Process Migration

DEJAN S. MILOJÍČIĆ

HP Labs

FRED DOUGLIS

AT&T Labs–Research

YVES PAINDAVEINE

TOG Research Institute

RICHARD WHEELER

EMC

AND

SONGNIAN ZHOU

University of Toronto and Platform Computing

Process migration is the act of transferring a process between two machines. It enables dynamic load distribution, fault resilience, eased system administration, and data access locality. Despite these goals and ongoing research efforts, migration has not achieved widespread use. With the increasing deployment of distributed systems in general, and distributed operating systems in particular, process migration is again receiving more attention in both research and product development. As high-performance facilities shift from supercomputers to networks of workstations, and with the ever-increasing role of the World Wide Web, we expect migration to play a more important role and eventually to be widely adopted.

This survey reviews the field of process migration by summarizing the key concepts and giving an overview of the most important implementations. Design and implementation issues of process migration are analyzed in general, and then revisited for each of the case studies described: MOSIX, Sprite, Mach, and Load Sharing Facility. The benefits and drawbacks of process migration depend on the details of implementation and, therefore, this paper focuses on practical matters. This survey will help in understanding the potentials of process migration and why it has not caught on.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*network operating systems*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; D.4.8 [**Operating Systems**]: Performance—*measurements*; D.4.2 [**Operating Systems**]: Storage Management—*distributed memories*

General Terms: Design, Experimentation

Additional Key Words and Phrases: Process migration, distributed systems, distributed operating systems, load distribution

1. INTRODUCTION

A process is an operating system abstraction representing an instance of a running computer program. Process migration is the act of transferring a process between two machines during its execution. Several implementations have been built for different operating systems, including MOSIX [Barak and Litman, 1985], V [Cheriton, 1988], Accent [Rashid and Robertson, 1981], Sprite [Ousterhout et al., 1988], Mach [Accetta et al., 1986], and OSF/1 AD TNC [Zajcew et al., 1993]. In addition, some systems provide mechanisms that checkpoint active processes and resume their execution in essentially the same state on another machine, including Condor [Litzkow et al., 1988] and Load Sharing Facility (LSF) [Zhou et al., 1994]. Process migration enables:

- **dynamic load distribution**, by migrating processes from overloaded nodes to less loaded ones,
- **fault resilience**, by migrating processes from nodes that may have experienced a partial failure,
- **improved system administration**, by migrating processes from the nodes that are about to be shut down or otherwise made unavailable, and
- **data access locality**, by migrating processes closer to the source of some data.

Despite these goals and ongoing research efforts, migration has not achieved widespread use. One reason for this is the complexity of adding transparent migration to systems originally designed to run stand-alone, since designing new systems with migration in mind from the beginning is not a realistic option anymore. Another reason is that there has not been a compelling commercial argument for operating system vendors to support process migration. Checkpoint-restart approaches offer a compromise here, since they can run on more loosely-coupled systems by restricting the types of processes that can migrate.

In spite of these barriers, process migration continues to attract research. We believe that the main reason is the potentials offered by mobility as well as the attraction to hard problems, so inherent to the research community. There have been many different goals and approaches to process migration because

of the potentials migration can offer to different applications (see Section 2.3 on goals, Section 4 on approaches, and Section 2.4 on applications).

With the increasing deployment of distributed systems in general, and distributed operating systems in particular, the interest in process migration is again on the rise both in research and in product development. As high-performance facilities shift from supercomputers to Networks of Workstations (NOW) [Anderson et al., 1995] and large-scale distributed systems, we expect migration to play a more important role and eventually gain wider acceptance.

Operating systems developers in industry have considered supporting process migration, for example Solaris MC [Khalidi et al., 1996], but thus far the availability of process migration in commercial systems is non-existent as we describe below. Checkpoint-restart systems are becoming increasingly deployed for long-running jobs. Finally, techniques originally developed for process migration have been employed in developing mobile agents on the World Wide Web. Recent interpreted programming languages, such as Java [Gosling et al., 1996], Telescript [White, 1996] and Tcl/Tk [Ousterhout, 1994] provide additional support for agent mobility.

There exist a few books that discuss process migration [Goscinski, 1991; Barak et al., 1993; Singhal and Shivaratri, 1994; Milojičić et al., 1999]; a number of surveys [Smith, 1988; Eskicioglu, 1990; Nuttal, 1994], though none as detailed as this survey; and Ph.D. theses that deal directly with migration [Theimer et al., 1985; Zayas, 1987a; Lu, 1988; Douglis, 1990; Philippe, 1993; Milojičić, 1993c; Zhu, 1992; Roush, 1995], or that are related to migration [Dannenberg, 1982; Nichols, 1990; Tracey, 1991; Chapin, 1993; Knabe, 1995; Jacqmot, 1996].

