A Survey of Process Migration Schemes

Jeremy Elson
jelson@usc.edu

18 December 1998

1 Introduction

In days gone by, computers were prohibitively expensive. Most organizations had only a small number of computers, and each computer operated autonomously. However, as the cost of computers dramatically fell, it became reasonable to conceive of systems whose overall computational power could be increased by using more than one processor to simultaneously do work on the same problem.

There are a variety of paradigms currently in use for the execution of processes on remote processors. Often remote execution takes the form of a simple request-response protocol as in RPC systems: a process sends a message to a remote processor, requesting that some computation be done; the processor performs that computation and then returns the result to the requester.

Another common paradigm is explicit parallel programming: a programmer defines multiple concurrent tasks, encapsulating each in an OS process, and each process is run on a different processor. Most parallel programming environments decide where each process will be run at the time the process is born. Once a process begins, it must remain resident on the same processor until the completion of the task. Even if the scheduling of processes to processors turns out to be suboptimal, it cannot be changed. Often it is impossible to predict how the load on the system will change over time, and thus often impossible to optimally schedule processes to processors.

In an attempt to gain better performance, a number of researchers have developed a different class of parallel execution environments that allow processes to move from processor to processor dynamically at any point in the life of the process. A change in processor residency in the middle of the lifetime of the process is typically called "process migration." By dynamically moving processes throughout their lifetimes, the system can potentially adapt better to changes in load that could not be foreseen at the start of the tasks. Proponents of process migration claim that this dynamic adaptation leads to a better system-wide utilization of available resources than static process scheduling.

This paper will examine various aspects of process migration schemes that have developed over the past several years. Section 2 will describe some of the operating system features that were necessary precursors to the development of process migration schemes. Section 3 reviews some of the factors that motivated the development of process migration schemes. Many of the more common implementation issues that all such schemes must address will be considered in Section 4. Section 5 will describe several real operating systems that support process migration, and compare the implementation decisions made by their designers. Conclusions are presented in Section 6.

2 Precursors to Process Migration

Process migration is not a concept that was suddenly invented in a vacuum; like many other features, it is an incremental development that builds on and is motivated by other features found in many operating systems. This section will review some of this initial groundwork.

Perhaps the biggest single idea that is critical to the success of process migration is *transparency.* Transparency, in its most basic form, is the replication of an environment on more than one computer. This replication gives a user the ability to run the same program on any of a number of different computers, with the expectation that the program will have the same behavior regardless of which computer is actually running it.

Sun's Network File System [1], or NFS, is an early example of a system that supported limited transparency. By carefully mounting the same network filesystems in the same place on each client node, the namespace of the filesystem would appear to be the same on all of the nodes. Other facilities, such as Sun Network Information Service [2], or NIS, allows easy replication of network configuration data and user authentication information.

The first steps described above led to the development of "computing clusters," or large-scale groups of computers under the same administrative pervue that present a homogeneous computing environment to the user. Examples of these cluster projects can be found in MIT's Project Athena [3] and CMU's Andrew [4]. The goal of such clusters was to allow a user to sit down at any workstation on campus and gain access to her familiar environment, including her home directory.

In a cluster of homogeneous workstations such as those described above, enough transparency was provided to allow remote process execution using the simple Berkeley *rsh* command. Advanced users soon became adept at splitting tasks into multiple, parallel jobs, and running each job on a different machine using *rsh*. While this method was effective in some cases, it does suffer from some obvious problems: for example, users running parallel jobs had to manually determine which machines were available for running jobs.

The LOCUS operating system [5], developed at the UCLA, was another early example of a system developed with transparency as one of its primary goals. It was designed "from the ground up" as a distributed operating system, supporting transparent access to data via a network-wide filesystem, and transparent remote process execution in a more elegant way than *rsh*. Although LOCUS did arguably present the user with a better interface than other systems, it still required manual selection of nodes on which to run processes.

At this point, the state of the art in operating systems provided high levels of transparency between machines—users now had at their disposal dozens or even hundreds of different computers, all of which could run a process with the same results. The next logical step was to develop methods to automate the distribution of processes to processors, based on usage or load patterns on the processors. Systems such as *butler* [6] helped to solve this problem by monitoring the load on the machines in a cluster and assigning processes to machines with low load. This is called *static scheduling*, because once the system makes the decision of the set of machines to receive processes, the processes must run to completion on those machines.

