Process Migration in Distributed Systems

Shek Lun Ho
Computer Science Department
University of Southern California
Los Angeles, CA90007
shekho@usc.edu

Process Migration is the ability of a system (operating system or user-space program) to transfer processes between different nodes in a network. The motivations behind process migration are to balance load, improve availability, enhance communication performance and ease system administration[13]. However, since a process involves and interacts with many different components of an operating system, migration of a process is technically complicated and demands a great deal of researches.

This paper presents key concepts of process migration, design and implementation issues, and illustrates these concepts by several case studies on the typical implementations – MOSIX, Sprite, Condor and V.

---

1. Introduction

Process Migration has been in the computer science literature for more than two decades. One of the earliest concept papers introduces programs that can span machine boundaries in search of free machine for executions as "worms"[3]. Subsequent papers refine the concept of process migration as a potential mean to share processing power and other resources among multiple processors. Numerous papers and implementations have been proposed to introduce process migration on different architectures such as symmetric multiprocessors, Non-Uniform Memory Access (NUMA) multiprocessor, Massively Parallel Processors (MPP) and Local Area Network (LAN) of computers.

A process is an operating system (OS) entity of a program in execution. Associated with it are address space and other OS internal attributes such as home directory, open file descriptors, user id, program counter and so on. Process migration is defined as the transfer of a process between different nodes connected by a network.

The motivations for process migration are [1,4,13]:

- *Dynamic Load Balancing.* It allows processes to take advantage of less loaded

nodes by migrating from overloaded ones.

- *Availability.* Processes reside on a failure node can be migrated to other healthy nodes.
- *System Administration.* Processes that reside on a node that is to be undergone system maintenance can be migrated to other nodes.
- *Data Locality.* Processes can take advantage of locality of data or other special capabilities of a particular node.
- *Mobility.* Processes can be migrated from a handheld device or laptop computer to a more powerful server computer before the device get disconnected from the network.
- *Fault Recovery.* The mechanism to halt, transport and resume a process is technically useful to support fault recovery in mission-critical or transaction-based applications[11,12].

Various works have been done on the realization of process migration, including MOSIX[5], Sprite[6,7], Charlotte[8], V[9], Condor[11,12] and Mach[10]. Although process migration is useful in many contexts, it is not widely deployed nowadays. One of the problems is the complexity to support process migration on top of a system that does not have supporting facilities in design[1]. These facilities include network transparency, naming transparency, location transparency and others. Implementing process migration on systems that do not have relevant facilities may lead to degradation in performance and security, complicated implementation and poor reliability.

This paper is organized as follows. Section 2 presents the key concepts and characteristics of process migration as a basis for discussion thereafter. Section 3 presents case studies of important process migration works. Section 4 provides a brief comparison between different approaches and implementations of process migration. The final section gives the status of current works, future direction and a brief summary of process migration.

2. Overview of Process Migration

In this section, we present an overview of process migration.

2.1 System Support

Although process migration can be classified into different implementation categories, such as UNIX (or variant) transparent migration, microkernel, message-passing kernel and user-space migration, based on the level of operating system it is built upon, all implementations tend to provide a common subset of system functionalities in order to support process migration effectively.

1. *Network and Naming Transparencies[1].* Process migrated to the destination node should be able to execute as if it is on the source node. The communication channels, I/O devices, file systems should be readily accessible to the migrated process. In order to provide the execution context, certain degree of network and naming transparencies are supported. For example, MOSIX uses a super-root, "/…", as a network-wide root to provide a uniform mechanism to access objects[5].
2. *Interfaces to Export and Import Process States[1].* In order to guarantee the migrated process can execute correctly at the remote node, certain process states needs to be extracted from the source node and imported to the remote node. These states include program counter (PC), files handles, CPU registers and others. Heterogeneous process migration may need another level of abstraction or translation to be meaningful to the target architecture.
3. *Process Transfer Mechanism.* Different implementations use different process transfer approaches. The choices include remote procedure calls (RPC), message passing (MP).
4. *Load Information Management (Optional)[1].* Load Information Management components exist in many implementations (e.g. Condor[11]) to make sensible decision on process migration based on the information disseminated from the local and remote nodes.

