking receive call, since a non-blocking receive is not available from our delta-t device driver. The host library uses the nuggetmaster library package to acquire a socket from the delta-t driver and to communicate through it. The first message sent by the application program on the host to a client on a node informs the simple-application package on the node of the proper socket to which to send all traffic destined for the host

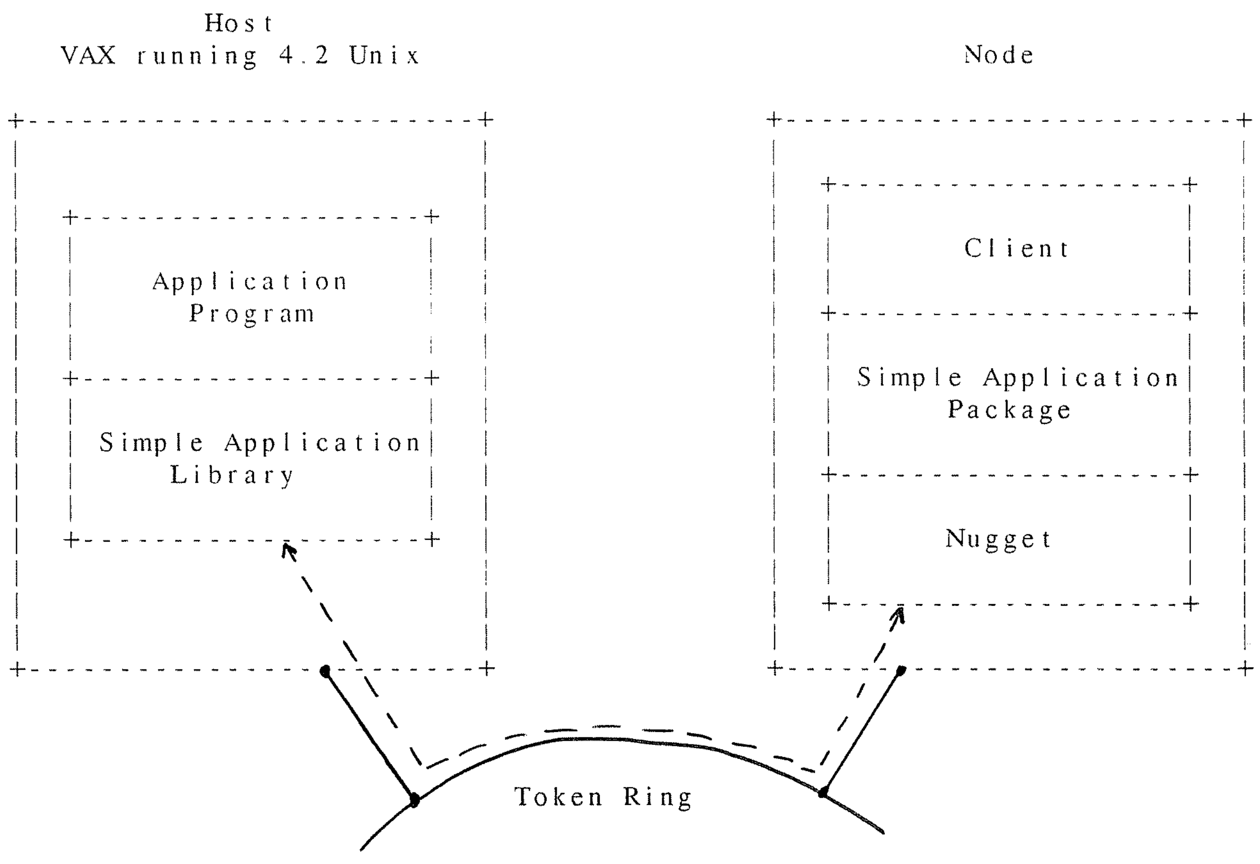

Figure 3

## 3.7. File Server

To provide long-term storage of data on Crystal, we have implemented a file server based on the Wisconsin Storage System (WiSS) [CHOU83]. The services provided by the file server include structured sequential files, byte-stream files as in Unix, $B^+$ indices, stretch data items, a sort utility, and a scan mechanism. A sequential file is a sequence of records. Records may vary in length (up to one page in length), and may be inserted and deleted at arbitrary locations within a sequential file. Optionally, each sequential file may have one or more associated indices. The index maps key values to the records of the sequential file that contain a matching value. The indexing mechanism is also used to construct Unix-style byte-stream files (the pages of the index correspond to the inode components of a Unix file). A stretch item is a sequence of bytes, very similar to a file under Unix. However, insertion and deletion at arbitrary locations is supported. Associated with each stretch item (and each record) is a unique identifier (RID). By including the RID of a stretch item in a record, one can construct records of arbitrary length.

The file server runs as a "permanent" client on one node machine. Access to the file server by a client in a partition is accomplished through a set of message-based utility routines linked to the client. The file server supports simultaneous access by multiple clients from multiple partitions. In addition to being able to execute "normal" file operations remotely (such as create, destroy, open, close, seek, read, write), a client can also scan a sequential file for records with fields that satisfy a given Boolean predicate (whether or not a suitable index is available).

## 3.8. Overview of Software Use

In this section we outline the steps that an average user might take to prepare and run an experiment under Crystal. For this example, we assume that the experiment is to evaluate the running speed of a distributed algorithm for solving some numerical problem. The algorithm has two kinds of processes, one supervisor (that will run on a Unix host machine) and any number of workers. Each worker corresponds to a client and will be assigned to its own node machine. The more workers are available, the faster the solution should be. Since workers run very similar code, only one worker program is to be written.

