CMU-CS-81-123

Accent: A communication oriented network operating system kernel

Richard F. Rashid

George G. Robertson

Department of Computer Science

Carnegie-Mellon University

Pittsburgh, PA 15213

DRAFT

30 April 1981

Abstract

Accent is a communication oriented operating system kernel being built at Carnegie-Mellon University to support the distributed personal computing project, Spice, and the development of a fault-tolerant distributed sensor network (DSN). Accent is built around a single, powerful abstraction of communication between processes, with all kernel functions, such as device access and virtual memory management accessible through messages and distributable throughout a network. In this paper, specific attention is given to system supplied facilities which support transparent network access and fault-tolerant behavior. Many of these facilities are already being provided under a modified version of VAX/UNIX. The Accent system itself is currently being implemented on the Three Rivers Corp. PERQ.

Keywords: Inter-process communication, networking, virtual memory, paging, UNIX, PERQ, VAX, network operating systems, distributed computation.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551.

Table of Contents

List of Figures

1. Introduction

Two independent projects, one for the development of a fault-tolerant distributed sensor network (DSN) and one for a distributed personal computing environment (Spice) are currently underway at CMU. Both projects require as their foundation a network operating system which allows flexible, transparent access to distributed resources. Accent[1] has been developed to satisfy the requirements of these projects.

Specifically, Accent was built to conform to the following design constraints which were viewed as essential to the success of large distributed systems such as Spice and DSN:

- *Modular decomposition.* It must be possible to decompose large problems into smaller modular units which can be run concurrently on a single processor or optionally distributed between several processors on a network.

- *Multiple language support.* The system should support not one but many language environments and provide tools for close interaction between languages.

- *Protection.* Within both the Spice and DSN environments, a single machine may be running a number of processes written by different individuals at different times. In Spice, for example, such programs as mail service daemons, games, editors, compilers and debuggers may all be running simultaneously to satisfy the desires of a particular user. If interactions between these processes were unpredictable, chaos could result. Blame for errors could not properly be assigned without knowledge of all the programs running at the time of an error.

- *Rapid error detection and tools for transparent fault-recovery, debugging and monitoring.* Multiple process debugging and monitoring, both on a single processor and over a network connection, are essential to building a reliable distributed system. Tools for rapid error detection and fault recovery are also necessary if the system is to be usable and maintainable.

- *Transparent network access.* The location of resources in the distributed system has to be transparent. Without transparent access, processes could not be transported from one machine to another and the decomposition of a task across machine boundaries would be impossible.

- *Uniform access to resources.* It should be possible for any feature provided by the kernel to be provided instead by a process. This includes functions normally thought to be exclusively the preserve of an operating system, such as virtual memory management.

- *Explicit representation of knowledge.* Where possible, information about the functioning of processes is made explicit rather than procedurally embedded. This allows greater latitude for optimization and error detection.

---

[1] Accent is a registered trademark of Accent International, Inc. The product it designates is sold as a spice and its only ingredient is monosodium glutamate (MSG).

Accent satisfies these constraints by:

- providing the ability to create and control a large number of independent processes on a single processor and supporting a sophisticated form of inter-process communication;
- supporting the notion of multiple, independent virtual address spaces and a virtual machine specification which can accommodate diverse interpretations of process state;
- supplying two kinds of protection: 1) address space protection, to insure that no process can affect another except through the use of the inter-process communication facility, and 2) access protection in the communication facility itself to prevent unauthorized communication between processes;
- defining inter-process communication in a way that allows transparent debugging, monitoring and fault recovery;
- taking advantage of these same mechanisms to allow transparent network extension independent of network hardware or protocols;
- allowing all services except the basic communication primitives to be viewed by processes as being provided through a communication interface; and
- structuring message communication to allow intermediary processes such as debuggers, protocol converters or network communication servers to better interpret the contents and purpose of messages.

Accent stands out as a relatively pure example of a *communication oriented* operating system -- i.e. an operating system which uses the abstraction of communication between processes as its basic organizing principle. The integration of virtual memory support, file access and inter-process communication in Accent makes possible significant performance improvements over previous communication oriented systems as well as a more "transparent" network operating system design.

2. System overview

Physically, the distributed system built around Accent can be viewed as a loosely-connected collection of *host machines*, each with its own primary and secondary storage. The notion of a host machine is flexible in that it can include either uni-processors or tightly coupled multi-processors. In addition, no specific assumptions are made about the technology or topology of the underlying communication network which links together these host machines. The initial host machine chosen for Spice/DSN, the Three Rivers Corporation PERQ, is a uni-processor which currently possesses a 3MHz Ethernet[5] and will eventually possess a 10MHz Ethernet.

Each host machine on the network possesses an *operating system kernel* which in turn supports a

collection of *processes* (See Figure 2-1.)

