A Survey of Process Migration Mechanisms

Jonathan M. Smith

Computer Science Department
Columbia University
New York, NY 10027

Technical Report CUCS-324-88

ABSTRACT

We define process migration as the transfer of a sufficient amount of a process's state from one machine to another for the process to execute on the target machine.

This paper surveys proposed and implemented mechanisms for process migration. We pay particular attention to the designer's goals, such as performance, load-balancing, and reliability. The effect of operating system design upon the ease of implementation is discussed in some detail; we conclude that message-passing systems simplify designs for migration.
A Survey of Process Migration Mechanisms

Jonathan M. Smith

Computer Science Department
Columbia University
New York, NY 10027

Technical Report CUCS-324-88

1. Introduction

An image is a description of a computation which can be executed by a computer. A process is an image in some state of execution. At any given time, the state of the process can be represented as two components: the initial state (the image) and the changes which have occurred due to execution. We note that the complete state of a computation may include information which is inaccessible, for example data kept in tables internal to the operating system.

Process migration is the transfer of some (significant) subset of this information to another location, so that an ongoing computation can be correctly continued. This is illustrated in Figure 1:

![Figure 1: Process Migration](image)

The flow of execution of the process which is being transferred is illustrated in Figure 2:

![Figure 2: Flow of Execution of Migrating Process](image)

Note that we have illustrated the case where the process does not begin execution on the destination host until all of its state is transferred; in fact, it is possible for the "migrating" process to begin execution on the destination almost immediately. This could be done by transferring relevant registers and setting up an address space; the remainder of the state would be sent later or demand-paged as needed. The advantage of complete state transfer is that the source machine can release resources immediately upon completion of the transfer.

We can define constraints on the computation so that the subset transferred is sufficient to provide correct computation with respect to the constraints.

If we transfer the state of a process from one machine to another, we have migrated the process. Process migration is most interesting in systems where the involved processors do not share main memory, as otherwise the state transfer is trivial. A typical environment where process migration is interesting is autonomous computers connected by a network.

Process migration is desirable in a distributed
system for several reasons, among them:

- Existing facilities may not provide sufficient power; process migration can provide a simple solution to the problems that exist as a result of the weakness in the facilities. For example, several existing implementations of distributed file systems do not provide transparent access to remote files; that is, the behavior of a remote file can be differentiated from the behavior of a local file - they have different semantics. Since the usual semantics are defined on the local processor, moving an active process to the relevant processor provides a mechanism by which semantically correct remote access (e.g., with exclusive-access "lock" mechanisms) can be accomplished.

- Long running processes may need to move in order to provide reliability, or perhaps more accurately, fault-tolerance, in the face of a certain class of faults about which advance notice can be achieved\(^1\). For example, the operating system may deliver to the process a notification that the system is about to shut down. In this case, a process which wants to continue should either migrate to another processor or ensure that it can be restarted at some later time on the current processor.

- Moving processes may serve as a tool for balancing the load across the processors in a distributed system. This load-balancing can have a great effect on the performance of a system. Eager, Lazowska, and Zahorjan [Eager1986a] indicate that there is an opportunity for performance increases. They derive asymptotes which define the window of opportunity for balancing, and they point out that the more communication necessary for the distributed system to perform the balancing, the worse the performance becomes. Several algorithms to performing the load balancing have been suggested, for example, the Drafting Algorithm of Ni [Ni1985a] and the algorithm used in the MOS system\(^2\), developed by Livny and Melman [Livny1982a]. Since a small subset of the processes running on a multiprocessing system often account for much of the load, a small amount of effort spent off-loading such processes may yield a big gain in performance. Empirical data gathered by Leland and Ott [Leland1986a], and Cabrera [Cabrera1986a] strongly support this.