In addition, parallel programming environments such as MPI [7], PVM [8], and Linda [9] made it easier for programmers to take advantage of transparent system clusters to solve problems in parallel. Each of these systems consisted of two components. The first component is a programming language or language library, used by the programmer to specify a parallel computation. The second component is a run-time environment, used to execute the specified parallel computation.

A parallel run-time environment is responsible for the details of creating processes and assigning the processes to processors. The parallel run-time systems mentioned above all use static scheduling. Some of the run-time environments make their processor selections based on information such as the current load on the set of possible target machines.

3 Motivations for Process Migration

Clearly, it seems to be much easier to decide on which node a process will run at the time that it's born than it is to disembody an actively running process from one node and resume its execution on another. As we will see in later sections, there are many subtle, nontrivial implementation issues to be considered. This leads us naturally to the question: is the benefit of process migration worth the cost? As with most engineering questions, there is no single, clear-cut answer. Different environments have different needs; in some, process migration can be a big benefit, and in others it has little positive impact.

For example, consider a loosely coupled parallel supercomputer such as the IBM SP/2 [10]. Such a machine is often used for so-called "Grand Challenge" problems such as fluid dynamics and molecular kinetics. These types of problems are "embarrassingly parallel," meaning they can scale evenly to virtually any number of processing elements. In addition, the processing elements are not shared with multiple users during computation; a node is assigned exclusively to a particular user for the duration of the job. In this scenario, the optimal distribution of processes to processors can be determined before the processes begin running. The load will always be evenly balanced, and there will never be short-term contention for a node's resources (i.e., contention takes place at course granularity when assigning processes to users for hours at a time). Process migration will be of virtually no use in this case.

On the other hand, consider an environment where there are a large number of processes concurrently running that belong to different users. It is impossible to predict how many users will be contending for resources or how long any of their jobs will be. This is a typical usage pattern of a central university computing facility. In this case, there are two typical solutions:

- Buy a very large, very expensive multi-processor SMP machine such as an SGI Challenge or Digital AlphaServer
- Buy numerous smaller, lower-powered workstations—configured with transparency, such as in MIT's Athena or CMU's Andrew system—and coarsely balance the load across them by distributing the users. (This can be done by asking users to "log in to a random machine", or by using software that distributes user logins, such as a load-balancing DNS [11] server.[1])

The SMP solution has effective load-balancing across processors due to the tightly coupled nature of the system. However, such a solution can be quite expensive and also suffers from a lack of redundancy—a failure of the single, central machine denies service to all users. It is also not very scalable; most SMP machines can not support more than 16 processors. The second solution, while more scalable and resistant to failures, will often suffer from severely unbalanced loads. This is because a "user" is not a constant unit of load; different users generate different loads, and a particular user's generated load can change from minute to minute.

In this case, an operating system that supports dynamic process migration can be very beneficial. With a pool of processing nodes (e.g., workstations or PCs) dedicated to servicing the user load, an efficient process migration scheme can balance the user load almost as effectively

[1] A load-balancing DNS server typically keeps records of the load on all the machines in the cluster. Users do not connect to a specific machine, but a generic alias representing the cluster. Each time a client attempts to resolve the cluster name into an IP address, the DNS server dynamically resolves it to whichever machine has the lowest load.

as an SMP machine. In addition, the pool of processors is far more redundant than a single SMP machine. Plus, it is much more scalable: if the control of the pool is appropriately designed (i.e., distributed), as many nodes as desired can be added, incrementally, as the expected nominal load on the system increases. These are the goals that motivated the development of MOSIX [12].

Other environments fall somewhere in between these two extremes. For example, a popular trend these days is to harness the power of idle personal workstations. In this case, there is no investment made in centralized computing resources, as in the previous two examples. Rather, the model is that a typical user—say, Alice—has a workstation on her desk. Alice is willing to let her workstation be put to good use when it's idle, as long as she can have complete control of the machine when she is sitting in front of it.