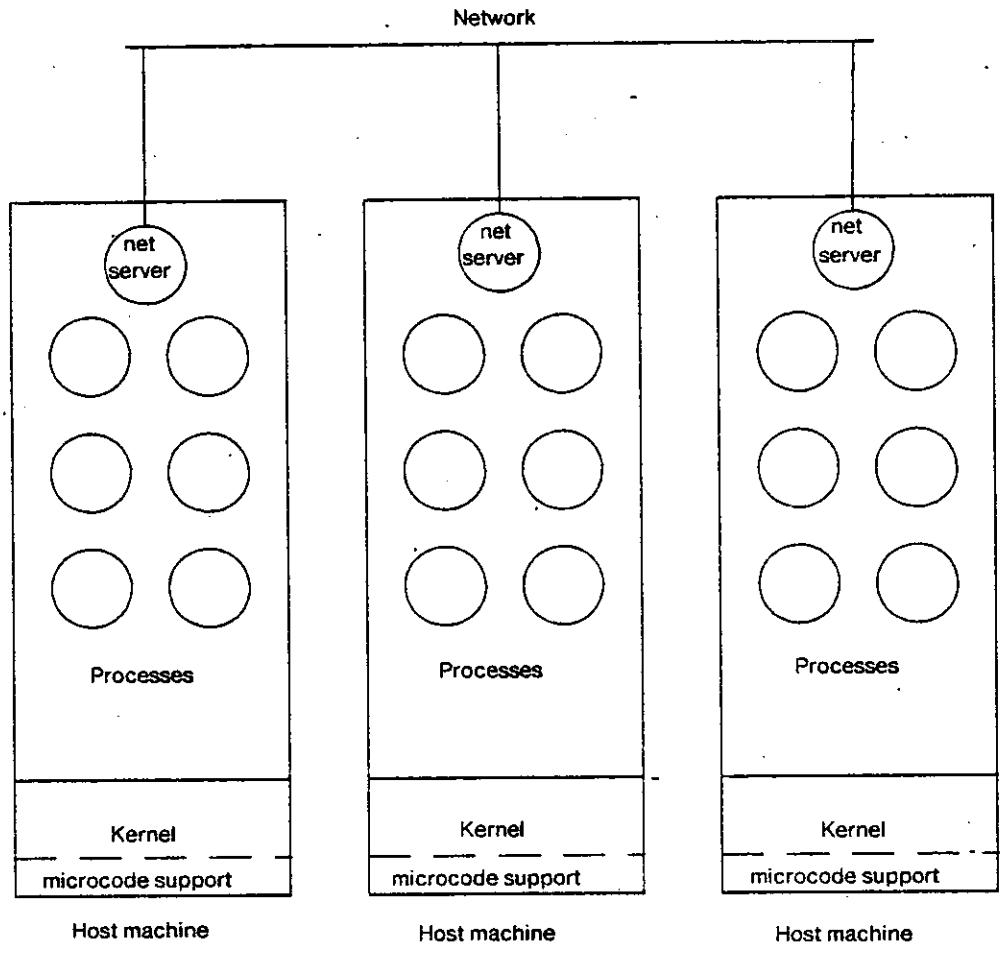

Figure 2-1: Overview of system

The function of the system kernel is to provide an execution environment for processes running on its host machine. This includes

- *inter-process communication,*
- *virtual memory management, and*
- *process management.*

In addition, the kernel provides

- *the low-level functions of process creation and destruction,*
- *access to devices through inter-process communication,*
- *support for language and application specific microcode, and*
- *rudimentary support for process monitoring and debugging.*

These kernel functions are similar to those provided by the RIG operating system[1,2] and satisfy the requirements for a distributed operating system kernel suggested by Watson[4].

The system as a whole can be viewed as having a number of layers, with the system kernel at the bottom and layers of processes providing successively more complex services building upon each other. Inter-process communication (IPC) is the glue which binds processes together. It provides a uniform interface at each level of the system. All objects and services in the system (including those provided by the kernel) are accessible through IPC. Even though the IPC facility of the Accent kernel is defined solely in terms of communication between processes on the same machine, communication can be extended transparently over a network by processes called *network servers*.

3. Inter-process communication

Although processes are the active components of the system, the notion of process is a poor one on which to base a communication facility. Different languages may define multiple concurrent tasks within a single kernel-supported process. Moreover, a given service may be provided over a period of time by a number of different processes.

The basic transport abstraction of the IPC is the notion of a *port*. A port is a protected kernel object into which messages may be placed by processes and from which messages may be removed. All services and facilities provided by a process are made available to other processes through one or more ports.

Ports are intended to be used by processes to represent specific services or data structures. For example, a file system process might associate a separate port with each open file, or a virtual terminal handling process might allocate a port to represent each virtual terminal. However, no restriction is placed on their use. Ports may be used by processes to communicate information in any mutually convenient way.

Logically associated with each port is a queue on which reside messages sent to that port but not yet removed from it by a process. The ability to remove messages from a port is called *receive access*. Only one process may have receive access to a port at a time, although receive access to a port may be transferred to another process in a message.

Ports cannot be directly manipulated or named by a process. Instead, the kernel provides processes with a secure *capability* to send a message to a port and/or extract (receive) a message from it. This capability is a local name for a system object, much in the way a UNIX file descriptor is a local name for a system maintained file, pipe, or device[9]. Port capabilities may be passed in messages, handed down to process children, or destroyed. A given process may have only one local name for a given port at a time. Whenever a port name is passed in a message the system kernel must map that name from the local name space of the sending process into the name space of the receiving process.

The ability to manipulate access to ports allows for the redirection of communication from one process to another and the explicit management of communication between two processes by a third process. It also allows a port to be used to refer to a specific service or process-provided object even in situations in which that service or object is handled by different servers at different times. Neither the address (i.e. the location) or the name of a process can be determined from a capability to send a message to one of its ports. See Figure 3-1.

3.1. Ports

3.1.1. Creating, accessing and destroying ports

A port is created by a process through an *AllocatePort* system call. It is a system object, distinct from the process which created it, but initially *owned* by the creating process. The result of the *AllocatePort* call is a local port name which refers to the created port. This name is logically an index into a correspondence table maintained by the kernel for each process. It has meaning only when used by that process.

Initially the owner of a port also has receive access to it. Ownership of a port and receive access are, however, logically distinct. Ownership of a port may be passed in a message from one process to another, but not shared. Receive access to a port may also be passed in a message but not shared. These restrictions prevent multiple servers from managing the same queue, and are necessary to avoid serious problems which occur when access to a single queue is shared by processes on different machines.