This survey reviews the field of process migration by summarizing the key concepts and describing the most important implementations. Design and implementation issues of process migration are analyzed in general and then revisited for each of the case studies described: MOSIX, Sprite, Mach, and LSF. The benefits and drawbacks of process migration depend on the details of implementation and therefore this paper focuses on practical matters. In this paper we address mainly process migration mechanisms. Process migration policies, such as load information management and distributed scheduling, are mentioned to the extent that they affect the systems being discussed. More detailed descriptions of policies have been reported elsewhere (e.g., Chapin's survey [1996]).

This survey will help in understanding the potential of process migration. It attempts to demonstrate how and why migration may be widely deployed. We assume that the reader has a general knowledge of operating systems.

Organization of the Paper

The paper is organized as follows. Section 2 provides background on process migration. Section 3 describes the process migration by surveying its main characteristics: complexity, performance, transparency, fault resilience, scalability and heterogeneity. Section 4 classifies various implementations of process migration mechanisms and then describes a couple of representatives for each class. Section 5 describes four case studies of process migration in more detail. In Section 6 we compare the process migration implementations presented earlier. In Section 7 we discuss why we believe that process migration has not caught on so far. In the last section we summarize the paper and describe opportunities for further research.

2. BACKGROUND

This section gives some background on process migration by providing an overview of process migration terminology, target architectures, goals, application taxonomy, migration algorithms, system requirements, load information management, distributed scheduling, and alternatives to migration.

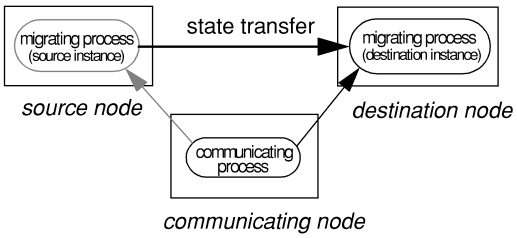

Fig. 1. High Level View of Process Migration. Process migration consists of extracting the state of the process on the source node, transferring it to the destination node where a new instance of the process is created, and updating the connections with other processes on communicating nodes.

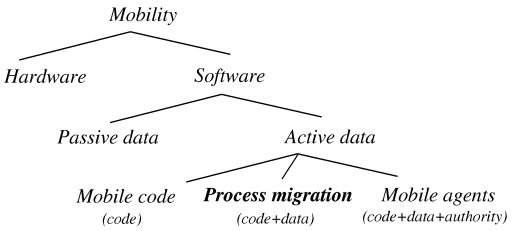

Fig. 2. Taxonomy of Mobility.

2.1. Terminology

A *process* is a key concept in operating systems [Tanenbaum, 1992]. It consists of data, a stack, register contents, and the state specific to the underlying Operating System (OS), such as parameters related to process, memory, and file management. A process can have one or more threads of control. Threads, also called lightweight processes, consist of their own stack and register contents, but share a process's address space and some of the operating-system-specific state, such as signals. The *task* concept was introduced as a generalization of the process concept, whereby a process is decoupled into a task and a number of threads. A traditional process is represented by a task with one thread of control.

Process migration is the act of transferring a process between two machines (the *source* and the *destination* node) during its execution. Some architectures also define a *host* or *home* node, which is the node where the process logically runs. A high-level view of process migration is shown in Figure 1. The transferred state includes the process's address space, execution point (register contents), communication state (e.g., open files and message channels) and other operating system dependent state. *Task migration* represents transferring a task between two machines during execution of its threads.

During migration, two instances of the migrating process exist: the *source instance* is the original process, and the *destination instance* is the new process created on the destination node. After migration, the destination instance becomes a *migrated process*. In systems with a home node, a process that is running on other machines may be called a *remote* process (from the perspective of the home node) or a *foreign* process (from the perspective of the hosting node).

Remote invocation is the creation of a process on a remote node. Remote invocation is usually a less "expensive" operation than process migration. Although the operation can involve the transfer of some state, such as code or open files, the contents of the address space need not be transferred.

Generally speaking, mobility can be classified into hardware and software mobility, as described in Figure 2. Hardware mobility deals with mobile computing, such as with limitations on the connectivity of mobile computers and mobile IP (see [Milojičić et al., 1999] for more details). A few techniques in mobile computing have an analogy in software mobility, such as security, locating, naming, and communication forwarding. Software mobility can be classified into the mobility of passive data and active data. Passive data represents traditional means of transferring data between computers; it has been employed ever since the first two computers were connected. Active data can be further classified into mobile code, process migration and mobile agents. These three classes represent incremental evolution of state transfer. Mobile code, such as Java applets, transfers only code between nodes. Process migration, which is the main theme of this paper, deals primarily with code and data transfer. It also

deals with the transfer of authority, for instance access to a shared file system, but in a limited way: authority is under the control of a single administrative domain. Finally, mobile agents transfer code, data, and especially authority to act on the owner's behalf on a wide scale, such as within the entire Internet.