This is an example where static scheduling goes a long way, but does not quite solve the problem completely. Static-scheduling systems such as Linda [9] can automatically monitor the load on all workstations, and schedule processes to idle processors when a parallel job begins. The trouble comes if Alice returns to her workstation while Bob is in the middle of running a large simulation on it. If we are using static scheduling, our options are limited:

- Allow Bob's job to run to completion. This makes Alice unhappy because her workstation will be slow until Bob's job is done.
- Immediately evict Bob's job from Alice's workstation by terminating Bob's process. This makes Bob unhappy because he loses work.

It was the desire to find a solution that satisfies both Alice and Bob that motivated the development of process migration in systems such as Sprite [13] and Condor [14]. The designers of these systems were less interested in doing dynamic load balancing in a pool of dedicated processors, but rather were looking for a graceful way to evict processes from foreign machines when the machines' owners returned.

4 Implementation Issues

There are many issues to consider when developing a system that will migrate an active process from one node (the *source*) to another (the *target*). Most of these issues must consider the interactions between two factors. The first factor is the level of *transparency* between the source and target—for example, do the two nodes share the same view of the filesystem? Or, do they have the same hardware peripherals?

The second factor is the management of the process' *state*. For each element of state that defines a process, the implementor of a migration scheme has three options when migrating the process to a target node:

- Move the state from the source node to the target node along with the process, and let the target node service requests that use the state;
- Keep the state on the source node, and have the process send requests to use that state from the target node back to the source node; and,

- Ignore the state, and accept the resulting change in the semantics of the program on the target node with respect to the source node.

Let us now illustrate the difference between these decisions with a design example. Each process has a set of file descriptors that describe aspects of files that the process is currently accessing; for example, the current access position within the file, and (implicitly) the name of the file being accessed. When migrating a process to a target node, we might choose to forward this file state information to the target node along with the process. Later, when the process running on the target node attempts to access a file, the target node can locally fulfill that request using the state information imported from the source node.

Alternately, the state of the file can be kept on the source node. When the process running on the target node attempts to access a file, it forwards the request back to the source node. The source node fulfills the request and returns the results (for example, data read from the file) to the target node.

This example also serves to illustrate the relationship between the process state and the transparency between the nodes. Forwarding the state to the target node is only useful if transparency between the nodes gives us a guarantee that the state will have same meaning on the target node as it did on the source. For example, forwarding the name of an open file and current access position to a target node is of no use if the target node does not have the same view of the filesystem as the source does.

Our open-file example illustrates the interplay between the decision of where to store a process' state and the level of transparency available on the system. There are a number of other decisions similar to the open-file decision that must be made as well. These are explored in the next section, along with the decisions made by the designers of several working migration systems.

5 Comparison of Implementations

In this section we will explore some of the similarities and differences in the design decisions among several systems that support process migration. Specifically, we will examine MOSIX [12], developed at the Hebrew University of Jerusalem, Israel; Condor [14], developed at the University of Wisconsin–Madison; and Sprite [13], developed at the University of California at Berkeley. A summary of each system's primary design decisions is presented in Table 1. Each of these decisions is discussed in more detail below.

5.1 General Philosophy

Condor, MOSIX, and Sprite appear similar on the surface in that they all have implemented a process migration. Underneath, however, each of the three systems is trying to solve a different problem. The design decisions that went into the systems are necessarily different, because the designs were based on different assumptions.

MOSIX might be best described as an attempt to create a low-cost equivalent of a scalable, SMP (multi-CPU) server. In an SMP system such as the Digital AlphaServer or SGI Challenge, multiple CPUs are tightly coupled, and the operating system can do very fine-grained load balancing across those CPUs. In an SMP, any job can be scheduled to any processor with virtually no overhead. MOSIX, similarly, attempts to implement very low-overhead process migration so that the multicomputer, taken as a whole, might be capable of fine-grained load balancing akin to an SMP. The MOSIX designers have expended a great deal of effort implementing very fast network protocols, optimizing network device drivers, and doing other analyses to push the performance of their network as far as possible.