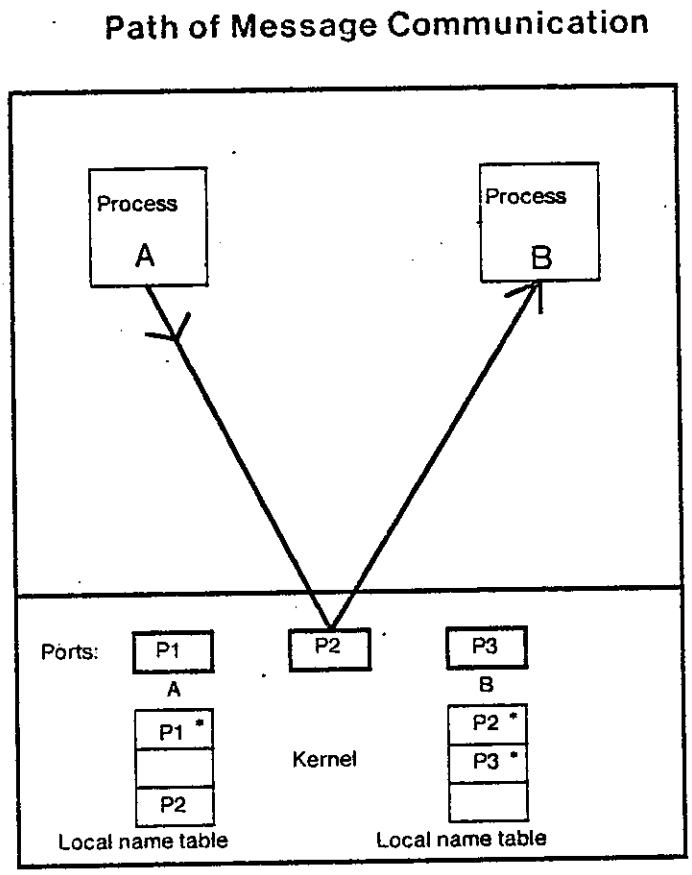

Figure 3-1: Path of message communication

If a single process both owns and has receive access to a port, that process may destroy the port by performing a *DeallocatePort* system call. A process with a capability (local name) to a port may release the capability via the same system call.

A port is automatically destroyed when its owner and the process with receive access to it both die. In either case, all processes with access to that port are notified via emergency messages. If the same process does not both own and have receive access to a port, then the deallocation of the port name by either process results in an emergency error message being sent to the other accompanied by full access rights to that port (i.e., both ownership and receive rights).

The purpose of distinguishing receive access from ownership is to allow a process to take over

services or functions provided by other processes in the event those processes should die or malfunction. This is particularly important when writing fail-soft software and can also be used to provide orderly shut-down of services after catastrophic failures.

3.1.2. Software or pseudo interrupts

In the case where a process does not wish to wait for messages explicitly, it is possible to enable a software interrupt which will be triggered upon message reception. The interrupt service routine, executing in the context of that process, can then receive the incoming message and process it or notify the main program of the event in a user defined manner. A mechanism such as this can allow processes which are inherently compute bound to react quickly to incoming messages without using some form of message polling.

3.1.3. Flow control

The message queues attached to ports have a finite length. This prevents a sending process from queueing more messages to a receiving process than can be absorbed by the system and provides a means for controlling the flow of data between processes of mismatched processing speed.

Subject to implementation restrictions on maximum port size, the process owning a port is allowed to specify its *backlog* -- the maximum number of normal messages which may be queued for that port at one time. Should a process attempt to send a message to a full port, one of three things may happen depending on options specified by the sender:

1. The process is suspended until the message can be placed in the queue.

2. The process is notified of an error condition (after perhaps allowing itself to be suspended for up to a specific period of time waiting for the message to be sent).

3. The message is accepted and the kernel sends a message to the sending process when that message can actually be placed in the queue. A maximum of one message per sending process per receiving port may be outstanding in this fashion.

These three options correspond to three different programming situations:

1. The first option is the one most likely to be used by a 'user process' when communicating with a 'server process'. In this situation the user process does not care if it is suspended for some time waiting for a message to be delivered to the server (as in the case of a remote procedure call).

2. The second option is used when a process does not care whether a particular message is sent to a destination, but is using the message only to wake up a dormant partner. The fact that other messages are in the queue for the partner's port indicates that the partner is already scheduled to be activated.

3. The third option is the one most likely to be used by a service process when dealing with a user process. The server probably cannot afford to be suspended waiting on a user to clear its queue. It may also not want to just throw away the message or poll the user explicitly until the message can be sent. The system provides an explicit message event corresponding to the unblocking of the user's port queue.

The flow control scheme described here is not the only reasonable technique for restricting the flow of messages between processes. Other possible mechanisms include limiting the total number of messages which may be outstanding from a given process (as in Brinch Hansen's RC4000 system or CMU's Hydra) or limiting the outstanding messages from a given process to a particular port. Experience with RIG, however, has shown that the exact flow control policy is much less important to overall system performance than the higher level communication protocols devised to solve individual problems. Even more important to the functioning of the system is the provision for events to indicate when ports become unblocked. This prevents needless and expensive polling in server processes.

3.1.4. Waiting for messages

Although particular protocols may specify synchronous behavior, the receipt of a message is inherently an asynchronous event. In a network environment in particular it becomes possible for error conditions to cause messages to be sent to a process at almost any time. Mechanisms must therefore be provided for a process to check the state of its ports, to wait for activity on one or more of its ports, and to receive messages selectively. It should also be possible for a process to specify a maximum amount of time to wait for a message before reawakening.

Four basic primitives are provided for accomplishing this task. *Receive(SetOfPorts,Message,Timeout)* waits for at most *Timeout* milliseconds and if a message is available during that time from any of the ports designated by *SetOfPorts* then the message is read into *Message* and a boolean value of true is returned. *MessageWait(SetOfPorts,Timeout)* performs a similar function, but does not receive the pending message. It returns the capability of the port from which the next message would be received. *Preview(SetOfPorts,Message,Timeout)* is again similar to *Receive*, but it reads only the header of the next waiting message and does not actually dequeue that message from the port queue. *PortsWithMessagesWaiting(SetOfPorts)* is an informational routine which checks the status of all ports and returns the set of ports with messages waiting.