2.2. Target Architectures

Process migration research started with the appearance of distributed processing among multiple processors. Process migration introduces opportunities for sharing processing power and other resources, such as memory and communication channels. It is addressed in early multiprocessor systems [Stone, 1978; Bokhari, 1979]. Current multiprocessor systems, especially symmetric multiprocessors, are scheduled using traditional scheduling methods. They are not used as an environment for process migration research.

Process migration in NUMA (Non-Uniform Memory Access) multiprocessor architectures is still an active area of research [Gait, 1990; Squillante and Nelson, 1991; Vaswani and Zahorjan, 1991; Nelson and Squillante, 1995]. The NUMA architectures have a different access time to the memory of the local processor, compared to the memory of a remote processor, or to a global memory. The access time to the memory of a remote processor can be variable, depending on the type of interconnect and the distance to the remote processor. Migration in NUMA architectures is heavily dependent on the memory footprint that processes have, both in memory and in caches. Recent research on virtual machines on scalable shared memory multiprocessors [Bugnion, et al., 1997] represents another potential for migration. Migration of whole virtual machines between processors of a multiprocessor abstracts away most of the complexities of operating systems, reducing the migrateable state only to memory and to state contained in a virtual monitor [Teodosiu, 2000]. Therefore, migration is easier to implement if there is a notion of a virtual machine.

Massively Parallel Processors (MPP) are another type of architecture used for migration research [Tritscher and Bemmerl, 1992; Zajcew et al., 1993]. MPP machines have a large number of processors that are usually shared between multiple users by providing each of them with a subset, or partition, of the processors. After a user relinquishes a partition, it can be reused by another user. MPP computers are typically of a NORMA (NO Remote Memory Access) type, i.e., there is no remote memory access. In that respect they are similar to network clusters, except they have a much faster interconnect. Migration represents a convenient tool to achieve repartitioning. Since MPP machines have a large number of processors, the probability of failure is also larger. Migrating a running process from a partially failed node, for example after a bank of memory unrelated to the process fails, allows the process to continue running safely. MPP machines also use migration for load distribution, such as the *psched* daemon on Cray T3E, or Loadleveler on IBM SP2 machines.

Since its inception, a Local Area Network (LAN) of computers has been the most frequently used architecture for process migration. The bulk of the systems described in this paper, including all of the case studies, are implemented on LANs. Systems such as NOW [Anderson et al., 1995] or Solaris [Khalidi et al., 1996] have recently investigated process migration using clusters of workstations on LANs. It was observed that at any point in time many autonomous workstations on a LAN are unused, offering potential for other users based on process migration [Mutka and Livny, 1987]. There is, however, a sociological aspect to the autonomous workstation model. Users are not willing to share their computers with others if this means affecting their own performance [Douglis and Ousterhout, 1991]. The priority of the incoming processes (processing, VM, IPC priorities) may be reduced in order to allow for minimal impact on the workstation's owner [Douglis and Ousterhout, 1991; Krueger and Chawla, 1991].

Most recently, wide-area networks have presented a huge potential for migration. The evolution of the Web has significantly improved the relevance and the opportunities for using a wide-area network for distributed computing. This has resulted in the appearance of mobile agents, entities that freely roam the network and represent the user in conducting his tasks. Mobile agents can either appear on the Internet [Johansen et al., 1995] or in closed networks, as in the original version of Telescript [White, 1996].

2.3. Goals

The goals of process migration are closely tied with the type of applications that use migration, as described in next section. The goals of process migration include:

Accessing more processing power is a goal of migration when it is used for load distribution. Migration is particularly important in the *receiver-initiated* distributed scheduling algorithms, where a lightly loaded node announces its availability and initiates process migration from an overloaded node. This was the goal of many systems described in this survey, such as Locus [Walker et al., 1983], MOSIX [Barak and Shiloh, 1985], and Mach [Milojičić et al., 1993a]. Load distribution also depends on load information management and distributed scheduling (see Sections 2.7 and 2.8). A variation of this goal is harnessing the computing power of temporarily free workstations in large clusters. In this case, process migration is used to evict processes upon the owner's return, such as in the case of Sprite (see Section 5.2).

Exploitation of resource locality is a goal of migration in cases when it is more efficient to access resources locally than remotely. Moving a process to another end of a communication channel transforms remote communication to local and thereby significantly improves performance. It is also possible that the resource is not remotely accessible, as in the case when there are different semantics for local and remote accesses. Examples include work by Jul [1989], Milojičić et al. [1993], and Miller and Presotto [1981].