2.2 Migration Mechanism

Depending on the particular system goals for migration, different systems implement migration in slightly different ways. One notable example is LOCUS transfers the entire virtual address space for the migrated process in trade of implementation simplicity, while Accent takes lazy-copying approach where pages are retrieved from the source machine as page faults occur on the target node for minimal initial migration performance impact. Despite of the slight derivations of these approaches, all of them exhibit a high degree of similarity in the mechanism (See Fig 1).

A description of process migration implementation is summarized as follows[1,5].

1. *Source node issues a migration request to the destination node.* Process-specific information is sent alongside with the request.

2. *The to-be-migrated process is removed from its execution context.* The to-be-migrated process is suspended for states extraction as described in step 4.

3. *"Communication is temporarily redirected* by queuing up arriving messages directed to the migrated process, and by delivering them to the process after migration."[1]

4. *States of the process is extracted.* Process states, such as address space,

communication channels, processor states and others, are extracted. Some of these states may reside on the source node after migration depending on implementations. State translation or abstraction may be needed if the target is of different architecture (e.g. Tui[13]).

5. *An empty process is instantiated on the destination node.* Transferred process state is to be imported. The process is not activated until there are enough process states.
6. *Process state is imported into the new instance.*
7. *Pending messages are forwarded.* Messages received since step 3 are forwarded to the new instance to maintain the communication channel. Depending on implementations, there may exist residual dependencies where some process states are intentionally left on the source and/or intermediate nodes involved in the transfer process.
8. *The new process instance is activated.* The new instance is restarted and starts execution.

The following diagram is copied from D. S. Milojicic et al [1].

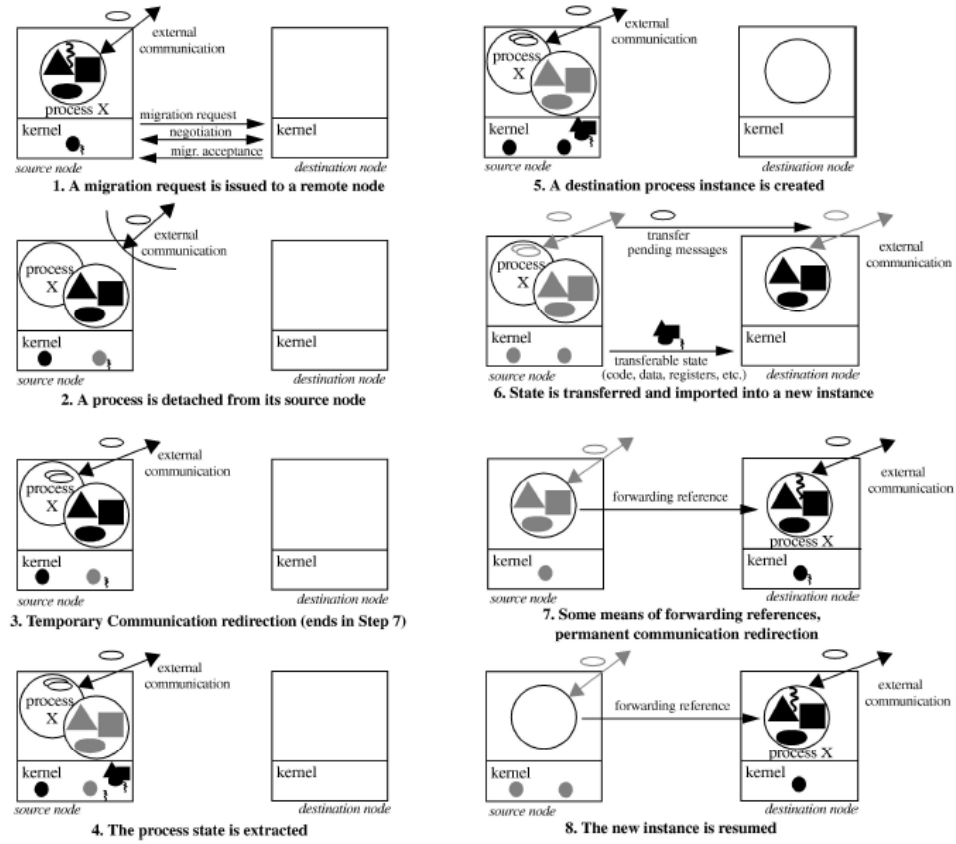

Figure 1. Migration Mechanism.