- Whenever the process performs data reduction\(^3\) on some volume of data larger than the process's size, it may be advantageous to move the process to the data. For example, a distributed database system may be asked to perform a join operation on a remotely-located dataset of size N. If the join is expected to reduce the data volume by a significant amount and the database is sufficiently large, it may be advantageous to move the local process object to the remote file object rather than fetching the remote file object for processing by the local process object. Another example is a process which performs statistical analysis of a large volume of data. In the case where the process object is moved, our data transfer consists of size(process)+size(processed data); while the data transfer in the case of the remote file object being transferred to our site is size(remote file).

- The resource desired is not remotely accessible. This is particularly true of special-purpose hardware devices. For example, it may be difficult to provide remote access to facilities to perform Fast Fourier Transforms or Array Processing; or this access may be sufficiently slow to prohibit successful accomplishment of real-time objectives.

The next sections discuss several systems which have been designed to support process migration. While some of the systems are operational and others are only paper designs, the basis for inclusion was an innovative solution to some problem. The

---

1.) Surprisingly, there are many such situations. For example, an application which requires a dedicated node to ensure real-time response, such as robot control, may cause a notice to be posted to other computations, which can then choose alternate nodes to continue their execution. Another example is presented above.

2.) MOS is mentioned in a later section of this paper, but the discussion does not provide details on the load balancing algorithm.

3.) That is, it analyzes and reduces the volume of data by generating some result.
implementation status is noted in each section.

2. LOCUS

The LOCUS operating system [Walker1983a] developed at UCLA provides a facility to migrate processes; the LOCUS developers refer to this facility as "network" tasking. This development is described in Butterfield and Popek [Butterfield1984a].

The LOCUS operating system is based on UNIX, and attempts to provide semantics equivalent to a single machine UNIX system. Kernel modifications provide transparent access to remote resources, as well as enhanced reliability and availability.

The version of UNIX upon which LOCUS is based has very limited interprocess communication facilities. The major mechanism is the pipe, a one-way link connecting two processes, a reader and a writer; the kernel provides synchronization between processes by suspending execution of the writer(reader) and awakening the reader(writer) when the pipe is full(empty). Pipes have the unfortunate characteristic that they must connect related processes, where related is in the sense of common ancestry in a family tree defined by fork() system calls. This deficiency is remedied in other versions of UNIX [Presotto1985a, Joy1982a]. Other mechanisms for communication exist, such as signals but these are clumsy, carry little data (about 4 bits per signal), and depend on privilege or inheritance relationships between the communicating processes.

LOCUS addresses the access of remote resources through modification of the file system interface. This is consistent with LOCUS's basis in UNIX [Ritchie1974a], as UNIX resource access is primarily through files. LOCUS provides transparent access to remote resources by divorcing the location of the referenced object from its name; the LOCUS file system appears to the user to be a single rooted tree. Path name interpretation results in a file handle, roughly equivalent to a UNIX i-node[Thompson1978a]. This handle has associated with it a file group; a file group is similar in function to a UNIX file system, except for the following significant differences:

- There may be multiple copies of the file group, each on different nodes of the system.
- A given copy of the file group may be incomplete; that is, it may contain only a subset of the files contained in the entirety of the file group.

For example, a tape drive attached to a particular LOCUS node might be accessed via the name /dev/tape12; the name resolution process (which may include several remote references in the course of traversing the directory tree) would result in a handle which would be used in (perhaps remotely) controlling the device. Note that multiple copies of whatever is the result of the resolution process cannot exist in this case, since there is in fact only one copy of the physical device available.

UNIX file system semantics are such that the following is true:

- Name interpretation is done in order to return an object which can be used to access the named resource, the file descriptor.
- All access is performed by means of the file descriptor, e.g., data transfer.
- The name space is distinct from the object space, in the following sense:
  - objects may have several names, via links
  - objects exist which have no name. An obvious example is the pipe; a less obvious but still common example is provided by this sequence of 2 system calls, the first of which obtains a descriptor for the file "name" and the second which removes "name" from the namespace.

```c
    fd = open( "name" );
    unlink( "name" );
```

The descriptor remains valid, thus this a popular method for creating temporary files which do not exist past program termination.

The motivation for file descriptors is the quick lookup of the state information the kernel maintains about the file, as well as saving the overhead of path name interpretation on each file operation. As files can be accessed a byte at a time, this overhead is potentially enormous.