Resource sharing is enabled by migration to a specific node with a special hardware device, large amounts of free memory, or some other unique resource. Examples include NOW [Anderson et al., 1995] for utilizing memory of remote nodes, and the use of parallel *make* in Sprite [Douglis and Ousterhout, 1991] and work by Skordos [1995] for utilizing unused workstations.

Fault resilience is improved by migration from a partially failed node, or in the case of long-running applications when failures of different kinds (network, devices) are probable [Chu et al., 1980]. In this context, migration can be used in combination with checkpointing, such as in Condor [Litzkow and Solomon, 1992] or Utopia [Zhou et al., 1994]. Large-scale systems where there is a likelihood that some of the systems can fail can also benefit from migration, such as in Hive [Chapin et al., 1995] and OSF/1 AD TNC [Zajcew et al., 1993].

System administration is simplified if long-running computations can be temporarily transferred to other machines. For example, an application could migrate from a node that will be shutdown, and then migrate back after the node is brought back up. Another example is the repartitioning of large machines, such as in the OSF/1 AD TNC Paragon configuration [Zajcew et al., 1993].

Mobile computing also increases the demand for migration. Users may want to migrate running applications from a host to their mobile computer as they connect to a network at their current location or back again when they disconnect [Bharat and Cardelli, 1995].

2.4. Application Taxonomy

The type of applications that can benefit from process migration include:

Parallelizable applications can be started on certain nodes, and then migrated at the application level or by a system-wide migration facility in response to things like load balancing considerations. Parallel Virtual Machine (PVM) [Beguelin et al., 1993] is an example of application-level support for parallel

invocation and interprocess communication, while Migratory PVM (MPVM) [Casas et al., 1995] extends PVM to allow instances of a parallel application to migrate among nodes. Some other applications are inherently parallelizable, such as the *make* tool [Baalbergen, 1988]. For example, Sprite provides a migration-aware parallel *make* utility that distributes a compilation across several nodes [Douglis and Ousterhout, 1991]. Certain processor-bound applications, such as scientific computations, can be parallelized and executed on multiple nodes. An example includes work by Skordos [1995], where an acoustic application is parallelized and executed on a cluster of workstations. Applications that perform I/O and other nonidempotent operations are better suited to a system-wide remote execution facility that provides location transparency and, if possible, preemptive migration.

Long-running applications, which can run for days or even weeks, can suffer various interruptions, for example partial node failures or administrative shutdowns. Process migration can relocate these applications transparently to prevent interruption. Examples of such systems include work by Freedman [1991] and MPVM [Casas et al., 1995]. Migration can also be supported at the application level [Zhou et al., 1994] by providing a checkpoint/restart mechanism which the application can invoke periodically or upon notification of an impending interruption.

Generic multiuser workloads, for example the random job mix that an undergraduate computer laboratory produces, can benefit greatly from process migration. As users come and go, the load on individual nodes varies widely. Dynamic process migration [Barak and Wheeler, 1989, Douglis and Ousterhout, 1991] can automatically spread processes across all nodes, including those applications that are not enhanced to exploit the migration mechanism.

An individual generic application, which is preemptable, can be used with various goals in mind (see Section 2.3). Such an application can either migrate itself, or it can be migrated by another authority. This type of application is most common in various systems described in Section 4 and in the case studies described in Section 5. Note that it is difficult to select such applications without detailed knowledge of past behavior, since many applications are short-lived and do not execute long enough to justify the overhead of migration (see Section 2.7).

Migration-aware applications are applications that have been coded to explicitly take advantage of process migration. Dynamic process migration can automatically redistribute these related processes if the load becomes uneven on different nodes, e.g. if processes are dynamically created, or there are many more processes than nodes. Work by Skordos [1995], Freedman [1991] and Cardelli [1995] represent this class of application. They are described in more detail in Section 4.6.

Network applications are the most recent example of the potential use of migration: for instance, mobile agents and mobile objects (see Sections 4.7 and 4.8). These applications are designed with mobility in mind. Although this mobility differs significantly from the kinds of "process migration" considered elsewhere in this paper, it uses some of the same techniques: location policies, checkpointing, transparency, and locating and communicating with a mobile entity.

2.5. Migration Algorithm

Although there are many different migration implementations and designs, most of them can be summarized in the following steps (see also Figure 3):

1. **A migration request is issued to a remote node**. After negotiation, migration has been accepted.
2. **A process is detached from its source node** by suspending its execution, declaring it to be in a migrating state, and temporarily redirecting communication as described in the following step.