Also in line with the SMP model, MOSIX goes to great lengths to maintain the same semantics of a centralized OS from the point of view of both processes and users. Even when a process migrates, signal semantics remain the same, IPC channels such as pipes and TCP/IP sockets can still be used, and the process still appears to be on its "home node" according to programs such as `ps`.

As a result, the MOSIX implementation typically takes the form of a "pool of processors"—a large number of CPUs dedicated to acting as migration targets for high-throughput scientific computing. Although MOSIX can be used to borrow idle cycles from unused desktop workstations, that mode of operation is not its primary focus.

In contrast, Condor's primary motivation for process migration seems to be to provide a graceful way for processes that were using idle CPU cycles on a foreign machine to be evicted from that machine when it is no longer idle. They made many simplifying assumptions; for example, that the remotely-executing processes will be running in a vacuum, not requiring contact with other processes via IPC channels. Their migration strategy does not provide a fully transparent migration model; processes "know" that they are running on a foreign machine, and the home machine has no record of the process' existence. These assumptions, while more limiting than the MOSIX model, do buy a fair amount of simplicity: Condor's designers were able to implement its process migration *without modifying the kernel.*

In motivation, Sprite seems to be a cross between Condor and MOSIX. Like MOSIX, Sprite strives for a very pure migration model—one in which the semantics of the process are almost exactly the same as if the process had been running locally. IPC channels, signal semantics, and error transparency are all important to the Sprite design. However, their migration policy is much more akin to Condor's. Similar to Condor, they seem primarily motivated by the desire to gracefully evict processes from machines which are no longer idle. When processes are first created with *exec()*, they are migrated to idle workstations if possible; later, they are migrated again only if the workstation owner returns and evicts the process. Unlike MOSIX, Sprite has no desire to dynamically re-balance the load on systems once processes have been assigned to processors.

5.2 Specific Design Decisions

In this section, we will explore some of the specific design decisions of the three systems in more detail.

User Space vs. Kernel Implementation. Sprite and MOSIX both involve extensive modifications to their respective kernels to support process migration. Amazingly, Condor is a process migration scheme that is implemented in user-space. Although no source code changes are necessary, users do need to link their programs with Condor's process migration library. The library intercepts certain system calls in cases where it needs to record state about the system call. The library also sets up a signal handler so that it can respond to signals from daemons running on the machines, telling it to checkpoint itself and terminate.

Centralized vs. Distributed Control. Condor and Sprite both rely on a centralized controller, which limits those systems' scalability and introduces a single point of failure for the entire system. In contrast, MOSIX nodes are all autonomous, and each uses a novel probabilistic information distribution algorithm to gather information from a randomly selected (small) subset of the other nodes in the system. This makes MOSIX much more scalable, and its completely decentralized control makes it more robust in the face of failures.

Filesystem Model. Condor does not assume that filesystems available on a process' home machine are also available on the target machine when a process migrates. Instead, it forwards all filesystem requests back to the home machine, which fulfills the request and forwards the results back to the migration target. In contrast, Sprite has a special cluster-wide network file system; it can assume that the same filesystem is available on every migration target. Sprite forwards the state of open files to the target machines and filesystem requests are carried out locally.[2] Similar to Sprite, MOSIX assumes that the same filesystem will be globally available, but MOSIX uses standard NFS.

Fully Transparent Execution Location. MOSIX and Sprite support what might be called "full transparency": in these systems, the process still appears to be running on the home node regardless of its actual execution location. The process itself always thinks that it's running on its home node, even if it migrated and is actually running on some other node. This has several important side effects. For example, IPC channels such TCP/IP connections, named pipes, and the like, are maintained in MOSIX and Sprite despite migration of processes on either end of the IPC channel. Data is received at the communications endpoint—i.e., a process' home node—where it is then forwarded to the node on which the process is actually running. In contrast, Condor does not support such a strong transparency model; a Condor process that migrates appears to be running on the migration target. For this reason, migration of processes that are involved in IPC is not allowed.