3.1.5. Sending and receiving kernel messages

Each newly created process has access to two ports whose primary purpose is to allow messages to be sent to and received from the Accent kernel. The first is called the *kernel port* of the process and the kernel logically has the receive rights for this port while the created process has the send

rights. The second is called the *data port* and it is normally used by a process to receive messages from the kernel.

The father of a process can, at the time of process creation, ask that other ports to which it has access be given to its child process. The father can also get access to the kernel and data port of the child process. This, taken together with the ability to send port access rights in messages, is the basic mechanism for establishing communication between processes.

3.2. Messages

A message is logically a collection of typed data objects copied from the address space of the sender at the time of a message send call and into the address space of the receiver when a receive call is performed. Physically, a message is divided into two parts: 1) the message header -- which contains information normally associated with all messages and 2) an optional description of structured data to be sent. The purpose of this division is primarily to optimize the transmission of short control messages and to make it easier for the kernel to find critical information which must always be contained in a message -- such as its destination port.

3.2.1. The message header

The message header contains a small amount of system required message information and is used to form an anchor for the structured part of a message. At minimum it specifies the type of a message and contains a capability for the destination port of a message and a field for a capability (which may be empty) to be used for a reply. The destination and reply ports are more commonly referred to as the *remote* and *local* ports, respectively. The header also contains an ID field to be used for discriminating messages and a pointer to the structured data part of the message.

The header may be checked without actually receiving the data portion of a message by calling *Preview*. The purpose of this facility is to allow a process to check the ID of the message received and select an appropriate message structure for receiving it.

The type of a message contains information which determines the kind of service it requires of the IPC. The following service classes are provided:

- *Flow control*. Normally messages are flow controlled according to the above procedure. The message type can specify, however, that a larger flow control backlog should apply to this message. This allows special high priority messages to be sent to otherwise full ports.

- *Priority*. A range of message priorities is provided. Messages waiting at the same port

with different priorities will be received according to the order of their priorities.

- *Sequentiality*. Messages from the same process to a particular port with the same priority are normally guaranteed to arrive in the order they were sent (FIFO). It is possible to relax this constraint by noting in the message type that the order of reception of a particular message is unimportant.

- *Reliability*. A message is guaranteed to arrive at its destination reliably, as long as the destination exists long enough for the message to be received. A message could simply be advisory, however, and it may be unimportant that it arrive at its destination. A message may therefore contain in its message type an indication that it may be delivered unreliably.

- *Maximum age*. A message could become out of date after a certain length of time has passed. A message may therefore be marked as having a maximum age after which it may be destroyed rather than delivered.

- *Security*. A message may be marked as requiring special procedures for ensuring the securify of its data. Messages containing password information would fall into this class. Messages marked as 'secure' are encrypted when transmitted on a network or placed on a storage medium which may not be secure.

Two message types have been given special names and play a dominate role in communication:

1. *Normal messages*. A 'normal' message is flow controlled, sequential, reliable, not secure, of lowest priority and has no maximum age. This is the default message type and is assumed to satisfy most communication requirements.

2. *Emergency messages*. Emergency messages are specially flow controlled, sequential, reliable, not secure, of highest priority, and have no maximum age. Emergency messages play an important role in error handling. Because of their high priority, they are guaranteed to be received before any normal messages sent to a port. Their purpose is to allow urgent information to be delivered to a process regardless of that process' current message backlog or message queue. They are used for error notification, special event processing, and debugging.

The notion of message type allows the programmer the ability to specify the exact requirements of his IPC use. This gives the underlying system more information about a message and thus makes possible optimizations in the delivery and management of messages. This use of message types is consistent with the overall goal of making as much of the inner workings of the communication system as possible *visible* to the 'outside world' rather than hidden inside compiled algorithms, thus allowing greater flexibility in optimization, management and monitoring.

The service classes associated with sequentiality, reliability, maximum age, and security are considered advisory. The system may choose, for example, to deliver a message reliably which has

been marked as unreliable without affecting the correctness of the programs which use unreliable messages. For the most part they have a direct effect only on the management of message traffic by network servers and other intermediary processes. Priority and flow control, however, are important to program correctness and cannot be changed without possibly introducing deadlocks.

3.2.2. Message structure

The structured data part of a message is described in detail in[7]. Currently any data structure which contains no self references can be described. A message format which allows an arbitrary graph to be transmitted is now being designed for the DSN project. The purpose of message structure is to provide a standard protocol for communicating typed information. Among the important kinds of typed data which can be passed are capabilities for ports. Only port capabilities are currently checked by the system kernel for correctness. All other data is passed unchecked by the kernel, but may be interpreted, if necessary, by intermediary processes such as network servers or debuggers.

3.3. Exceptional condition handling

Distributed programming imposes a heavy responsibility to handle a multitude of error conditions. Message activity can be pipelined or multiplexed, and the relationships between incoming and outgoing messages can be much richer than in a conventional programming environment (i.e., one in which subroutines are used as the primary structuring mechanism). A faulty process could conceivably crash many processes by sending illegal messages, making it very hard to identify the source of the problem. As a practical matter, it is difficult to ensure complete compatibility between similar programs written by different individuals in different languages. Supposedly interchangeable processes may differ in subtle ways. A failure in a message-based system can quickly lead to finger-pointing.

A variety of facilities for protection, error detection and error handling have been built into the IPC. Illegal process addresses are noted within the send and receive primitives, and the appropriate error returned to the offending process. In addition, processes are protected from accidental or malicious access through the use of port capabilities rather than global port or process identifiers. Whenever a port is destroyed the kernel notifies all processes which still have access to that port via an emergency message. Emergency messages themselves are important because the give processes a a way to reliably communicate errors in situations where normal communication channels are blocked. Software interrupts allow processes to handle error conditions without interfering with normal execution.