2.3 Migration Policy

As we see from the preceding section, process migration incurs a considerable amount of performance overhead and network traffic. Moreover, there exist possibilities that some of the message interactions with the migrating process initiated by a migration-unaware application may experience a timeout and thus pose an adverse effect on the semantic transparency of such an application. Therefore, we need some policies to guide and justify the process migration decision.

1. *Criteria to Choose a Process (What)[6].* Studies in UNIX load patterns show that "More than 78% of processes have lifetimes of less than 1 second"[14] and "97% of processes have a lifetime of less than 8 seconds" [14] suggest that "costs involved with migrating short-lived processes can outweigh the migration benefits"[1].
2. *Time to Initiate a Migration (When)[6].* Two schemes are widely adopted: periodic or event-based. In general, the lower the cost of migration, the more frequent the load information is disseminated[1].
3. *Criteria to Choose a Destination Node (Where)[6].* Several schemes are used depending of the system goals – heuristic (or random) algorithm, periodic load information dissemination, user-specified host and others.
4. *Entity to Make the Decision (Who)[6].* Depending on the distributed scheduling policies used, several schemes are possible. For systems like V, user can choose any remote machine for execution of a program by specifying the machine name. Other systems like Sprite, it has a central migration server that makes the decision based on the load information reported by the load-average daemon executing in every Sprite machine.

2.4 Characteristics

In designing a process migration facility, we need to consider numerous characteristics of how such a design affect the system as a whole. These characteristics have important impacts on the deployment of process migration.

The main ideas of the following descriptions are taken from D S Milojicic et al [1].

1. *Implementation Complexity.* Different levels of implementations yield different degree of complexity in implementation. Process migration can be implemented at the user-space, within the kernel, in a messaging-passing system, in an RPC-based or in a UNIX-like environment. Generally, user-space implementation yields a simpler design and implementation but leads to poorer performance. User-space implementations often exist as a run-time library and require applications to re-compile and re-link in order to take advantages of the features. However, user-space level implementation is at a better position to make decisions on the migration

policies than other implementations. Kernel-based implementations yield better performance, but it is more complicated to implement (e.g. transparency) and maintain.

2. *Performance Overhead.* Process migration involves performance overhead in the process state transfer. The overhead can be classified as initial cost and run-time cost. Some implementations that transfer the entire process state (e.g. LOCUS, Charlotte) incur only initial cost. Another extreme is that some implementations postpone state transfer until page faults occur (e.g. Accent). This type of implementation incurs run-time penalty only. Other possibilities that go between the two extremes (e.g. Sprite) incur both initial and run-time cost. Runtime cost can also be appeared as a consequence of residual dependencies that some process states are left on the source and/or intermediate nodes and require continual computation or forwarding facilities from these nodes.
3. *Naming/Location Transparency.* Process migration requires the migrated process to execute as on the source node. Otherwise, a migrated process may fail to execute when there is a discrepancy in the mechanism it access resources as on the source node. Different levels of transparency are implemented in different systems. For example, Sprite associates a home directory to all processes that executes host-specific code. Some location transparencies, however, are inevitably difficult to maintain in cases like accessing a local device.
4. *Fault Tolerance.* Any exceptions occur in the process of migration, such as network partition and node failure, may severely affect the reliability of the system. Design issues such as reduction of residual dependency may improve the fault tolerance.
5. *Scalability.* Scalability takes 3 forms: Numeric, geographic and administrative, and it is affected by the design choices chosen. For example, process states accumulate as the number of migration increases in Mach[10] and hence adversely affect the scalability. Other consideration issues include residual dependencies, granularity of the transfer segment, virtual memory transfer techniques[6], etc..