Associated with a running process are such data as:

1.) A table of currently open file descriptors, with which are associated other kernel state such as the current position within the file.
2.) A current working directory, used to facilitate abbreviated relative path names.
3.) A process identifier.
4.) Pending signals, which are not handled until
the process is next run.
5.) Zero or more child processes, created
through \textit{fork()}, from each of which an exit
status value can be obtained.
6.) Kernel table entries which are used to pro-
vide various system services, for example,
virtual addressing.
7.) Miscellaneous data, such as parent process
identifier, CPU utilization, etcetera.

Process creation is done via the \textit{fork()} system
call, which creates a new process executing the same
program image as the caller (the return value of the call
allows parent and child processes to identify them-
selves as such). \textit{Exec()} replaces the code and data of a
running process with a named program image. These
calls are modified in the LOCUS system so that the
newly created process may begin executing at an alter-
native site; mechanisms exist with which a set of possi-
bile execution sites may be associated with a process.

In addition, a system call \textit{migrate()} was added to per-
mit a process to change its location while executing.
\textit{Exec()} can be performed across heterogeneous CPU
types, as the new process state is derived from an
image file which can be specific to each processor
type; \textit{fork()} and \textit{migrate()} cannot, as existing state
information such as registers and the instruction stream
cannot be translated. A complete discussion of the
heterogeneous \textit{exec()} mechanism is provided in Butter-
field and Popek [Butterfield1984a].

As the LOCUS system provides global access to
objects at the kernel level, the descriptors in the pos-
session of the migrated process are still valid. Other
UNIX semantics required more implementation effort
to preserve. In particular, the delivery of signals and
shared file pointers are difficult as a result of the seman-
tics of \textit{inheritance}; signalling a child process
requires that the current location of the child process be
known or discovered. This can be difficult where the
child process has migrated, perhaps more than once.
The file pointer problem requires that new system file
table entries be created for a migrated process; see

3. DEMOS/MP

DEMOS/MP is a distributed system developed at
the University of California, Berkeley [Powell1983a].
DEMOS/MP is based on the message-passing para-
digm, where communication between active processes
is carried on by means of kernel-managed messages. It
is based on the earlier DEMOS system for the CRAY-1
[Baskett1975a] which used message passing as the
communication mechanism. Messages are passed by
means of \textit{links} associated with a process; a process
willing to accept messages creates a \textit{link} ; the \textit{link}
can be passed in a message. \textit{Links} are managed by the
DEMOS/MP kernel; the kernel participates in all link
operations even though conceptual control remains
with the process the link addresses (its creator). \textit{Links}
are context-independent in the sense that a link always
points to this originator, even if it has been passed to
another process.

Associated with the link is a \textit{message process
address}, consisting of a \textit{< process identifier, last
known address >} pair. The globally unique \textit{process
identifier} is in turn composed of a \textit{< creating machine,
local identifier >} pair. When a process is moved to a
remote system, the following sequence of events
occurs:

1.) Remove the process from execution. This
state is marked, but the system state (e.g.,
BLOCKED, RUNNING) is left untouched;
arriving messages are placed on the message
queues associated with each link.
2.) Ask the \textit{destination} kernel to move the pro-
cess. Note that control is held by the \textit{desti-
nation} kernel up until Step #6.
3.) Allocate process state on the \textit{destination
processor}. The newly allocated process state
has the same \textit{process identifier} as the migrate-
ging process. Resources such as the swap
space are reserved at this time.
4.) Transfer the process state information main-
tained by the kernel.
5.) Transfer the program. Memory allocation, \textit{et
cetera}, are taken care of by the primitives for
data transfer. Control returns to the \textit{source
kernel} at completion.
6.) Pending messages are forwarded to the \textit{desti-
nation} ; the location portion of the process
address for the message is changed to reflect
the new location.
7.) On the \textit{source}, all state data (e.g., memory)
is deallocated. A degenerate process state,
called the \textit{forwarding address}, points to the
\textit{last known} machine to which the process was
migrated (This information can be updated
later, if newer data arrives). This is the
clean-up.