The first step is to develop the algorithm. Distributed and serial algorithms have both similarities and differences. At this stage, little general advice can be given the designer of parallel algorithms. The fact that the problem has been decomposed into supervisor and worker roles implies that the designer has already put some thought into the proper division of work and flow of data.

The next step is to encode the algorithm. Since the experimenter of our example has programmed serial solutions to this problem in Fortran and feels comfortable with that language, it is also chosen here. The program for the worker will need to know which worker it is and how many workers exist. This information can be obtained directly from the nugget, but the experimenter is unwilling to deal with the nugget directly, so Fortran-based simple-application package is chosen instead. This package can be used by the worker to obtain relevant parameters of an experiment. In addition, it provides a simple way to direct messages to other workers and to the supervisor. Data values should be transmitted as soon as they are ready, and incoming data values should be ignored until they are needed, in order to overlap computation with communication to the fullest extent.

For this experiment, the supervisor will run on a Unix host. This program will interactively ask for the partition number and will then use the Fortran simple-application library to find out how many workers there are and to communicate with them. In addition, the supervisor can read Unix files to get the initial data for the problem and to store the results of the numerical computation. These initial data can be sent as needed to the workers. Unix provides a clock, which allows the supervisor to time the entire experiment.

Once the supervisor and workers have been programmed, they are compiled on the host and linked with the appropriate interfaces. The recipe for compilation and linking can be preserved in a file and executed automatically under Unix.

The next step is debugging the program. To get a partition, nci (the nuggetmaster command interpreter) is invoked interactively on Unix. The experimenter wants to debug the algorithm on a partition of one node. The "new 1" command to nci acquires a node and reports the partition number. Nci is then told to link the worker program with the nugget and to load it onto that partition. Leaving nci in the background, the experimenter then starts the supervisor program, telling it the partition number interactively. The supervisor starts sending data to the worker, and the program is running. If all goes well, the supervisor will receive the answers it is expecting and display them. If not, the worker will encounter some error and will be unable to continue, or a logical error in the communication interface between supervisor and worker will cause them both to enter a state where they are waiting for messages from the other.

Debugging is hard enough on a serial program: it is much harder in a distributed one. Although tools are in preparation to assist the Charlotte user in debugging distributed programs, these tools will not help our experimenter, who will most likely place output statements in both supervisor and worker programs to indicate the current state after major events. The output from the worker can be directed to the host machine to be displayed on a virtual terminal, which the experimenter can start up at the same time as the supervisor program.

Once a single-worker version of the program works, a multiple-worker version can be tried. The nci program is told to build a larger partition, but otherwise the scenario is the same. New bugs are likely to surface as the inter-worker communication is tested for the first time.

When the program has been thoroughly tested for accuracy, timing tests can be made by the supervisor, which records the time both at the start and at the end of the computation. The test can be repeated several times with different partition sizes. Each worker can also collect its own timing statistics, distinguishing time spent computing and communication.

The scenario given above can include more sophisticated use of current facilities. Instead of using the nci program, the supervisor could use the nuggetmaster library to acquire the desired partition. It could then avoid interactive use, and could run extended experiments over a long period of time (several days) repeatedly acquiring different-sized partitions and running many problems on each partition.

The file server could also be used to advantage. Instead of sending initial data between the host and the nodes, data (including intermediate information) could be saved on the file server and recalled when needed. Exclusive access to this file service can be guaranteed while measurements are being taken.

If the experimenter contemplates many processes, a number of which are often idle at any time, Charlotte would be a better vehicle for the experiment. Many more processes can be formed than there are node machines, and process migration will be available to balance the number of active processes on each machine.

## 4. Four Example Applications of the Crystal Multicomputer

In this section we describe four of the projects that have been implemented using the Crystal multicomputer facility. First, we present two examples of the use of Crystal for research on parallel algorithms. Next, we discuss a facility known as "remote unix" that permits users to place computationally intensive Unix jobs on node machines. Finally, we describe the Charlotte distributed operating system.

### 4.1. Recursive backtrack

Many time-consuming applications have the general form of recursive search of a tree. One can list applications of this style in order of increasing complexity:

(1) Recursive backtrack, which finds all solutions to some problem by visiting the entire tree and counting or listing all the leaves. The eight-queens problem is a good example: Find all the ways eight queens can be placed on a chess board so that no two attack each other.

(2) Branch and bound, which finds the least-cost path in a tree by visiting all paths, but pruning a path before reaching a leaf if the path is already more expensive than a previous path. The traveling salesman problem is a good example: Find the cheapest order to visit n cities, given the cost of traveling from any city to any other.

(3) Tree evaluation, which derives some recursively-defined value for a tree by combining the values of all the children according to some set function. Negamax evaluation of a checkers lookahead tree is a good example: The value of the current board position is the negative of the maximum of the values of the possible successors to that board position.

(4) Tree evaluation with cutoffs, which performs a tree evaluation but uses locally-derived information to prune parts of the tree that are certain not to effect the value of the whole tree. Alpha-beta search is a good example: A window of acceptable values is inherited by each node in the tree, and if any child returns a value outside the range, no other children need to be examined.

Parallel algorithms to perform recursive search have been investigated for some time [BAUD78, FINK83, AKL80].