2.4 Alternatives

As process migration incurs significant overhead and requires sophisticated system level functionalities, several alternatives are more feasible in certain scenarios.

1. *Remote Execution[6].* Remote execution is used to invoke some code on a remote machine. The semantic is well understood, simple to implement and avoid most of the overhead incurred in process migration. However, it doesn't not provide any kind of transparencies, provide no mechanisms to evict the remote process and decision is made not based on usage.
2. *Middleware[1,15].* Middleware such as CORBA provides an excellent abstraction of the

underlying hardware/software architecture. Thus, making heterogeneous object invocation much simpler to implement.

2.5 Categories of Process Migration

Process migration has can be implemented on different levels of a system (See references). The level and system in which process migration is implemented greatly affects the design choices: some designs are only feasible at a particular level in certain system architectures. In this section, we divide the levels/systems into four categories and examine the corresponding design choices.

1. *UNIX-like Kernel*[1]. As naming and location transparencies are not fully supported in UNIX, transparent process migration requires significant modification to the monolithic kernel. Examples are LOCUS and Sprite.
2. *Message-Passing Kernel*[1]. As messages can be easily redirected between the sender and the receiver, it facilitates the change of communication endpoints in process migration. For example, System V.
3. *Microkernel*[1]. Microkernel builds process migration facilities on top of the operating system as a separate module. As transparencies are often available in these systems and message passing communication mechanism is utilized, process migration seems to be relatively easily to implement. Mach is a typical example.
4. *User-Space Migration*[1]. As user-space migration does not require a modification of the kernel, it implies that the entire address space must be extracted and transferred to rebuild it at the destination node. Moreover, processes that involve signal processing, shared libraries or IPC are usually unable to be migrated.

3. Case Studies

In this section, we examine several typical process migration implementations in the research area. Most of the implementations are used as a test bed for distributed computing, and quite a few of them are commercialized.

The systems we discuss are MOSIX, Sprite, Condor and V. The design goals, mechanisms and implications are discussed.

3.1 MOSIX

“MOSIX is a general-purpose Multicomputer Operating System which Integrates a cluster of loosely connected, independent computers (nodes) into a single-machine UNIX environment. The main properties of MOSIX are its high degree of integration and the possibility of scaling the configuration to a large number of nodes”[5]

The design goals of MOSIX are:

1. *Single System Image*[5]*.* It provides a single view of the file system and network transparency is provided at the user level.
2. *Autonomy*[5]*.* Kernel is replicated in each processor and thus autonomic. It makes its own control decision independent of other nodes.
3. *Scalability*[5]*.* Probabilistic algorithms are used to minimize system management and network overhead.
4. *Dynamic Configuration*[5]*.* Node may join and leave a cluster at will without significant adverse effect.

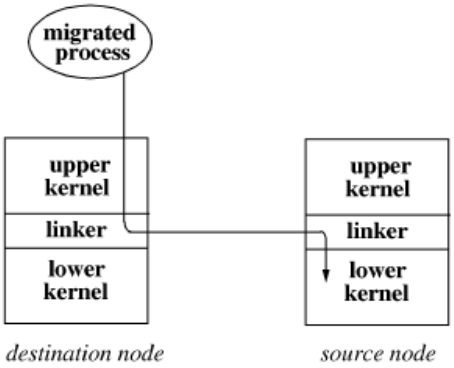

Figure 2. The MOSIX architecture

The following descriptions referenced and summarized from [1,5].

MOSIX organizes it kernel into three layers: the upper kernel, linker and lower kernel (see figure 2). The lower-kernel is machine-dependent and operates independently to provide normal services such as access to local disks devices, context-switching and others. The machine-independent upper kernel provides standard UNIX system call interface and has a complete knowledge about the location of all objects it handles. The linker provides inter-node communication, data transfer, process migration and load balancing algorithms. By this way, the upper kernel provides an abstraction of the execution context with location transparency to the process running above it (see Design Goals 1).