8.) The process is restarted on the destination.

4. XOS

X-TREE is an architecture for the design and construction of distributed microprocessor computer systems. It provides a model for building powerful, low-cost systems comprised of many identical microprocessor chips, communicating using a tree structure. Computers using such a tree-structured architecture exist and are operational in research environments [Lerner1985a, Shaw1981a] For example the DADO-2 processor [Stolfo1987a] developed at Columbia University is composed of 1023 Intel 8751 processors connected in a binary tree structure. Input/Output devices in the X-TREE architecture are attached to the leaf nodes of the tree\(^4\). This is illustrated in Figure 3.

```
The idea is that this could aid in effective utilization of the connection tree by reducing traffic in the tree. Traffic would be reduced by clustering communicating processes closely; in this way, communications which spanned long stretches of the tree would be minimized, thus reducing the global traffic.

XOS uses the paradigm of processes communicating via messages and message streams. A process exists on only one processor at a time although it may migrate from one processor to another during the course of its computation. The process is described in its entirety by the Process Work Object (PWO). A single capability (the process pointer) points to the PWO; the PWO encapsulates all of the information necessary for process control. The compact representation enables both rapid context switches and the ability to swap the process to disk and from there on to another node of the X-TREE. This transfer can also take place directly between the nodes, without the intervening swap. It should be clear that these actions are sufficient to migrate the process to another processor.

The interprocess communication is similar to that of DEMOS [Baskett1975a] in that it is message-based, unidirectional, and capability accessed. Remote and local communication appear the same to processes.

All messages are sent to a port, owned by some process. When a process wishes to receive messages, it creates a port object and passes send capabilities for that port to other processes. It can do this either by:

- saving the capabilities in commonly-accessible (i.e., "well-known") objects; or
- handing them off to a "switchboard" process with which every process can communicate.

Note that when a process moves, all processes communicating with it have to update their pointers to its port objects, which have moved with it. The pointers are treated as hints to the sender; they may be wrong. If a send arrives at the wrong node, a special NAK is sent to inform the sender that the process has moved. In this case, the port object is fetched from the disk, and the send retried; this continues until the "roving" process is found, i.e., the message catches up with it.

Interestingly, XOS supports both Datagram and Stream (Virtual Circuit) types of communication.

\(^4\) This is not the only method for connecting I/O devices. DADO-2, for example, does not use this method, while the NON-Von-4 proposed connecting them at an intermediate level.

Figure 3: X-TREE Architecture

One of the specific goals of the XOS operating system [Miller1981a] designed for the X-TREE architecture is to provide support for process migration.
5. V

The V kernel [Cheriton1984a, Lazowska1984a], developed at Stanford University, is a message-oriented kernel which provides uniform local and network interprocess communication. It is modelled after the earlier Thoth [Cheriton1982a, Cheriton1979a] operating system which influenced the choice of kernel primitives. These are:

- `Send(message, pid)`
- `pid = Receive(message)`
- `<pid, count> = ReceiveWithSegment(message, segptr, segsize)`
- `Reply(message, pid)`
- `ReplyWithSegment(message, pid, destptr, segptr, segsize)`
- `MoveFrom(srcpid, dest, src, count)`
- `MoveTo(destpid, dest, src, count)`
- `SetPid(logical_id, pid, scope)`
- `pid = GetPid(logical_id, scope)`

Processes are identified by means of a globally unique process identifier, or `pid`, where global is meant to be interpreted within the context of a given local network. A `<host id, local unique id>` pair comprise the `pid`; thus the mapping from `pid` to process location can be done rapidly. Interprocess communication is designed to be performed via fixed-size messages; these are 32 bytes in length. The typical interaction between two communicating processes A and B is illustrated in Figure 4:

![Figure 4: Typical Interprocess Communication in V](image)