Migration Policies. As mentioned earlier, MOSIX attempts to dynamically balance the load continuously throughout the lifetime of all running processes, in an attempt to maximize the overall CPU utilization of the cluster. Sprite schedules a process to an idle processor once, when the process is born, and migrates that process back to its home node if the foreign node's owner returns. Once an eviction has occurred, Sprite does not re-migrate the evicted process to another idle processor. Condor falls somewhere in between these two. Like Sprite, Condor assigns a process to an idle node when the jobs is created. However, unlike Sprite, Condor attempts to find another idle node for the process every time it gets evicted. Absent of evictions, Condor does not attempt to dynamically re-balance the load as MOSIX does.

Checkpointing Capability. The mechanics of Condor's process migration implementation are such that the complete state of the process is written to disk when a migration occurs. After the process state is written to disk, the process is terminated, the state transferred to a new machine, and the process reconstructed. This implementation has a useful side effects. For example, the process state file can be saved. Saving it has the effect of "checkpointing" the process, so that it can be restarted from a previous point in case of a hardware failure or other abnormal termination. The frozen process can also be *queued*; who is to say that the frozen process has to be restarted immediately? The state file, with its process in stasis, can be kept

[2]This is *usually* but not always true. In some cases where there is a potential for concurrency, filesystem requests are serialized through the process' home machine to ensure consistency.

Table 1: Comparison of process migration schemes

	MOSIX	**Condor**	**Sprite**
Area of Implementation	Kernel	User Space	Kernel
Control Algorithms	Distributed	Centralized	Centralized
Filesystem Model	State of open files transferred; all nodes have same view of filesystem	Requests forwarded to home node; results forwarded back to remote node	State of open files transferred; all nodes have same view of filesystem
Full Transparency (IPC, signals, etc.)	Yes	No	Yes
Migration Policies	Continuous; dynamic	Assign on each eviction	Assign once; return if evicted
Checkpointing Capability	No	Yes	No

indefinitely—perhaps waiting for another idle processor to become available before restarting. The MOSIX and Sprite implementations are generally memory-to-memory and preclude these interesting possibilities.

6 Conclusions

Process migration, or the technique of moving a process from one computer to another during execution, can be a very useful load-balancing techniques in some computing environments. Of course, the actual speedups seen in practice depend on the nature of the application load. In situations where pools of processors are reserved for one job, and the optimal assignment of processes to processors can be determined *a priori*, process migration does not provide much benefit. However, if jobs arrive in an unpredictable or *ad hoc* nature, static scheduling can lead to long-term load imbalances. In these cases, process migration can lead to significant speedups.

There have been various implementations of process migration, some of which have been covered by this paper. The three systems reviewed—Condor, MOSIX, and Sprite—cover a range of design decisions and underlying assumptions. Our analysis of these systems should give the reader a sense of the issues and complexities involved in a successful process migration system.

References

[1] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 Usenix Conference*, pages 119–130, June 1985.

[2] Hal Stern. *Managing NFS and NIS.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991.

[3] George A. Champine, Daniel E Geer Jr., and William N. Ruh. Project Athena as a distributed computer operating system. *IEEE Computer*, 23(9):40–51, September 1990.

[4] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[5] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 49–70, October 1983.

[6] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 5–12, Austin TX (USA), November 1987. ACM.

[7] Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Catalin Rosu, and Ray Strong. Efficient message passing interface (MPI) for parallel computing on clusters of workstations. Technical Report CORNELLCS//TR95-1474, Cornell University, Computer Science Department, February 08 1995.

[8] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. A user's guide to PVM parallel virtual machine. Technical Report UT-CS-91-136, Department of Computer Science, University of Tennessee, July 1991. Tue, 1 Apr 97 18:13:17 GMT.

[9] N. Carriero and D. Gelernter. The S/Net's Linda Kernel. *ACM Transactions on Computer Systems*, 4(2):110–129, May 1986.

[10] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* New York: McGraw-Hill, 1993.

[11] D.B. Terry, M. Painter, D.W. Riggle, and S. Zhou. The berkeley internet domain server. In *Proceedings of the Summer 1984 USENIX Conference*, pages 23–31, June 1984.

[12] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4–5):361–372, March 1998.

[13] Fred Douglis and John K. Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software—Practice and Experience*, 21(8):757–785, August 1991.

[14] Allan Bricker, Michael Litzkow, and Miron Livny. Condor technical report. Technical Report TR 1069, Computer Sciences Department, University of Wisconsin-Madison, January 1992.