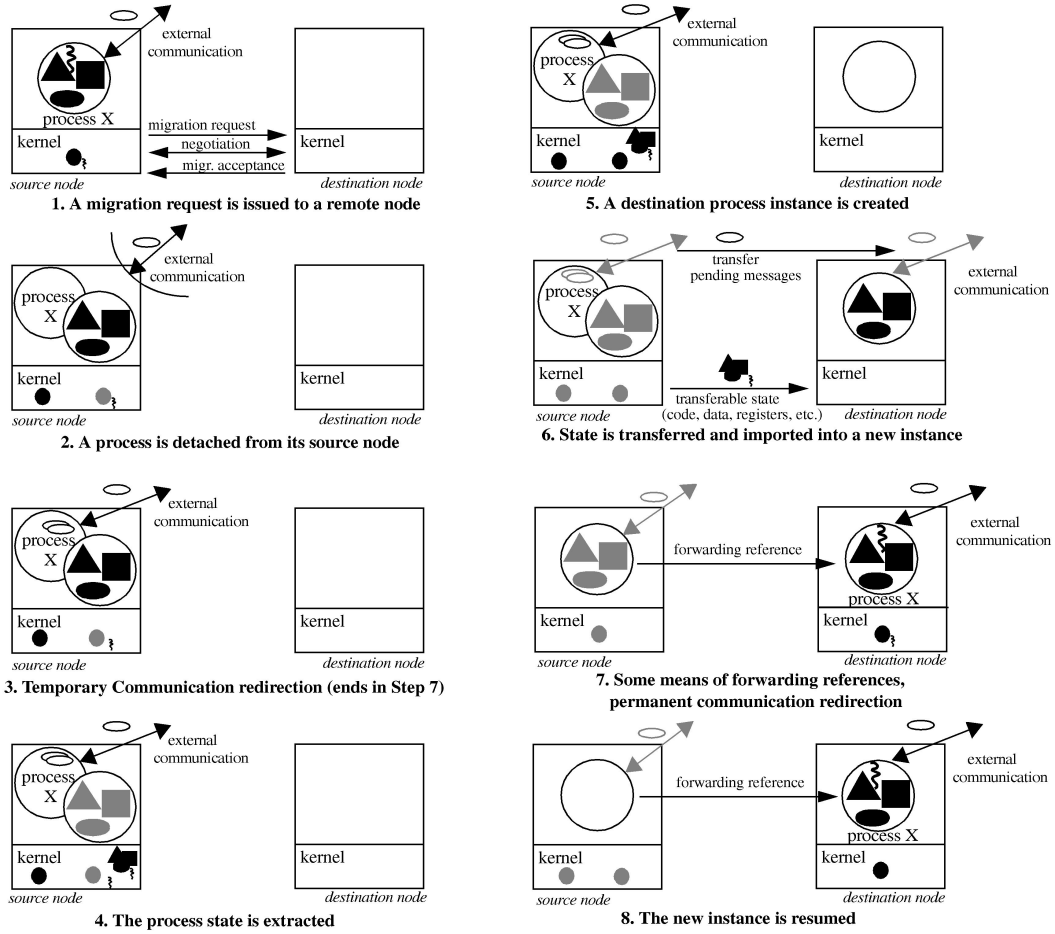

Fig. 3. Migration Algorithm. Many details have been simplified, such as user v. kernel migration, when is process actually suspended, when is the state transferred, how are message transferred, etc. These details vary subject to particular implementation.

3. **Communication is temporarily redirected** by queuing up arriving messages directed to the migrated process, and by delivering them to the process after migration. This step continues in parallel with steps 4, 5, and 6, as long as there are additional incoming messages. Once the communication channels are enabled after migration (as a result of step 7), the migrated process is known to the external world.
4. **The process state is extracted**, including memory contents; processor state (register contents); communication state (e.g., opened files and message channels); and relevant kernel context. The communication state and kernel context are OS-dependent. Some of the local OS internal state is not transferable. The process state is typically retained on the source node until the end of migration, and in some systems it remains there even after migration completes. Processor dependencies, such as register and stack contents, have to be eliminated in the case of heterogeneous migration.
5. **A destination process instance is created** into which the transferred state will be imported. A destination instance is not activated until a sufficient

amount of state has been transferred from the source process instance. After that, the destination instance will be promoted into a regular process.

6. **State is transferred and imported into a new instance** on the remote node. Not all of the state needs to be transferred; some of the state could be lazily brought over after migration is completed (see lazy evaluation in Section 3.2).
7. **Some means of forwarding references** to the migrated process must be maintained. This is required in order to communicate with the process or to control it. It can be achieved by registering the current location at the *home* node (e.g. in Sprite), by searching for the migrated process (e.g. in the V Kernel, at the communication protocol level), or by forwarding messages across all visited nodes (e.g. in Charlotte). This step also enables migrated communication channels at the destination and it ends step 3 as communication is permanently redirected.
8. **The new instance is resumed** when sufficient state has been transferred and imported. With this step, process migration completes. Once all of the state has been transferred from the original instance, it may be deleted on the source node.