Process A remains blocked until process B posts a reply; note that the synchronous nature of communication allows copying directly between process address spaces as well as re-use of the message "Amsg". Thus no kernel buffers are necessary and queueing problems are thus reduced. Messages are queued for receiving processes in FIFO order. The other message primitives are for larger data transfers, for example page or file access. Experience with earlier designs led to this distinction between small messages and a facility for data transfer. The non-communication kernel primitives manipulate `pids`; scopes are `local` or `remote`; `logical_ids` are, e.g., `fisleserver` or `nameserver`; thus, `SetPid()` could be used to identify a particular process to the network as a `fileseserver`. Much of the design of the V kernel is motivated by high performance; particular attention is paid to the efficiency of the kernel's execution time with respect to network `penalty`, a measure of the difference in cost between performing an operation locally and performing it remotely.

Facilities for process migration exist in the V system [Theimer1985a]; the designers refer to these as facilities for preemptable remote execution, the idea being that remote execution is a good thing but shouldn't cause users to lose control of their workstations. The preemptable remote execution facilities of the V kernel allow idle workstations to be used as a "pool of processors". Three basic issues are addressed in the design:

1.) Programs should have a network-transparent `execution environment`, where the names, operations, and data with which the program can interact comprise this environment. Environments including directly addressed hardware devices such as graphics frame buffers present a problem.

2.) Migration of a program should not introduce excess interference, either to the progress of the process involved, or to the system as a whole. Migration requires atomic transfer of a copy of the program state to another host. Atomic transfer ensures that other system components cannot detect the existence of multiple process copies. Suspending the execution may lead to failures due to delays in interactions with other processes, and hence the time must be kept short.

3.) A migrated program should not exhibit dependencies on previous locations, in the sense that the previous host should not
contain state information about the process, e.g., location pointers necessary to forward messages queued at the previous host. Otherwise, it is argued, there is still a load imposed on the host, which reduces the benefits of migration. In addition, previous host failure may cause a program to fail due to dependencies.

In V, the execution environment is transparent, because:

- The address space is virtualized by the kernel equivalently across nodes, and is thus transparent.
- All references outside the address space are performed using network-transparent interprocess communication primitives and globally unique identifiers, as described previously. The exceptions are the host-specific kernel server and program manager.
- Programs which directly address a device cannot be migrated. This is typically not a problem, as most programs access devices through globally accessible device servers which remain co-resident with the device.
- The exceptions to the transparent access, the kernel server and the program manager, provide identical services to all processes; they can always be located by virtue of their membership in "well-known" process groups.

The V implementation reduces the amount of time a process is suspended by pre-copying a large amount of state; the designers note that with multi-megabyte virtual address spaces, the amount of state implies a great deal of data transfer. Migration of a process is actually migration of the logical host containing the process. A logical host is defined by an address space in which multiple V processes may run.

The procedure to migrate a logical host consists of the following 5 steps:

1.) Locate another workstation (via IPC with members of the program manager group) that is willing and able to accept the migrating logical host.

2.) Initialize the new host to accept the logical host.

3.) Pre-copy the state of the logical host to be migrated to the new site.

4.) Freeze the migrating logical host and complete the copy of its state.

5.) Unfreeze the new logical host, delete the old logical host, and rebind references.

Of particular interest are the mechanics of setting up the new logical host and pre-copying its state from the previous host. The new logical host is set up with a distinct logical host id; this allows it to be accessed distinctly from the previous logical host, e.g., by the CopyTo() and CopyFrom() primitives for data transfer. When pre-copying is completed, the source logical host is frozen, and the destination logical host assumes the previous logical host id, in order to facilitate relocation of that logical host. Pre-copying is implemented (once a destination logical host has been set up) by the following algorithm:

```plaintext
repeat
    transfer all state in the
    old logical host
    which has changed since
    the last state transfer
    to the new logical host;
until( changed state is small );
```

Figure 5: V Pre-copying Algorithm

Each iteration of this loop should be more rapid, as the amount of data transferred should decrease, thus decreasing the amount of state data that the source logical host has opportunity to modify.

At termination of this loop, the copy operation is completed, by:

- copying the remainder of the frozen source logical host's changed state.

- deleting outstanding interprocess communication requests. Senders are prompted to resend to the new host; V's interprocess communication mechanism ensures that senders will retry until successful receipt of a reply.

As described so far, this approach only deals with state in the address space, kernel, and program manager. Relevant state which is not in some globally accessible server, e.g., a network file server, is migrated with the logical host in order to remove dependencies on previous hosts. This includes open files located on disks local to a node; they are considered extensions of the program state. The previously described mechanisms could therefore be used to
move them as well. It is noted, however, that the files could be arbitrarily large, thus introducing considerable delay. A given file may already exist on the destination, thus saving copying; if the remote copy is a different version of the file, the copy may be destructive and hence undesirable. A program may have the symbolic name of the file stored internally, thus preventing changing the symbolic name upon migration. This issue, of open file migration, is currently not addressed, as the V System consists of diskless workstations.

6. WORMS

The notion of a worm process is described in Shoch and Hupp [Shoch1982a]. The idea is somewhat different than the other process migration schemes discussed in this proposal, in that other schemes have aimed to be transparent to the process which is being migrated, while the worm mechanism and supported processing are very much aware of the underlying network and its topology. Their basic model of a worm process is: "A program or computation that can move from machine to machine, harnessing resources as needed, and replicating itself when necessary". Shoch and Hupp provide an example which is referred to as "The Blob",

"... a program that started out running in one machine, but as its appetite for computing cycles grew, it could reach out, find unused machines, and grow to encompass those resources. In the middle of the night, such a program could mobilize hundreds of machines in one building; in the morning, as users reclaimed their machines, the "blob" would have to retreat in an orderly manner, gathering up the intermediate results of its computation. Holed up in one or two machines during the day, the program could emerge again later as resources became available, again expanding the computation."

We make two observations before discussing the details of worm processes:

1.) The worm program makes decisions about where and when to move.
2.) The program logic is aware of the distributed nature of the computation.

6.1. What's in a worm?

The worm is so-called as a result of the organization of the computation; the computation is broken up into segments; the segments are distributed across one or more machines. The name derives from the ability of a segment to regenerate the entire worm if the need arises. The segments of the worm remain in communication with each other while the computation is carried out; the mode of communication is described later, when the worm mechanism is discussed. The worm mechanism is the support mechanism designed to create (e.g., allocate machines for) and maintain the worm segments, and is thus distinct from the user programs built on top of the mechanism. The mechanism is discussed in the next section.

6.2. Worm Mechanism

A worm consists of the following logical pieces:

- Initialization code to run when the worm is started on the first machine.
- Initialization code to run when started on any subsequent machine.
- The main program, incorporating the maintenance portions of the worm mechanism.

The tasks of this mechanism are as follows:

1.) Locating other machines. (The physical configuration of the testbed is a set of over 100 Xerox Alto [Thacker1982a] workstations, connected via the Xerox experimental Ethernet [Metcalf1976a].) The first task of a worm is to fill out its full complement of segments. A simple protocol using a special packet format is used to find free machines; communication is point-to-point.

2.) Booting an idle machine. The idle machine is instructed to reboot from the network; instead of the normal file-server supplied bootstrap procedure, the worm supplies itself as bootstrapping code. Thus, the worm code copied will be an exact copy of the running segment which is attempting to obtain a new machine. Some necessary initialization is performed by the new segment after arriving at the new node.

5.) Special in the sense that it is customized to this application, rather than using a packet format from a general purpose protocol.

6.) There is logic in the worm mechanism which allows a new-
3.) Intra-worm communication. "I'm alive" status packets are sent via brute-force multicasts\(^7\) which are used to update status tables in receiving segments; the "death" of a segment causes, after a time, a new copy of the dead segment to be spawned by one of the remaining segments. Note that in the case of a network partition, two (or more, depending on the nature of the partition) complete worms may be created, if the worm is split across the partition.