3.4. Network inter-process communication

3.4.1. The notion of an intermediary process

The fact that messages are sent to ports rather than processes and that ports are only referred to indirectly by processes allows a single process N to act as an intermediary for communication between two distinct process groups, A and B. Such an intermediary process allocates ports which 'mirror' only those ports used by each of the two process groups for communication with the other. When a message is sent between the two groups it arrives at one of these 'alias' ports and is forwarded to its appropriate final destination. Processes in group A believe that the intermediary ports of N are in fact owned by processes in group B, since all messages sent to those ports have the effect of being delivered to ports which are in fact owned by group B processes. The same is true for processes in group B. Because messages are strongly typed, capabilities for ports can be passed in messages in the same way that they are passed by the kernel, with N substituting the names of its 'alias' ports for those of the corresponding 'real' ports of A and B. Whenever a new, previously unseen, port of A is sent to B in a message, N creates a new local port to correspond to it and passes that onto B in the forwarded message. It is in this way that new communication connections are made between A and B through N. Of course, N must provide some means by which A and B establish an initial connection. This could be done either through string name lookup or through some special function provided by N and known to A and B. See Figure 3-2.

3.4.2. The network server as an intermediary process

A network server which provides a transparent network extension of the IPC can function much like the intermediary process described in the previous section. In the case of inter-machine communication, the process groups A and B are the processes of two host machines X and Y respectively. Two network servers are required, N_1 and N_2 which communicate with each other across a network to provide the correspondence between 'alias' ports. The kernel requires no knowledge of networking or networks for such a mechanism to be provided. See Figure 3-3.

The primary requirement of a network server is that it provide some form of reliable, flow controlled communication between machines which can accommodate the semantics of local IPC communication. A communication path for messages to a port on another machine on the network is identified with a port which belongs to the network server and to which processes may send messages to be forwarded across the network. As in the case of the intermediary process, a process A on machine X can possess a capability for a port to which messages can be sent destined for a process B on machine Y. This capability is in fact a capability for a port owned by a network server N_1 on X, but this is both unknowable and unimportant to A. The exact nature of the connection, the

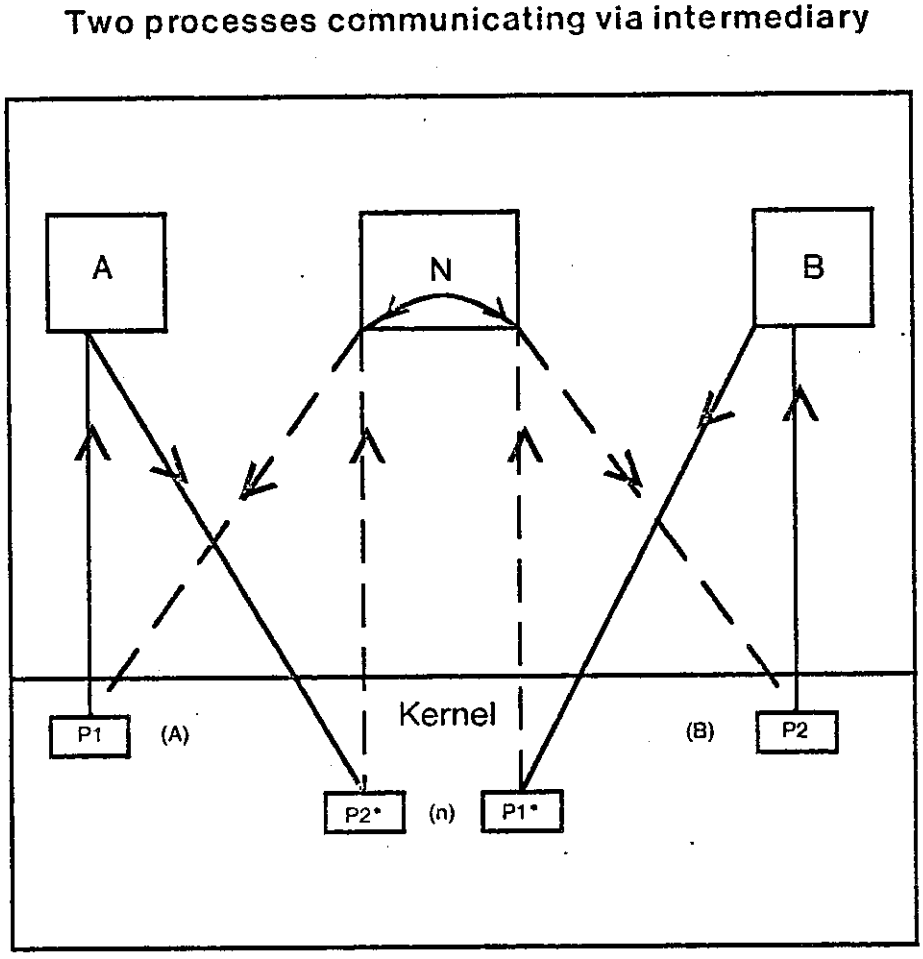

Figure 3-2: Communication via an intermediary process

protocols used, the topology of the network, and even the existence of multiple competing networks linking X and Y can be hidden from process A.

For this reason, there is no requirement that there be only a single network server or that all network servers use the same communication protocols or even that 'network servers' actually implement communication on a physical communication link. This allows multiple competing versions of network software to be written and debugged or multiple network links to be provided without affecting the system kernel or existing software.