2.6. System Requirements for Migration

To support migration effectively, a system should provide the following types of functionality:

- **Exporting/importing the process state.** The system must provide some type of export/import interfaces that allow the process migration mechanism to extract a process's state from the source node and import this state on the destination node. These interfaces may be provided by the underlying operating system, the programming language, or other elements of the programming environment that the process has access to. State includes processor registers, process address space and communication state, such as open message channels in the case of message-based systems, or open files and signal masks in the case of UNIX-like systems.
- **Naming/accessing the process and its resources.** After migration, the migrated process should be accessible by the same name and mechanisms as if migration never occurred. The same applies to process's resources, such as threads, communication channels, files and devices. During migration, access to a process and/or some of its resources can be temporarily suspended. Varying degrees of transparency are achieved in naming and accessing resources during and after migration (see Section 3.3).
- **Cleaning up the process's non-migratable state.** Frequently, the migrated process has associated system state that is not migratable (examples include a local process identifier, *pid*, and the local time; a local *pid* is relevant only to the local OS, and every host may have a slightly different value for the local time—something that may or may not matter to a migrating process). Migration must wait until the process finishes or aborts any pending system operation. If the operation can be arbitrarily long, it is typically aborted and restarted on the destination node. For example, migration can wait for the completion of local file operations or local device requests that are guaranteed to return in a limited time frame. Waiting for a message or accessing a remote device are examples of operations that need to be aborted and restarted on the remote node. Processes that cannot have their non-migrateable state cleaned cannot be considered for migration.

2.7. Load Information Management

The local processes and the resources of local and remote nodes have to be characterized, in order to select a process for migration and a destination node, as well as to justify migration. This task is commonly known as load information management. Load information is collected and passed to a distributed scheduling policy

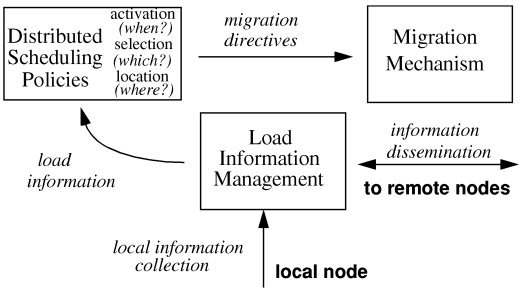

Fig. 4. Load Information Management Module collects load information on the local node and disseminates it among the nodes. **Distributed Scheduling** instructs the migration mechanism when, where, and which process to migrate.

(see Figure 4). Load information management is concerned with the following three questions:

What is load information and how is it represented? The node load is typically represented by one or more of the following load indices: utilization of the CPU, the length of the queue of processes waiting to be executed, the stretch factor (ratio between turnaround- and execution-time-submission to completion v. start to completion) [Ferrari and Zhou 1986], the number of running processes, the number of background processes, paging, communication [Milojičić, 1993c], disk utilization, and the interrupt rate [Hwang et al., 1982]. A process load is typically characterized by process lifetime, CPU usage, memory consumption (virtual and physical), file usage [Hac, 1989a], communication [Lo, 1989], and paging [Milojičić, 1993c]. Kuntz uses a combination of workload descriptions for distributed scheduling [Kunz, 1991]. The application type is considered in Cedar [Hagmann, 1986].

When are load information collection and dissemination activated? These can be periodic or event-based. A typical period is in the range of 1 second or longer, while typical events are process creation, termination, or migration. The frequency of information dissemination is usually lower than the frequency of information collection, i.e. it is averaged over time in order to prevent instability [Casavant and Kuhl, 1988b]. It also depends on the costs involved with dissemination and the costs of process migration. The lower the costs, the shorter the period can be; the higher the costs, less frequently load information is disseminated.

How much information should be transferred? It can be the entire state, but typically only a subset is transferred in order to minimize the transfer costs and have a scalable solution. In large systems, approximations are applied. For example, only a subset of the information might be transferred, or it might be derived from the subset of all nodes [Barak and Shiloh, 1985; Alon et al., 1987; Han and Finkel, 1988; Chapin and Spafford, 1994].

There are two important observations derived from the research in load information management. The first one is that just a small amount of information can lead to substantial performance improvements. This observation is related to load distribution in general, but it also applies to process migration. Eager et al. were among the first to argue that load sharing using minimal load information can gain dramatic improvements in performance over the non-load-sharing case, and perform nearly as well as more complex policies using more information [Eager et al., 1986b]. The minimal load information they use consists of the process queue length of a small number of successively probed remote nodes. A small amount of state also reduces communication overhead. Kunz comes to the same conclusion using the concept of stochastic learning automata to implement a task scheduler [Kunz, 1991].