4.) Termination. Worms release the machines they are using by causing the machine to reboot the code a machine runs when otherwise idle, a memory diagnostic, from the network.

Some features for worm management are also present; in particular, there is an escape mechanism which causes the worm to stop whatever it is doing when a special packet is received over the network. This was used to avert a catastrophe caused by an unstable worm (it took over almost all of the machines on their network), and led to the development of a worm management utility to detect and prevent continuation of unusual growth. This, along with better error detection and a larger exchange of information, led to more stable worm behavior.

The next section discusses several applications which were built using the worm mechanisms.

6.3. Applications

The first application was an "existential worm". This served to demonstrate the efficacy of the mechanism; the worm consisted of a multi-segment worm which ran an essentially null application program. This served to test the management mechanism, communication mechanism, stability of the system, and the ability of the worm to operate despite machine failures. Machine failures were artificially induced; this served to demonstrate the robustness provided by a multi-machine worm.

While there were other applications, we will discuss only one other here, the "alarm clock worm", which was an application requiring reliability. The worm implemented the alarm by means of an outgoing call made via a dialer. The interesting features of this worm are:

- Must maintain a consistent database of the alarms to be rung.
- Each segment retains its own copy of the database.
- Newly-created segments are given the current list (this should happen by default, as the complete state of the creating segment will be passed to the new segment) when they start up.
- New alarm requests are propagated by the segment which accepted the request.
- Synchronization and locking of the alarm delivery were made by the segment delivering the alarm; it began by informing the other segments that it was about to make the call, and when finished with the call, it informed them to delete the entry from their database copies.
- A segment of the worm had to be found to place the alarm request into a segment; a separate user program was written which made contact with a segment.

7. Application-Directed Process Migration

The basic idea behind Application-Directed Process Migration [Maguire1986a] is that the application involved specifies when it is to migrate, and perhaps where it is to migrate to. Thus transparency is not desired at this layer of the system; the migration from point-to-point must occur under process control rather than transparently. This is necessary, for example, to provide the widest possible dispersion of processes across processors - a mechanism which precludes knowledge of the process to processor mapping cannot provide this. This dispersion is a desirable attribute where we wish to take advantage of available hardware redundancy.

There are essentially two things which must be specified about migration and associated process replication:

1.) When the process is to moved/copied.
2.) Where the process is to be moved/copied to.

One way to coordinate the migration of the process is to have the process migrate itself; this can be
done with a checkpoint/restart facility; such a facility is discussed in the following sub-section.

7.1. Checkpoint/Restart

A checkpoint/restart facility is one which will allow a process to save its state to a checkpoint; this checkpoint will later be subjected to a restart procedure which will resume execution of the checkpointed process at the point at which the checkpoint was made. Such a facility is referred to as a "Checkpoint/Restart" mechanism; such mechanisms have been available in operating systems since the 1960s; see Austerland, et al [Austerland1981a]. for a historical perspective.

There are several choices one can make in the restoration of process state. For example, the DEMOS/MP system [Powell1983a, Powell1983b] described above records all messages which are sent. Since all communication between a process and other entities (e.g. the operating system or other processes) takes place via messages, recording of these messages essentially captures and records any information which could have caused a state change. Thus replaying the messages serves to bring a process up to the correct state, from the last time it had been completely checkpointed. This scheme, of course, relies quite heavily on the reliability of the recorder. The idea of capturing state changes in terms of messages, and recovery with message replay, was also used in the Auragen™ System, described by Borg, et al [Borg1983a].

We can use a checkpoint/restart facility combined with file transfer facilities as a simple scheme to provide process migration. This works as follows:

1.) A running process creates a checkpoint.
2.) The data of the checkpoint is transferred to the remote destination.
3.) The checkpoint data is used to create an up-to-date running process on the remote machine.

These three steps carry out the actions illustrated in Figures 1 and 2. A description of the detailed construction of such a mechanism is in Ioannidis and Smith [Ioannidis1987a].