Recently we have developed a general-purpose recursive search package [MANB84] that is easy to interface to any application of the first two types of search: recursive backtrack and branch-and-bound. Extensions to the third and fourth types are underway.

The application program needs only to specify what the root of the tree looks like, how to generate a first child or a sibling node, and what constitutes an answer. The package, written in Modula, uses inter-machine communication (based on the simple-application package) to distribute work to all the machines in the partition.

The algorithm used for distribution is both simple and effective: When a machine has finished all its work, it asks a neighbor for work. On receiving this request, the neighbor gives away half its remaining work. If the neighbor is itself waiting for work, it forwards the request. The machines are arranged logically in a ring. If a machine hears its own request, it drops out of the computation.

For problems of the first type, a machine reports each answer as it is found to the host machine. If only a count is desired, the count is sent only when an entire batch of work has been finished and the machine is idle. For problems of the second type, new bounds can be generated by the application; if they surpass the current bound, the package will send the new bound to the successor machine. For problems of the third type, answers must be directed back to the machine that originated the node that has been evaluated. For problems of the fourth type, updated bounds must be given to machines that are already engaged in work.

Our initial experience with this package has been very promising. Applications such as eight queens, knights tour, generating permutations, and traveling salesman have been coded quite easily. It takes less than an hour to encode and debug simple applications. Recursive backtrack problems achieve almost linear speedup with the number of machines. As an example, the ten-queens problem was run with partitions of size 1 to 7. Both work time and idle time were measured to the nearest millisecond. Idle time includes time waiting for more work and assisting a neighbor by giving it work, but does not include reporting results to the host. The results of this example are shown in Table 2. There were 724 answers; only counts were reported to the host. The amount of idle time is very small (and depends on the network size, not the problem size). All machines finish at almost identical times. The overall speedup is close to perfect.

| Partition Size | Average Work Time in Seconds | Average Idle Time in Seconds | Speedup |
|---|---|---|---|
| 1 | 156.98 | 0.03 | 1.00 |
| 2 | 78.95 | 0.62 | 1.99 |
| 3 | 52.90 | 1.14 | 2.96 |
| 4 | 39.78 | 1.39 | 3.94 |
| 5 | 32.06 | 3.63 | 4.89 |
| 6 | 26.89 | 2.56 | 5.83 |
| 7 | 23.02 | 2.41 | 6.81 |

Table 2

## 4.2. Jacobi Methods

There are a number of serial algorithms for solution of partial differential equations. As pointed out in [FISH81], many of these algorithms are locally defined and iterative: A rectangular array of numbers A is given; A′, A′′ and so forth are determined by a locally defined rule. That is, the value of an element is some function of the values of its immediate neighbors in the previous array.

Locally defined, iterative algorithms lend themselves to parallel computation in a straightforward way. The array is partitioned into a number of large, contiguous regions. Each region is assigned to a dedicated machine for computation. The internal values for a given iteration can be computed directly. To compute new boundary values we require the values of the boundaries of adjacent regions.

We have implemented the Jacobi method on four machines as a simple example of such an algorithm. This test was coded in Fortran and made use of the simple-application package. The algorithm executed by each node machine is as follows:

```
1     Discover which quadrant I am to deal with.
2     Accept initial data for that quadrant from machine 0 (the host).
3     loop
4         Estimate global error.
5         if global error small enough then exit.
6         Send boundary values to neighbors.
7         Calculate new internal values.
8         Accept boundary values from neighbors.
9         Calculate new boundary values.
10    end.
```

The most time-consuming step is in line 7. Communication is therefore overlapped with computation, since line 6 starts sending data to neighbors, but the data need not arrive until the neighbor reaches line 8. Estimating the global error (line 4) requires estimating the local error, sending it to all nodes, and accepting their local errors. Since every node therefore receives the same values, each can independently reach the same value for global error.

Our experience in programming this application has shown us that the simple-application package is appropriate for most of the algorithm, but that two features would have made the coding easier.

(1) In line 8, we expect boundary values. In line 4, we expect local error estimates. However, other nodes may not be synchronized with us, so we may, at either time, receive the other kind of message. Such unwanted messages must be saved so that later, when the data is needed, it is still available. Charlotte provides exactly the sort of receive semantics we need. Under Charlotte, different links would be used for data and for error estimates, and selective receive would easily distinguish these messages.

(2) Distribution of error estimates would logically require broadcast. Although the Proteon interface allows broadcast, we have omitted this feature from the nugget, since reliable broadcast requires acknowledgments from each destination, and separate transmission is not much more expensive. Since the nugget does not provide broadcast, the simple-application package does not, either. For larger versions of the Jacobi method, using many more machines, a hierarchical error-estimate collection scheme would be preferable to an effective broadcast.

Other approaches to the same problem are also amenable to similar decomposition; we chose the Jacobi method for its simplicity. The Gauss-Seidel method with red-black coloring is easily fit into a similar mold. In addition, we are investigating multi-grid techniques. These techniques form coarse approximations to the desired problem and solve them exactly, producing an iterative improvement toward the solution of the original problem. The exact solution to the coarse problems can be done recursively or can use Gaussian elimination. Distributed Gaussian elimination is not as easy as the Jacobi method, since information transfer is not localized within the matrix. However, we have found a different organization for Gaussian elimination (following a paradigm that looks something like pipelining) that appears promising.