The second observation is that the current lifetime of a process can be used for load distribution purposes. The issue is to find how old the process needs to be before it is worth to migrate it. Costs involved with migrating short-lived processes can outweigh the benefits. Leland and Ott were the first to account for the process age in the balancing policy [1986]. Cabrera finds that it is possible to predict a process's expected lifetime from how long it has already lived [Cabrera, 1986]. This justifies migrating processes that manage to live to a certain age. In particular, he finds that over 40% of processes doubled

their age. He also finds that the most UNIX processes are short-lived, more than 78% of the observed processes have a lifetime shorter than 1s and 97% shorter than 4s.

Harchol-Balter and Downey explore the correlation between process lifetime and acceptable migration costs [Harchol-Balter and Downey, 1997]. They derive a more accurate form of the process lifetime distribution that allows them to predict the life-time correlated to the process age and to derive a cost criterion for migration. Svensson filters out short-running processes by relying on statistics [Svensson, 1990], whereas Wang et al. deploy AI theory for the same purpose [Wang et al., 1993].

2.8. Distributed Scheduling

This section addresses distributed scheduling closely related to process migration mechanisms. General surveys are presented elsewhere [Wang and Morris, 1985; Casavant and Kuhl, 1988a; Hac, 1989b; Goscinski, 1991; Chapin, 1996].

Distributed scheduling uses the information provided by the load information management module to make migration decisions, as described in Figure 4. The main goal is to determine *when* to migrate *which* process *where*. The activation policy provides the answer to the question *when* to migrate. Scheduling is activated periodically or it is event-driven. After activation, the load is inspected, and if it is above/below a threshold, actions are undertaken according to the selected strategy. The selection policy answers the question *which* process to migrate. The processes are inspected and some of them are selected for migration according to the specified criteria. *Where* to migrate depends on the location policy algorithm, which chooses a remote node based on the available information.

There are a few well-known classes of distributed scheduling policies:

- **A sender-initiated policy** is activated on the node that is overloaded and that wishes to off-load to other nodes. A sender-initiated policy is preferable for low and medium loaded systems, which have a few overloaded nodes. This strategy is convenient for remote invocation strategies [Eager et al., 1986a; Krueger and Livny, 1987b; Agrawal and Ezzat, 1987].
- **A receiver-initiated policy** is activated on underloaded nodes willing to accept the load from overloaded ones. A receiver-initiated policy is preferable for high load systems, with many overloaded nodes and few underloaded ones. Process migration is particularly well-suited for this strategy, since only with migration can one initiate process transfer at an arbitrary point in time [Bryant and Finkel, 1981; Eager et al., 1986a; Krueger and Livny, 1988].
- **A symmetric policy** is the combination of the previous two policies, in an attempt to take advantage of the good characteristics of both of them. It is suitable for a broader range of conditions than either receiver-initiated or sender-initiated strategies alone [Krueger and Livny, 1987b; Shivaratri et al., 1992].
- **A random policy** chooses the destination node randomly from all nodes in a distributed system. This simple strategy can result in a significant performance improvement [Alon et al., 1987; Eager et al., 1986b; Kunz, 1991].

The following are some of the issues in distributed scheduling related to the process migration mechanism:

- **Adaptability** is concerned with the scheduling impact on system behavior [Stankovic, 1984]. Based on the current host and network load, the relative importance of load parameters may change. The policy should adapt to these changes. Process migration is inherently adaptable because it allows processes to run prior to dispatching them to other nodes, giving them a chance to adapt. Migration can happen at any time (thereby adapting to sudden load changes), whereas initial placement happens only prior to starting a process. Examples of adaptive load distribution include work by Agrawal and Ezzat [1987], Krueger

and Livny [1988], Concepcion and Eleazar [1988], Efe and Groselj [1989], Venkatesh and Dattatreya [1990], Shivaratri and Krueger [1990], and Mehra and Wah [1992].