Process migration in MOSIX is cooperative in the sense that the source and destination nodes cooperate to make a migration decision. During migration, only dirty pages and user areas of the migrating process are transferred while clean pages are “paged-in” whenever there is a page fault at the destination node. In other words, it keeps residual dependencies. Moreover, a process in MOSIX is site-independent that each system call gets routed to the appropriate node by the linker and lower kernel. Thus, process that involves signal handling, memory-mapped files manipulation and other host-specific

application can be easily migrated.

3.2 Sprite

Sprite is an operating system to provide a network of personal workstations act as a time-sliced system. “Each host runs a copy of the Sprite kernel and work closely together using a remote-procedure-call (RPC) mechanism”[6]. Processes in Sprite can access files or devices on any host, and data can be cached across the network while consistency is maintained with the one-copy semantic.

The design goals process migration in Sprite are:

1. *Utilize Idle Hosts[6].* Idle hosts are plentiful even at the busiest time of the day. Utilization of these otherwise wasted computing resources can give a boost in performance to other loaded hosts.
2. *Exclusive Use of Workstations by their owner[6].* Computer resources privilege is given to the local user. Migrated processes are evicted back to their source nodes whenever the workstation owner log into his/her machine.
3. *Kernel RPC[6].* Sprite uses protected kernel RPC as a form of interprocess communication. Redirection of communication channels is not as obvious as it is in the message-passing kernel.

Sprite associates each process with its designated home machine, regardless of its physical location. To provide a location transparency abstraction to the migration process, host-specific system calls are forwarded to the shadow process on the host machine via kernel-to-kernel RPCs. Other calls, such as memory allocation and the distributed file system-related calls, are handled locally. The migrated process get evicted once the owner returns back to the remote machine.

3.3 System V

System V is a distributed operating system that is run on a cluster of networked workstations. Every host runs a small identical kernel plus some service modules and run-time libraries[16]. The implementation scheme of V provides transparency, minimal interfaces and residual dependencies on the source host[2].

The design goals V’s process migration are:

1. *Network Transparency[9].* Network transparency is implemented with the use of V IPC primitives and global naming to provide communication channels between a program and the operating system.
2. *Minimal Interference to the system[9].* Process migration is regarded as an atomic transfer of process states between two hosts without interfering other parts of the

system.

3. *No Residual Dependencies*[9]*.* Process states are not left over on the source node once the process migration has been committed to "protect programs executing on behalf of different (remote) users on the same workstation."[9]

System V organizes process and its address space into logical hosts identified by a structure (logical-host-id, local-index). Process migration is done by the migration of the logical host in which the program is running on. V uses a special mechanism to transfer the address space of the migrated process – precopy. It iteratively performs precopy until the number of pages fall below a threshold during the actual migration. Precopy effectively reduces the "freezing" time of a process in contrast to Charlotte or LOCUS. The net effect is that it can improve the triggering time for critical operations, such as timed system calls, which has a fixed timeout value. System V solves the residual dependencies problem by requiring that the execution context of a program is either resided in its address space or in a globally accessible server.

3.4 Condor

Condor is a software package that provides identification of idle machines in a network and offloads processes to those machines.

The design goals of process migration in Condor are:

1. *Maximize Computation Utilization.* Idle workstations are potential destinations for process migration because otherwise the computation cycles may be wasted.
2. *No Modification to Kernel.* UNIX systems are proprietary and access to the internals of the system is not possible at the time Condor is proposed.

Condor is implemented as a software package for long-running, computation-intensive jobs[11]. The way Condor extracts and imports process states is through the use of core-dump facility (check pointing) provided by traditional UNIX system. However, it has some inherent limitations such as the inability to restore process states like signals and timers, lack of support of inter-process communication, inconsistency between different core dump file formats and performance.

4. Comparison

This section compares the systems described in the previous section, i.e. MOSIX, Sprite, Condor and V.