## 4.3. Remote Unix

As Crystal became operational, the use of Crystal to provide additional cycles for compute-bound tasks such as simulations emerged as an obvious application. Constructing a parallel simulation facility on the Crystal node machines is a research activity we intend to pursue. Since simulation projects generally involve repeated execution of the model while varying the input parameters, an alternative way to utilize the available parallelism is to execute each simulation run on a separate machine. As we developed this idea further a more general facility that we call "remote unix" emerged.

The "remote unix" facility extends the Unix notion of forking a background process to permit the invocation and execution of a Unix process on a Crystal node machine. Our design was based on two principal objectives. First, since the Crystal node machines and our production Unix machines have the same instruction set, it seemed feasible to design the facility in such a way that recompilation would be unnecessary. Second, since many programs use the system calls provided by Unix these routines should be also be available in addition to standard input and output capabilities.

The organization of our remote-unix software is shown in Figure 4. Unix system calls issued by the program running on the node machine are trapped and translated into remote procedure calls to the host machine.

The remote unix facility is invoked by "ru X" where X is the name of the program to be executed and its parameters. Ru first acquires acquire a Crystal partition containing one node machine. Next, the "remote

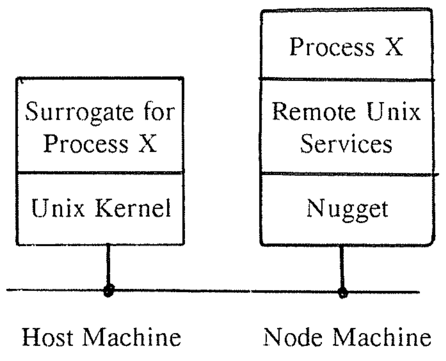

Figure 4

unix services" module is loaded onto the node machine. It, in turn, requests that X be sent for loading. Finally, ru becomes a surrogate process for X on the host Unix machine. Since it is being run on behalf of X, it has the same access rights (to files, for example) and runs in the same environment as X would have.

Once everything has been initiated, execution of X begins. Whenever X executes a Unix system call (like open, read, write), the call is trapped by the remote unix services module. This module packages a code specifying the call type and the parameter values into a message sent to X's surrogate running on the host machine. The surrogate process unpackages each incoming message, decodes the call type, and executes the appropriate system call. The result of the call is then packaged as a message and sent to the calling program via the remote unix services package. At the present time, ru supports 48 of the 85 system calls provided by 4.2 Unix.

Timing information of some simple tests in shown in Table 3 below. The first column describes the test conducted. The second column specifies the elapsed time to execute the program using the remote unix facility. The third column indicates the elapsed time to run the program on an unloaded Vax 11/750 running Berkeley 4.2 Unix. Finally, in the fourth column we have included the time to run the program on a remote 11/750 running Berkeley 4.2 Unix via the rsh command (the data accessed was already at the remote site). These results indicate to us that although the startup cost of the remote unix facility is relatively high, the system is quite useful for cpu-intensive programs.

At the present time, the ru facility has several limitations. First, the process must fit in the physical memory available after the nugget and remote unix are loaded. The nugget and the remote unix module together occupy approximately 100K bytes of physical memory. Second, the process cannot spawn additional processes. Considering the intended application, we do not view this as a serious limitation of the system. Finally, pipes are inefficient. The command, "ru A | ru B" buffers the output from process A through the invoking machine instead of being piped directly from A's machine to B's machine. We are currently considering extending the ru facility to allow "ru 'A | B".

| Test | Remote Unix | 4.2 Unix | 4.2 Unix using rsh |
|---|---|---|---|
| open file | 14.2 seconds | .29 seconds | 4.0 seconds |
| open file and read 1000 bytes | 14.2 seconds | .31 seconds | 4.2 seconds |
| open file and read 100,000 bytes in 1000 byte chunks | 21.9 seconds | 1.12 seconds | 4.92 seconds |
| open file and read 100,000 bytes in 1 chunk | 18.9 seconds | 1.08 seconds | 4.92 seconds |
| nroff a file of 30,000 characters | 49.1 seconds | 36.2 seconds | 40.1 seconds |

Table 3

## 4.4. Charlotte

Charlotte [ARTS84] is an ongoing research project supported by the Defense Advanced Research Projects Agency (DARPA). Its goal is to develop algorithms and support software for distributed computation. A major focus of this project is designing and implementing an operating system that supports closely interacting processes cooperating to solve a computationally intensive problem. The Charlotte operating system is intended to fill a middle ground between local area networks, which allow autonomous computer systems to communicate and share resources, and multiprocessor systems, in which multiple processing units communicate through a common memory space. Charlotte software is responsible for allocating processors to processes and insulating the processes from the details of inter-processor communication. The design and construction of Charlotte is intended to fulfill two goals: to explore operating-system design for multiprocess applications, maximizing functionality while minimizing overhead; and to serve as a test-bed for research in distributed algorithms. We look forward to a symbiotic relationship between Charlotte and application programs written for it. On one hand, the ease with which such applications can be written tests the quality of the operating-system primitives provided, and system overhead in running applications tests the efficiency of the implementation. On the other hand, Charlotte provides a convenient environment to test and measure multiprocess algorithms.