Where efficiency is critical, the fact that the kernel can act like a process allows the implementation

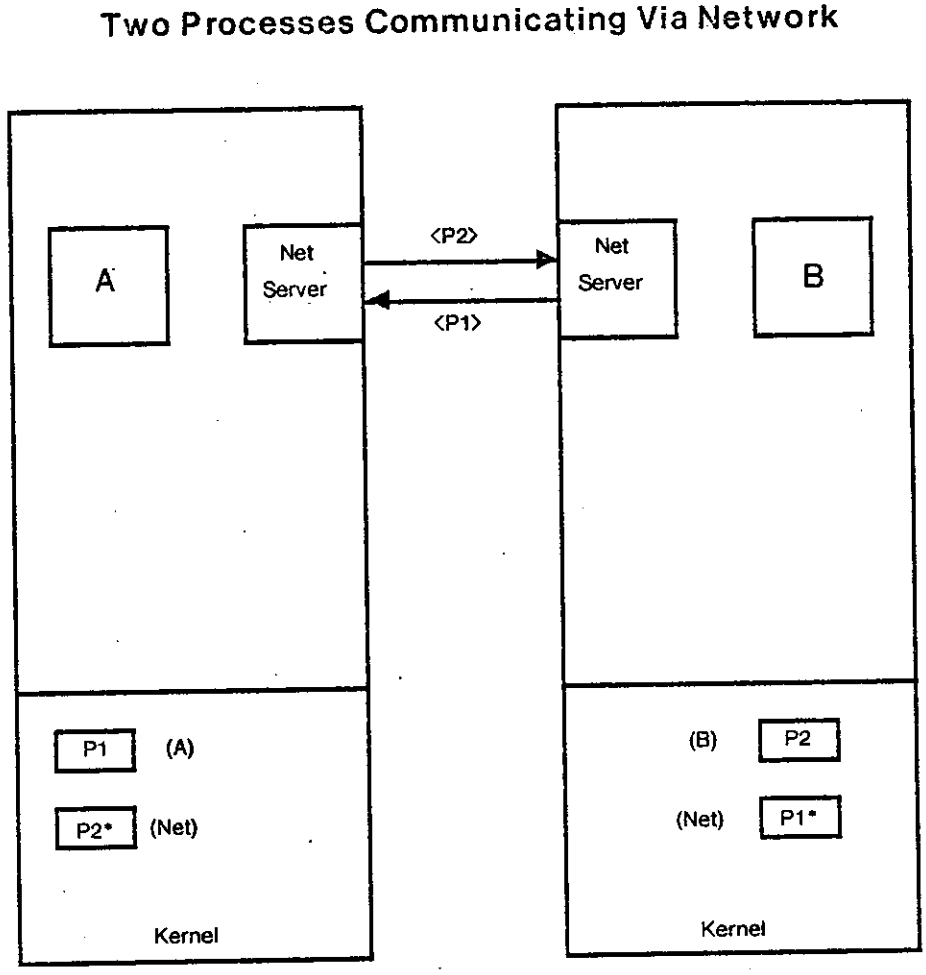

Figure 3-3: Network communication

of the network server in the Accent kernel.

3.4.3. Port migration

Just as a network server can fabricate a local port to correspond to a remote port, it is also possible for a port to be 'passed' between processes through a network server. One requirement of this scheme is that there be a way for network servers to know when a message contains a port capability. Along with each such capability sent in a message goes an implicit network connection which would be needed should a message be sent to that port. It is the responsibility of the network server to convert any references to capabilities found in messages forwarded over a network into capabilities local to its host machine.

3.4.4. The use of message types

The concept of message type is important to the efficient functioning of a network server. Message service classes encoded into the message type, such as *unreliable*, *maximum age*, and *security* can be directly translated into actions which can be taken by the network server to speed service or make communication more secure. For example, a distributed sensor net with real time constraints could mark messages containing audio data as having a specific lifetime or as not requiring reliable transmission. If such messages arrive too late they could have no use to the tracking processes. Also, redundancy in the handling of audio data for tracking purposes could make retransmission of error packets unnecessary. At the same time a network server may choose to ignore such advisory class distinctions in the name of simplicity or debugging (where reproducibility is more important than speed).

3.4.5. The use of structured messages

Message structuring is important not only for handling the transmission of port capabilities across networks but also for data conversion in networks with machines which have widely varying data representations for integers, reals, characters, etc. As stated above, it would be appropriate in such networks to reserve certain 'user types' which would have network understood meanings. In this way network servers could provide for data conversion across machine boundaries. Specifically, process A on machine X is communicating with process B on machine Y. A sends B a message containing arbitrary data. The kernel on X packages the data into a host-dependent linearized form and delivers it to the network server, which then delivers the data across the network in a manner which Y can understand. The network server on Y unpacks the data into the host-dependent linearized form for Y and delivers the message to its kernel. If word-size mismatches occur, B can specify in its receive that the data should either be truncated or expanded.

By having this translation performed by the network server rather than the end processes, greater location and implementation independence are obtained and the knowledge of which machines are actually in communication can be hidden by the network. In addition, the translation of data need occur only once rather than twice.

3.4.6. Issues in process migration

The definition of the IPC makes it possible for a process B to migrate from one machine to another on the network without a cooperating process A being aware of the change as long as the network servers involved handle the transfer of the logical communication channel between A and B. Although the transfer of all ports associated with a process and their corresponding message queues is possible using the IPC, process migration remains a difficult problem due to issues about

the definition of process state. The problem of process migration is made especially acute by the fact that the most common reason for migration is probably hardware failure. In such a case the recreated state of the migrated process corresponds to the last 'good' or checkpointed process state and some message communication may have gone on since it was saved. The issue of process migration for fault-tolerance is therefore not only a system problem but also one of properly structuring message communication into atomic transactions.

4. Virtual memory management

In Accent, virtual memory, file storage and IPC are integrated together in a way that preserves the logical structure of inter-process communication while providing significant performance advantages over previous communication based operating systems. This same melding of virtual memory with IPC makes it possible for Accent to allow one process to manage the virtual address space of another (either by allocating virtual memory from the kernel and sending it to another process or by explicitly managing page faults) and in so doing provides a clean, kernel-transparent mechanism for cross-network paging.