8. Discussion

The introduction to the paper [Powell1983a] describing DEMOS/MP's process migration scheme makes the following observation:

Process migration has been proposed as a feature in a number of systems, but successful implementations are rare. Some of the problems encountered relate to disconnecting the process from its old environment and connecting it with its new one, not only making the the new location of the process transparent to other processes, but performing the transition without affecting operations in progress. In many systems, the state of a process is distributed among a number of tables in the system making it hard to extract that information from the source processor and create corresponding entries on the destination processor. In other systems, the presence of a machine identifier as part of the process identifier used in communication makes continuous transparent interaction with other processes impossible. In most systems, the fact that some parts of the system interact with processes in a location-dependent way has meant that the system is not free to move a process at any point in time.

This observation gives us some insight into the reasons why port or message based systems (such as DEMOS/MP and Stanford's V system) implement process migration more easily than other system designs. For example, the Amoeba distributed operating system [Tanenbaum1981a, Tanenbaum1986a] developed at the Vrije Universiteit, Amsterdam, under the direction of Andrew Tanenbaum, has as basic components: processes, messages, and ports; processes are active entities, communicating by exchanging messages via their ports. Tanenbaum and Van Renesse [Tanenbaum1985a] compare the implementation to some other systems. The Accent [Rashid1981a] system developed at Carnegie-Mellon University (CMU) also uses the paradigm of message-passing between processes through ports as the means of interprocess communication; the kernel provides support for message passing, process creation and deletion, virtual address spaces, and little else. While the Accent system as described by Rashid and Robertson [Rashid1981a] does not support process migration, it has been implemented [Zayas1987a].

The reason for the ease of implementation is the message-oriented system's designs8; a small kernel of

8.) Designers of early message-based distributed systems, such
message-passing routines contains little state not in the process’s context, and thus there is little at a given location that a process can be dependent upon. All of the changes in the process’s state are the result of passed messages. This property is taken advantage of by Powell and Presotto [Powell1983] in order to build a reliable distributed system; all messages are recorded, to be replayed if a process fails in order to provide it with the correct state. This idea is essentially “logging” from Database systems [Bernstein1987, Date1982] applied to the context of process address spaces. XOS would be expected to have this same advantage had it been completely implemented: it is interesting mainly as an example of how process migration could allow better utilization of a tree-structured multiprocessor.

LOCUS has more difficulty as the UNIX process model [Thompson1978] requires a great deal of context to be maintained. However, given that the file system is the main point of interface, and that the file system name space is global in LOCUS, process migration is eased somewhat. Without such a name space, there are several troublesome issues; some of these are discussed by Cagle [Cagle1986], Chen [Chen1986], and Ioannidis and Smith [Ioannidis1987].

The MOS system, developed at the Hebrew University of Jerusalem [Barak1985, Barak1986], is also based on UNIX and attempts to emulate a single machine UNIX system using a collection of loosely connected independent homogeneous computers. MOS has a global name space; a feature of the implementation is the notion of a universal i-node pointer; these effectively provide uniform (i.e., transparent remote) access to resources, as in the LOCUS system; thus while the process possesses context, a great deal of it is location-independent. This makes process migration less difficult. This same sort of global naming scheme is employed by the UNIX-like Sprite [Ousterhout1983] operating system.

Typically, the designers have intended their efforts to be transparent; this is not always the case. Consider WORM processes, which are aware of their components and location. State is explicitly managed by the WORM mechanism, and the programmer of the WORM develops an application’s fault-tolerance if that is required.

In any case, these process migration mechanisms demonstrate that the state of an executing process can be moved between homogeneous machines, and that the execution can be continued. The transfer of address spaces is interesting because the methodology has a strong effect on the utility of the scheme. For example, Sprite [Ousterhout1985] and the checkpoint-based schemes create state descriptions in the file system; thus mechanisms which exist to copy files can be used to create replicas of processes.

9. References


---

as Farber’s [Mockapetris1977] Distributed Computing System noted the ease of implementation.

9.) Smith and Maguire [Smith1988a] point out some problems with migration between heterogeneous architectures.


[Date1982a] Christopher J. Date, An Introduction to Database Systems, Addison-Wesley (1982).


[Rashid1981a] Richard F. Rashid and George G.