The current version of Charlotte has several antecedents. The first was the Arachne (originally called Roscoe) operating system, constructed in 1978 for the network of five PDP 11/03 computers [SOLO79], and substantially rewritten for the network of PDP 11/23 computers connected by the Megalink. In Arachne, an identical *kernel* residing on each processor supports creation and destruction of *processes* and communication between them. Other "operating system" services are provided by *utility processes*, including a file server, a terminal driver, and a resource manager. Inter-process communication in Arachne is based directly on the Demos operating system for the Cray-I [BASK77]. Processes address each other using capabilities to simplex logical communications paths called *links*. The *Send* primitive specifies the recipient of a message indirectly, by referring to a link; there is no way a process can name another directly. The sending end of a link can be transferred from one process to another by including it in a message. The message supplied by the *Send* call is copied from the sender's address space into a kernel buffer and the sender is allowed to continue. The kernels on the component processors cooperate in delivering the message to the processor of the destination process. A process receives a message by executing the *Receive* primitive, which specifies a buffer and set of incoming links and blocks the receiver until a message has arrived.

Charlotte is designed with several modifications to Arachne communications primitives to remedy what we perceived to be defects in the Arachne design. These included replacing simplex links with full-duplex links, using synchronized message transfer instead of kernel buffering, and introducing a more symmetrical arrangement in which neither *send* nor *receive* is blocking. We also proposed a richer set of utility processes, for example replacing the resource manager with a set of processes for managing various resources. A pilot implementation of the new operating system was written for the network of PDP 11/23 computers in Modula [WIRT77]. With the arrival of the Crystal hardware, Charlotte was extensively rewritten to run on the VAX computers using the nugget for inter-processor communication. We took this opportunity to reorganize the kernel and modify the semantics of many of the kernel primitives. The remainder of this section discusses this latest version of Charlotte.

Charlotte consists of a set of identical *kernels*, one on each processor, and a set of *utility processes*. Each kernel runs as the client program on one node of Crystal. The kernels provide mechanism for short-term process scheduling and inter-process communication. The utility processes cooperate to control medium to long-term scheduling, allocate resources, and provide higher-level functions such as file and directory services. Processes invoke kernel primitives by submitting *kernel calls*, which appear much like subroutine calls.

### 4.4.1. Inter-process communication

Processes communicate by exchanging messages over links. A *link* is a logical, full-duplex connection between two processes, each of which has a capability to one end of the link. A process never refers to another process directly, but only by naming a *link identifier*, which is an index into a table maintained by the kernel. (When we speak of "the kernel", we include the case in which the two ends are on separate machines, each governed by its own kernel.) Information about the name of and route to the process at the other end of the link is stored by the kernel at each end. A process may use a link to send a message, or it may include an end of a link in a message, thereby transferring ownership of the link from the sender to the receiver. Links are created by the **makeLink** kernel call, which constructs a new link with the caller holding both ends. If a process wishes to introduce two colleagues *A* and *B* together, it can form a new link and give one end to *A* and the other to *B*.

Process-level communication is *non-blocking* (a process can generally continue executing while the kernel is transmitting a message on its behalf), *unbuffered* (a message is not transmitted until the receiver has provided a place to put it), and *synchronous* (processes are not generally interrupted by the arrival of messages).

The **send** kernel call specifies a buffer (a contiguous region in the caller's memory) containing a message, a link over which to send it, and an optional link to include with the message. The **send** call specifies that the data is to be sent when the process at the other end of the link has expressed a willingness to receive them. The sending process is allowed to proceed immediately, although it should not modify the contents of the buffer or attempt another **send** over the same link until the data actually is sent.

The **receive** kernel call also specifies a link (or the special value *allLinks*) and a buffer and allows the caller to continue. When a matching **send** and **receive** have been posted, the kernel (or pair or kernels) transfers the data (possibly breaking the message into multiple packets and using acknowledgements to assure reliability) and then posts *completion events* to indicate that the transfer is completed.

A process can wait for a completion event by executing the **wait** kernel call, which specifies a link (or *allLinks*) and direction (*send*, *receive*, or *either*) and blocks the process until a matching operation completes. The **wait** call returns the link number and direction associated with and event together with an indication of success or failure, the number of bytes transferred, and, for **receive** completions, an indication of any enclosed link. The **wait** call returns immediately with an error indication if there are no events to wait for, either completed or in progress. The **getResult** call is similar to **wait** but returns immediately if no completion events are currently posted. It can be used for polling.

The **cancel** call takes a link number and a direction (*send* or *receive*) and requests that the indicated operation be cancelled. This call blocks the process until the kernel is able to report whether the operation was successfully cancelled or it has progressed too far to cancel.

The **destroy** call destroys a link. This call is legal even if a **send** or **receive** is pending on that link. Further **send** and **receive** operations on this link are not allowed; that is, they cause an immediate posting of a completion event indicating that the indicated link does not exist. As a side-effect of destroying a link, the kernel causes a message to be sent on that link to the remote end, indicating that the link has been destroyed. This message causes any **send** or **receive** pending on the remote end to terminate with completion code *link-Destroyed*.