This table is summarized from D S Milojicic et al [1], A Barak et al [5], F Douglis et al [6,7], M M Theimer et al [9], D S and M Litzkow et al [11,12]

	MOSIX	V	Sprite	Condor
Category	UNIX-like	Message-passing/Microkernel	UNIX-like	User-space
Goals	• Dynamic Process Migration • Single System Image • Autonomy • Dynamic Configuration • Scalability	• Network Transparency • Minimal Interference • No Residual Dependencies	• Autonomy • Location Transparency • Using Idle Cycle • Simplicity	• Maximize Utilization • No Modification to Kernel
Extensibility	Fair	Fair	Fair	Very Good
Transparency **1. (Open Files)** **2. (Fork)** **3. (Communication Channel)**	Full (Yes) (Yes) (Yes)	Full (Yes) (Yes) (Yes)	Full (Yes) (Yes) (Yes)	Limited No signals, timers, memory-mapped files and shared libraries
Transfer Strategy	Dirty Pages	Precopy	Flushing	Entire Address Space
Freeze Time	Moderate	Very Low	Moderate	High
Residual Dependency	No	No	No	No
Load information	Distributed	Unknown	Centralized	Combination of both
Performance	Good	Good	Good	Poor

Table 1. Comparison of process migration implementations

5. Future Researches & Summary

Although process migration offers some promising benefits, it is not widely deployed in the academia and the industry because of a variety of reasons. These obstacles include:

1. *Lack of Infrastructure*[2]*.* An overwhelming success of process migration demands the support of an infrastructure that allows different machine hardware/software architectures to participate and also the mobility of these devices. However, although some of the individual empowering technologies (like mobile IP, Java) are catching on, a mature infrastructure does not readily exist.
2. *Complexity.* Process migration involves many different research areas in computer science, a coherent solution that meets almost all the design goals among all areas is difficult to attain without loss of generosity.

With the successful deployment of the commercial Internet, more and more computing resources are available and connected. One of the prominent trends in the research of process mobility is the Mobile Agent. Mobile agent resembles process migration that both allow a task (or a process abstraction) to be migrated and executed on a different node. The major different is that process migration is often deployed on a cluster of relatively tightly coupled machines where mobile agent is often deployed in the scale of the Internet. The implication is that security and reliability are becoming a more dominant issue in the design such a mobile agent system.

In this paper we have presented some background information of process migration, such as motivations, mechanisms, policies, characteristics and others. Then we examine and compare several existing implementations, namely MOSIX, Sprite, Condor and V and how they address the design and implementation tradeoffs of process migration. At last, we address why process migration does not get widely adapted and point out direction for future researches.

References

1. Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration.
2. Dejan Milojicic, Frederick Douglis, and Richard Wheeler. Mobility, Processes, Computers, and Agents.
3. John F. Shoch, and Jon A. Hupp. The “Worm” Programs – Early Experience with a Distributed Computation.
4. Michael L. Powell, and Barton P. Miller. Process Migration in DEMOS/MP.
5. Amnon Barak and Richard Wheeler. MOSIX: An Integrated Multiprocessor UNIX.
6. Fred Douglis and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation.
7. Frederick Douglis. Sprite Process Migration: a Retrospective.
8. Yeshayahu Artsy and Raphael Finkel. Designing a process migration facility: the Charlotte experience.
9. Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System.
10. Dejan S Milojicic, Wolfgang Zint, Andreas Dangel and Peter Giese. Task Migration on the top of the Mach Microkernel.
11. Michael Litzkow, and Marvin Solomon. Supporting checkpointing and Process Migration outside the UNIX Kernel.
12. Michael litzkow, and Marvin Solomon. The Evolution of condor checkpointing.
13. Peter Smith, and Norman C. Hutchinson. Heterogeneous Process Migration: The Tui

System.
14. Luis-Felipe Cabrera. The Influence of workload on Load balancing Strategies.
15. Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments.
16. David R Cheriton. The V Distributed System.