4.1. Virtual memory and files

The virtual address space of an Accent process is flat and linearly addressable. On the PERQ this address space is 2**32 sixteen bit words. An *Accent segment* is the basic unit of virtual memory allocation and secondary storage management. All randomly accessible secondary storage is considered part of the virtual memory of the system and is organized into Accent segments and managed by the kernel.

There are two kinds of Accent segments: *temporary* and *permanent*. Temporary segments are allocated by processes as required for their memory needs and are released when all processes which have access to them are terminated. The storage contained in permanent segments form the basis for the Accent file system. Permanent segments are allocated by sending messages to a special port normally supplied only to special processes. They do not disappear except by explicit request.

Normally, new segments are allocated by the kernel in response to a *CreateSegment* message, with the kernel responding in a message with the newly created segment's identifier. A segment can be explicitly destroyed through the use of a *DestroySegment* message. The data contained in a segment can be read into a processes virtual address space using a *ReadSegment* message. The reply to a ReadSegment message contains the newly allocated pages which are introduced into the requesting process' address space through reception of the message. Similarly, data can be explicitly

transferred out of a process' address space through the use of a *WriteSegment* message.

4.2. Optimizing message transfers

Messages are logically copied from a process' address space into a kernel message data structure upon transmission and are logically copied from the kernel to a process' address space upon message reception. Since data is never shared between processes, message communication over a network has precisely the same semantics as local message communication.

Double copy semantics need not, however, imply actually copying the data twice. If a process sends a message pointing to pages in its virtual memory and either releases the memory or simply never changes it, double copy semantics can be preserved by marking the pages referenced in the message as copy-on-write and not actually copying them into the supervisor's address space. Moreover, if the receiver of the message doesn't care about its placement or desires that it be placed in its memory on the same page boundaries no copy of data need be performed at all. Instead, the data pages may be placed directly into the address map of the receiving process.

Thus, when a process sends a message to a port, whole virtual memory pages referenced in the message are not copied but instead marked as copy-on-write in the address space of the sender. Should the sender attempt to change any or all of these pages, new virtual pages will be allocated, filled with the appropriate data, and placed in its address space. On reception into an area of the receiver's virtual memory which is properly aligned, pages referred to in the incoming message are mapped in rather than copied. If the receive desires a different alignment of data than that specified by the sender, a copy operation will be performed.

The utilization of virtual memory by the IPC represents a functionally transparent *optimization* which is not required either for the functioning of the IPC or the virtual memory management facility. Nevertheless, this optimization can be of enormous value in increasing the speed of communication between processes on the same host.

4.3. Optimizing message transfers: an example

The advantage of integrating virtual memory, file storage and IPC is graphically illustrated by the example of file system access provided through messages sent to and from a file system process. The file system process can read secondary storage by sending a message to the kernel. The kernel then sends back a message which contain the virtual pages requested by the file system. This message need look no different than any other message containing data. Moreover the requested

pages need never be copied, but are simply placed into the address space of the file system in the normal way that data is placed into the address space of any process when received in a message. If a user requests a block of data, however large, the file system can send it that block in a message, again without ever having the underlying data referenced by either the kernel or the file system. In this way the speed advantages of large PMAPed[3] files can be obtained through the reception of normal IPC messages without resorting to special file mapping primitives.

4.4. Process directed management of virtual memory and network paging

Another advantage of the Accent approach to virtual memory is that it allows virtual memory to be considered a process provided resource. A process can create a *pseudo segment* and "read" it into an unused part of its address space. It is then possible for a process to send another process these 'pseudo pages' in a message. When the message is received, these pages are placed into the address map of the receiving process. When these pages are referenced by the receiving process, messages requesting their contents are sent to a port belonging to the process which created the segment When a changed 'pseudo page' is about to be purged from main memory, its new contents are also sent in a message. This allows a process which is not the kernel to provide kernel-like management of secondary storage. In particular, it makes cross-network paging possible using the standard IPC facility.

5. Process Management

The process management system forms the third major part of the operating system kernel. It interacts with the virtual memory management and inter-process communication systems to provide the execution environment for processes running on a host machine.

As mentioned earlier, all communication between a user process and the kernel is through a port, called its *kernel port*, which is created when the process is created. Since ports can be sent in messages to other processes, it is possible for process A to send its kernel port to process B. The process system is designed so that process B can manage process A's behavior, much the same way the virtual memory system allows one process to manage another's virtual memory. This mechanism forms the basis for remote debugging and monitoring systems.

A number of issues were addressed in building the process management component of Accent:

- *Simple user interface*. From the user's point of view, the process system is quite simple. The basic primitives allow process creation and destruction with *Fork* and *Terminate*, process monitoring with *Status*, and process control with *SetPriority*, *SetLimit* (which sets

a runtime time limit), *Suspend*, and *Resume*.

- *Multiple language support.* Mechanisms are provided for supporting languages with different notions of process state. In particular, different languages will have different underlying microcode support. In order to support multiple languages at this level, most of the process mechanisms are forced down to the microcode level. A *MicroKernel* has been defined to provide this support. The MicroKernel consists of all the language independent microcode support, including process queue management, process state switch and context swap, low level scheduling, I/O support, interrupt support, and virtual address translation. The MicroKernel is defined so that language microcode interfaces to it in a small number of well-defined ways (e.g., the number of registers to include in context swap is language dependent and is stored in a fixed register).

- *High performance.* The context swap mechanism is language independent, yet quite fast. Since the virtual memory system requires no change of page map on context swap, only language dependent microcode registers need to be swapped. Our current measurements indicate that context swap will be well under 100 microseconds.

- *Scheduling control.* The process scheduling system provides time slice scheduling with sixteen priority levels. Preemption is provided to enable time critical processes to meet their demands. Aging is provided to support some degree of fairness in scheduling.