### 4.4.2. Process Control

The Charlotte kernel maintains a dynamically varying set of processes. Each process is associated with two regions in physical memory: an *image*, consisting of code, initialized data, and global memory, and a *stack*. The kernel blocks processes in response to **wait**, **cancel** and other blocking kernel calls, unblocks them when operations complete, and alternates among ready processes using a simple round-robin time-slicing policy. Most process management is controlled by utility processes; the kernel only provides very low-level calls to give these processes access to the basic mechanism for process control. Process-control primitives include calls to create a new process context, start the process running, halt it or allow it to resume running, examine or alter the contents of its memory, or terminate it. These process-control kernel calls may only be used by the *KernJob*, a special utility process replicated exactly once on each processor. The principal function of the KernJob is to invoke process-control kernel calls on behalf of other processes. In that way, a process can be managed by a utility process running on a different processor. These privileged calls refer to a process by a *process identifier* rather than a link, and thus violate the principle that processes may not refer to each other directly.

The **makeProcess** kernel call creates a new process, but does not start it running. It takes as arguments two blocks of physical memory to use for the image and the stack and returns a process identifier to use as an argument for other kernel calls described in this paragraph. Another argument is a link to connect the new process to its parent (the caller of **makeProcess**). **peek** and **poke** can be used to access the private memory of a process. They are used for initial loading of a new process as well as debugging. Once a process has been created by **makeProcess** and loaded using **poke**, it can be made active with the **inspire** kernel call. A running process can be halted with the **suspend** kernel call, resumed with **resume**, and forcibly terminated with **expire**. (The kernel also provides the unprivileged **terminate** call, with which any process may terminate itself; **suspend(-1)** may be used by a process to suspend itself.) The physical-memory resource of the processor is not managed by the kernel; instead the kernel provides the **getMemoryMap** call whereby a untility process can inquire about the initial state of free memory at initialization time, and accepts explicit allocations of memory for new processes. Finally, the kernel provides two calls for use in initialization and error recovery: **getProcessDesc** can be used to get a list of active processes, and **kernLink** returns a link to the KernJob on a specified processor.

### 4.4.3. Implementation

The kernel is structured into three primary modules, each of which executes a Modula process. (To avoid confusion with Charlotte processes supported by the kernel, we refer to these Modula processes as *tasks*.) The *envelope* task chooses a job to run and switches to user mode until a kernel call or an interrupt, at which point it dispatches requests to other kernel tasks if necessary. The envelope is only permitted to run when all other tasks are blocked. The *automaton* task encodes a finite-state automaton for each link. This task implements an acknowledgement protocol for flow control and error recovery, and supports the inter-process communications facilities mentioned above. The *communication* task delivers packets from the automaton task to the nugget and receives incoming packets from the nugget. These three tasks communicate with each other by means of queues of unfinished work, which they manipulate using the atomic INSQUE and REMQUE instructions of the VAX.

### 4.4.4. Utilities

One of the design goals for the Charlotte operating system is to keep the kernel efficient, concise, and easily implemented. As a result, only those services essential to the entire system, such as inter-process communication and process control, are included in the kernel. All other services are implemented through utility processes, which wait for requests coming from client links. For reasons of efficiency and reliability, a given service may be provided by a *squad* of processes distributed throughout the network. Members of a squad all run the same code, but divide responsibilities among themselves, coordinating their activities by exchanging messages.

#### The KernJob

Sometimes a process needs to communicate with a particular kernel other than the kernel of the machine on which it resides. It cannot use messages, since kernels cannot send or receive messages. Neither can it use kernel calls, since a kernel call is always interpreted by the local copy of the kernel. Therefore, Charlotte provides a squad of KernJob processes, with exactly one member on each node. For example, a *Starter* process (described below) is responsible for assigning a new process to a node. Once it has selected a node, it uses messages to the KernJob on that node to accomplish the actual process creation. As another example, any process may be given the ability to control another by giving it a *control link*. A *kill* message sent over a control link has the effect of terminating the controlled process. A control link is implemented as a link to a KernJob, which translates the kill message into an **expire** kernel call.

### The Starter

The Starter squad manages the creation of the new child processes. Each member of the squad is responsible for a set of nodes. If a process requires another process to be created, it sends a message to a Starter naming a file containing the executable code for the child.

The Starter maintains information about all the nodes it serves, including current memory allocation and current processes states. Based on this information, the Starter can choose a node and memory location for the new process. The Starter communicates with a *FileServer* utility process (described below) to obtain the child's code and data and with the KernJob on the appropriate machine to cause the child to start and to have the proper contents. Each Starter also has a link to one or more other members of the Starter squad. The Starters periodically exchange information about load situations so that requests to start a new process can be redirected to the most appropriate Starter. We plan to use this information to govern process migration as well.

### The Switchboard

Often, a process needs a link to a process providing a particular generic service. For example, a process wishing to open a file will need a link to a FileServer process. Each new process is born with a link to a member of the *Switchboard* squad. A Switchboard allows a server process to *register* a link under a given character string name, and a client process to *locate* it by supplying a pattern that matches that name. Several Switchboards may be active at a time, in which case they cooperate to satisfy client requests.

### The FileServer

A process accesses a file by sending an *open* request to a member of the FileServer squad, giving the character-string name of the file and an indication of whether the file is to be read, written, or both. Each member of the squad is responsible for a subset of the existing files. The open request is relayed within the squad to the appropriate member, which responds with a new *file* link representing the open file. From the client's point of view, the file link appears to address a new process responsible for operations on the file. If the file was opened for writing, the client may send *data* messages containing data to be written. If the file was opened for reading, the FileServer immediately begins to read data from the file and send it to the client. Thus the client can read sequentially through the file by simply receiving messages; no explicit *read* requests are necessary. A client can send a *seek* message over the file link to alter the position of the next read or write operation.

The current version of the FileServer is a simple prototype, using the Unix file system. A file server relays requests for operations to a *daemon* process running on a Unix host, which translates them into Unix operating system calls. A "native" file server is under development.

### The Connector

The *Connector* is a tool to establish initial links in a group of processes. It is implemented as a free-standing utility process registered with the switchboard. A program that wishes to institute such a group (usually a command interpreter, but in general any top-level entity in a group of processes) sends a request naming a file that describes the group's component processes and their interconnections. Each component process is described by naming a file containing code for a new process or supplying a pattern naming an existing process (to be found by consulting the Switchboard). It is also possible to specify that a process is to be searched for in the Switchboard and created if not found. Connections are specified by listing the process and link identifier of each end. (For example, the description file may state that a link is to connect link 3 of one process to link 5 of another.) The code for members of the group must start with a call to the *LinkUp* library procedure. The connector creates the required processes and carries on conversations with the LinkUp code in each. When the LinkUp procedure returns in each child, the required connections are established.

### 4.4.5. Initialization and Recovery

The code loaded on a new Charlotte node includes the kernel and two utility processes, the KernJob and the *Primordial Connector*, with kernel tables initialized to indicate a link between them. The Primordial Connector contains just those parts of the Connector, Starter, and the FileServer necessary for establishing a minimal initial configuration. It reads a configuration file (like a Connector description file) to see which utility processes are to be placed on its node and how they are interconnected with each other and with processes on other nodes. When initialization is complete, the Primordial Connector terminates.

If a machine fails, any kernel dealing with it will discover its demise within a few seconds (as a consequence of the communications protocol), and deliver *linkDestroyed* messages to local processes holding links to processes on the failed node. Most clients that discover that their servers have disappeared will just terminate. More robust ones might attempt to get replacements from a Switchboard. Servers that have lost clients can usually recover and re-register with their switchboard. If a KernJob loses its controlling Starter, it searches for another KernJob that has a working Starter and introduces itself, informing the new Starter about the state of the node, which it can access through the **getProcessDesc** kernel call. A similar mechanism is used for a machine that has just been repaired and is trying to re-establish contact with the world.

Together, these facilities allow Charlotte to survive partial hardware and software failures, maintaining continuing computations and bringing nodes back into the fold when they are repaired.

## 5. Conclusions and Future Directions

In this paper we have presented an overview of the hardware and software components of the Crystal multicomputer project. Presently (July 1984) 20 VAX 11/750 node processors are installed and in production use. While awaiting delivery of an 80 megabit/second token ring, the nodes communicate among themselves and with our Unix machines using a 10 megabit/second ring. The basic Crystal software has been completed and is in production use. The simultaneous use of Crystal by multiple research projects is accomplished by partitioning the available nodes according to the demands of the different research projects. Users can use the Crystal multicomputer in many different ways. Those projects (e.g. operating systems, database machines) that need direct control of processor resources (clock, memory management, communication devices) can be implemented directly on top of the nugget, possibly using the simple-application package. Projects that prefer a higher level interface can be implemented using the Charlotte distributed operating system. Finally, those users who are interested in Crystal principally as a cycle server can run their Unix jobs on node machine using the "remote unix" service. Development, debugging, and execution of projects can take place remotely through any host. Acquiring a partition of machines, resetting each machine, and then loading an application onto each machine is performed by the Unix-resident nuggetmaster. Communication with node machines in a partition is facilitated by a virtual terminal and window mechanism.

With the basic services of Crystal now operational, we have begun to develop enhancements. One such enhancement is a debugging tool for distributed algorithms. Although the nugget currently provides pause, peek, poke, and resume capabilities, there is no debugger that understands algorithms running on multiple node machines. In addition to helping get a parallel algorithm running, we envision that such a tool would provide performance information about the run-time characteristics of the algorithm. We also plan to enhance our library packages for developing programs that use parallelism. The recursive backtracking package developed by Manber and Finkel is one such example. Adding a "parallel-for" capability to the Fortran package is also planned.

We are also examining how to integrate Crystal more tightly into our Unix environment. One approach being considered is an implementation of Wilkes' "processor bank" concept [WILK83] as an alternative to the workstation-per-user model of computing. Two features make workstations an attractive alternative to conventional time sharing systems such as a Vax running Unix. First is the guaranteed response time that only a single user machine can provide. The second is that most workstations provide a bit mapped display with windowing software and a mouse. The limitation of workstations is not being able to "borrow" processing resources from idle machines. The idea of a processor bank is to decouple the bit mapped display mechanism from the processor and to extend the notion of a window per process to a window per processor. By having a pool of processors and intelligent, high resolution displays, a user can acquire the amount of processing capability appropriate for the task at hand by forking processes onto multiple, standalone machines.

As the first step in implementing this capability we intend to extend our current remote unix capability to support arbitrary processes. We are currently examining whether to do this by removing features (e.g. TCP/IP) from 4.2 Unix or by writing our own Unix kernel using the nugget as a basis. We anticipate that files will continue to reside on a user's "home" machine or on the Crystal file server and that paging will be done remotely to the Crystal file server. We are currently evaluating several alternative terminals including the 5620 from Teletype (the BLIT[3]) and the Superscreen from Scion.