- *Simple interrupt structure.* The traditional interrupt structure is not being supported in this system. Hardware interrupts are serviced in the MicroKernel, and processes waiting on service are simply awakened. Preemptive scheduling is used if the awakened process is time critical. At the process level, there is no notion of hardware interrupt. However, there is a notion of software interrupt, which is tied closely to the inter-process communication system. Any receipt of a normal or emergency message will flag a software interrupt for the receiving process. Since different languages will represent software interrupts in different ways, it is left to the language microcode support to interpret the interrupt flag (on resumption from a context swap) and reflect it to the user in whatever way is appropriate for that language.

6. Current status of Accent

An implementation of the Accent IPC facility as a communication facility for VAX/UNIX has been in use since March, 1980. Network servers written in PASCAL have also been implemented and are now in use. The PERQ implementation of the full Accent kernel is now (April 1980) in its final stages.

7. Final thoughts

One of the first (and suprisingly most controversial) decisions made in the design of Accent was the determination that every kernel supported process should have a logically distinct and independent address space. The reasons for this decision were:

- Not all languages are "safe" in the sense that a properly compiled and loaded program can be expected to respect the integrity of storage which is not its own. Examples of

unsafe languages include BCPL, C, and PASCAL. Even languages which are often thought of as safe have embarrassing loopholes which make address space protection important.

- It is usually easier to construct a runtime environment for a language if it does not have to be integrated into the same address space as other language environments. Moreover, decisions about the use of an address space by a language, once made, need not be reviewed every time a new language gets added to the system. This is simply a modularity argument, but it can be important in practice.

- Both DSN and Spice have the requirement that a single processor may need to support a large number of simultaneous processes, many of them written by different individuals with varying levels of programming expertise. It is important for the isolation of problems that errors generated in a given subsystem not be allowed to propagate at random through other subsystems. Address space protection provides an important tool for controlling inter-program error propagation.

- It is very difficult to provide truly transparent networking if processes can potentially have interactions which the system itself cannot detect.

This decision to support multiple independent address spaces on a single personal computer should be contrasted with the approach being taken by others currently doing similar research. The XEROX Pilot operating system[8], for example, provides a single (currently 24 bit) address space for all concurrent processes it supports. It can do this because it supports only one programming language (MESA) which has been designed to reduce the probability of inter-program contamination.

One significant disadvantage of the Accent approach is that it is often time-consuming for existing computers to switch from one logical address space to another. One reason for Pilot's single address space design is that its target machine has a single large set of virtual address mapping registers which would have to be updated in order to switch contexts. As another example, the VAX 11/780 uses a content addressable address translation buffer to provide both a larger address space and faster context switching. Unfortunately the buffer must be flushed on a context switch and refilling the buffer for the new process results in switching times on the order of hundreds of microseconds. Some machines, however, do provide acceptable address space switching times. The PRIME 750, for example, can switch address spaces in less than ten microseconds[6]. The key to the PRIME approach is the use of the process identifier as part of the virtual address. In the PERQ system we have adopted a similar scheme with the result that context switching times are comparable to two PERQ procedure calls.

The decision to build a communications based system on the notion of protected ports and port capabilities was based on considerable past experience with message based operating systems both

at CMU and the University of Rochester. Ports simplify the problems of network communication and protection by decoupling networking from intra-machine communication and providing an abstraction of function which does not depend upon a particular implementation of processes. Ports can be used to represent functions independent of how those functions are implemented. They can be used to specify kernel objects (such as files or process management functions) or objects maintained by processes. All access to the outside world is mediated through ports with the notion of system call reduced to the notion of sending and receiving messages.

Protection derives from the fact that ports are protected kernel objects to which processes only have a local reference. The right to send a message to a port cannot be forged or accidentally created. This prevents processes which are either buggy or malevolent from gaining fraudulent access to resources and it allows the writer of a process to make a positive statement about the correctness of his program based on precise knowledge of which other processes are allowed to communicate with it.

The existence of access lists also provides the kernel with complete knowledge of the communication rights of processes. It is not possible for a process to hide a reference to another process' port in its private address space. The system can know who is talking with whom and notify processes which have access to a port when that port is destroyed. Knowledge of inter-process communication paths can also be used as an aid in intelligently scheduling systems with tightly coupled communicating processes.

Finally, the fact that the location of a port can change at will as can the nature of the process serving it, without affecting the processes using the port allows transparent process migration and fail-soft performance.

References

1. 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. on Software Eng. 2*, 4 (December 1976), 321-328.

2. J. E. Ball, E. Burke, I. Gertner, K. A. Lantz, and R. F. Rashid. Perspectives on message-based distributed computing. Proceedings 1979 Networking Symposium, IEEE, December, 1979, pp. .

3. Bobrow, D.G., Burchfiel, J.D., Murphy, D.L. and Tomlinson, R.S. "TENEX, a paged time sharing system for the PDP-10." *Comm. ACM 15*, 3 (March 1972), 135-143.

4. Lampson, Butler, ed.. *Distributed Systems Architecture and Implementation: An Advanced Course*. Springer-Verlag, 1981.

5. Metcalf, Robert and Boggs, David. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Comm. ACM 19*, 7 (July 1976), 395-404.

6. ----. Prime Reference Guide: System architecture and instructions. Tech. Rept. IDR3060, Prime Computer, Inc., July, 1978.

7. R. Rashid. Accent: A network operating system for SPICE/DSN. Tech. Rept. , Computer Science Department, Carnegie-Mellon University, May, 1981.

8. Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G. and Purcell, S.C. Pilot: An operating system for a personal computer. Proceedings of the 7th Symposium on Operating Systems Principles, December, 1979, pp. .

9. Ritchie, D.M. and Thompson, K. "The UNIX time-sharing system." *Bell System Technical Journal* (July 1978).