To enhance message passing performance in Crystal, we have begun to examine the possibility of a separate protocol-processor board. Two factors appear to limit current message passing performance: the overhead of the protocol processing and by the bandwidth of the Unibus on the Vax 11/750. By implementing a separate protocol processor board that plugs into the CMI bus of the 11/750, we hope to be able to increase the raw DMA bandwidth from approximately 4 megabits/second to approximately 40 megabits/second[4]. Furthermore, by offloading protocol processing to a separate processor that shares memory with the 11/750 CPU, we expect to significantly reduce the overhead of message processing.

## 6. Acknowledgments

Many people have helped make the Crystal project a success. Larry Landweber played a key role in shaping the nature of the project and writing the proposal to the National Science Foundation for funding. Bob Cook, another original principal investigator of the project, conceived the notions of the nugget and software partitioning. Digital Equipment Corporation deserves a special note of thanks for providing an extremely generous External Research Grant that made using VAX 11/750s as node machines feasible.

However, the staff members who turned our ideas into reality really deserve the credit. Tom Virgilio implemented the nugget and together with Don Neuhengen implemented the remote Unix software. The communications software was written by Russell Sandberg. Keith Thompson and Nancy Hall implemented the Modula compiler used to implement the nugget and Charlotte. Nancy Hall also implemented the virtual terminal package. The Crystal file server was implemented by Tad Lebeck with the help of H-T Chou. Bob Gerber has been responsible for implementing the nuggetmaster. The current version of Charlotte kernel was written by Yeshayahu Artsy and Hung-Yang Chang, various utility processes were written by Prasun Dewan, Vinod Kumar, Bryan Rosenburg, and Cui-Qing Yang. Many helpful ideas and comments were provided by Aaron Gordon, Bill Kalsow, and Michael Scott. The idea of the connector is due to Tony Bolmarcich. Finally, our facility manager Paul Beebe deserves thanks for worrying about (and solving) all our

[3] Bell Labs Intelligent Terminals - marketed as the 5620 by Teletype Corporation

[4] The CMI bus of the VAX 11/750 has a bandwidth of 5 megabytes/second. We have measured the DMA bandwidth of the Unibus on a 11/750 to be 4 megabits/second

space, air conditioning, and power problems.

## 7. References

[AKL80] Akl, S. G., Barnard, D. T., and R. J. Doran, Simulation and analysis in deriving time and storage requirements for a parallel alpha-beta algorithm, Proc. 1980 International Conference on Parallel Processing, pp. 231-234, August 1980.

[ARTS84] Artsy, Y., Chang, H-Y. R. Finkel, Charlotte: Design and implementation of a distributed kernel, University of Wisconsin-Madison Computer Sciences, Technical Report 544, September, 1984.

[BARA83] Barak, A. B., personal communication, 1983.

[BAUD78] Baudet, G. M., The Design and Analysis of Algorithms for Asynchronous Multiprocessors, Carnegie-Mellon University, Department of Computer Science, April 1978.

[BASK78] Baskett, F., Howard, J. H., and J. T. Montague, Task communication in Demos, *Proceedings of the 6th Symposium on Operating Systems Principles*, pp. 23-31, November 1977.

[CHOU83] Chou, H-T, DeWitt, D. J., Katz, R., and T. Klug, Design and Implementation of the Wisconsin Storage System (WiSS) Computer Sciences Department, University of Wisconsin, Technical Report #524, November 1983.

[FINK83] Finkel, R. A. and J. P. Fishburn, Improved speedup bounds for parallel alpha-beta search, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, January 1983.

[FISH81] Fishburn, J. P., Analysis of speedup in distributed algorithms, Ph.D. thesis, Dept. of Computer Sciences Technical Report 431, University of Wisconsin-Madison, May 1981.

[LEBL82] Leblanc, T. J., The Design and Performance of High-Level Language Primitives for Distributed Programming, Ph.D. thesis, Computer Sciences Department, University of Wisconsin-Madison, 1982.

[MANB84] Manber, U. and R. Finkel, The DIB distributed implementation of backtracking, in preparation, July 1984.

[POPE81] Popek, G. J., and et. al., LOCUS: A Network Transparent, High Reliability Distributed System, *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981.

[PROT83] Proteon Associates, Operation and Maintenance Manual for the ProNet Model p1000 Unibus, Waltham, Mass, 1983.

[RIEG81] Rieger, C., ZMOB: Doing it in Parallel, *Proceedings of the 1981 IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, November 1981.

[SHAW82] Shaw, D. E., The Non-Von Supercomputer, Department of Computer Science, Columbia University, Technical Report, August 1982.

[SOLO79] Solomon, M. H. and R. A. Finkel, The Roscoe distributed operating system, *Proceedings of the 7th Symposium on Operating Systems Principles*, pp. 108-114, December 1979.

[SPEC81] Spector, A. Z., Multiprocessing Architectures for Local Computer Networks, Ph.D. thesis, Computer Sciences Department, Stanford University, 1981.

[TORE83] Torek, C., The Maryland Window Library, Department of Computer Science, University of Maryland, 1983.

[WATS81] Watson, R. W., Timer-based mechanisms in reliable transport protocol connection management *Computer Networks*, Vol. 5, pp. 47-56, 1981.

[WILK83] Wilkes, M. V., Invited Keynote Address, 10th Annual Interational Symposium on Computer Architecture, June 1983.

[WIRT77] Wirth, N., Modula: A language for modular multiprogramming, *Software Practice and Experience*, Vol. 7, No. 1, pp. 3-35, 1977.