- **Stability** is defined as the ability to detect when the effects of further actions (e.g. load scheduling or paging) will not improve the system state as defined by a user's objective [Casavant and Kuhl, 1988b]. Due to the distributed state, some instability is inevitable, since it is impossible to transfer state changes across the system instantly. However, high levels of instability should be avoided. In some cases it is advisable not to perform any action, e.g. under extremely high loads it is better to abandon load distribution entirely. Process migration can negatively affect stability if processes are migrated back and forth among the nodes, similar to the thrashing introduced by paging [Denning, 1980]. To prevent such behavior a limit on the number of migrations can be imposed. Bryant and Finkel demonstrate how process migration can improve stability [Bryant and Finkel, 1981].
- **Approximate and heuristic scheduling** is necessary since optimal solutions are hard to achieve. Suboptimal solutions are reached either by approximating the search space with its subset or by using heuristics. Some of the examples of approximate and heuristic scheduling include work by Efe [1982], Leland and Ott [1986], Lo [1988], Casavant and Kuhl [1988a], and Xu and Hwang [1990]. Deploying process migration introduces more determinism and requires fewer heuristics than alternative load distribution mechanisms. Even when incorrect migration decisions are made, they can be alleviated by subsequent migrations, which is not the case with initial process placement where processes have to execute on the same node until the end of its lifetime.
- **Hierarchical scheduling** integrates distributed and centralized scheduling. It supports distributed scheduling within a group of nodes and centralized scheduling among the groups. This area has attracted much research [Bowen et al., 1988; Bonomi and Kumar, 1988; Feitelson and Rudolph, 1990; Gupta and Gopinath, 1990; Gopinath and Gupta, 1991; Chapin, 1995]. A process migration mechanism is a good fit for hierarchical scheduling since processes are typically migrated within a LAN or other smaller domain. Only in the case of large load discrepancies are processes migrated between domains, i.e. between peers at higher levels of the hierarchy.

The most important question that distributed scheduling studies address related to process migration is whether migration pays off. The answer depends heavily on the assumptions made. For example, Eager et al. compare the receiver- and sender-initiated policies [Eager et al., 1986a], and show that the sender-initiated policies outperform the receiver-initiated policies for light and moderate system loads. The receiver-initiated policy is better for higher loads, assuming that transfer costs are same. They argue that the transfer costs for the receiver policy, that requires some kind of migration, are much higher than the costs for mechanisms for the sender-initiated strategies, where initial placement suffices. They finally conclude that under no condition could migration provide significantly better performance than initial placement [Eager et al., 1988].

Krueger and Livny investigate the relationship between load balancing and load sharing [Krueger and Livny, 1988]. They argue that load balancing and load sharing represent various points in a continuum defined by a set of goals and load conditions [Krueger and Livny, 1987]. They claim that the work of Eager et al. [1988] is only valid for a part of the continuum, but it cannot be adopted generally. Based on better job distributions than those used by Eager et al., their simulation results show that migration can improve performance.

Harchol-Balter and Downey present the most recent results on the benefits of using process migration [Harchol-Balter and

Downey, 1997]. They use the measured distribution of process lifetimes for a variety of workloads in an academic environment. The crucial point of their work is understanding the correct lifetime distribution, which they find to be Pareto (heavy-tailed). Based on the trace-driven simulation, they demonstrate a 35-50% improvement in the mean delay when using process migration instead of remote execution (preemptive v. non-preemptive scheduling) even when the costs of migration are high.

Their work differs from Eager et al. [1988] in system model and workload description. Eager et al. model server farms, where the benefits of remote execution are overestimated: there are no associated costs and no affinity toward a particular node. Harchol-Balter and Downey model a network of workstations where remote execution entails costs, and there exists an affinity toward some of the nodes in a distributed system. The workload that Eager et al. use contains few jobs with non-zero life-times, resulting in a system with little imbalance and little need for process migration.

2.9. Alternatives to Process Migration

Given the relative complexity of implementation, and the expense incurred when process migration is invoked, researchers often choose to implement alternative mechanisms [Shivaratri et al., 1992; Kremien and Kramer, 1992].

Remote execution is the most frequently used alternative to process migration. Remote execution can be as simple as the invocation of some code on a remote node, or it can involve transferring the code to the remote node and inheriting some of the process environment, such as variables and opened files. Remote execution is usually faster than migration because it does not incur the cost of transferring a potentially large process state (such as the address space, which is created anew in the case of remote execution). For small address spaces, the costs for remote execution and migration can be similar. Remote execution is used in many systems such as COCANET [Rowe and Birman, 1982], Nest [Agrawal and Ezzat, 1987], Sprite [Ousterhout et al., 1988], Plan 9 [Pike et al., 1990], Amoeba [Mullender et al., 1990], Drums [Bond, 1993], Utopia [Zhou et al., 1994], and Hive [Chapin et al., 1995].

Remote execution has disadvantages as well. It allows creation of the remote instance only at the time of process creation, as opposed to process migration which allows moving the process at an arbitrary time. Allowing a process to run on the source node for some period of time is advantageous in some respects. This way, short-lived processes that are not worth migrating are naturally filtered out. Also, the longer a process runs, the more information about its behavior is available, such as whether and with whom it communicates. Based on this additional information, scheduling policies can make more appropriate decisions.

Cloning processes is useful in cases where the child process inherits state from the parent process. Cloning is typically achieved using a remote *fork* mechanism. A remote fork, followed by the termination of the parent, resembles process migration. The complexity of cloning processes is similar to migration, because the same amount of the process state is inherited (e.g. open files and address space). In the case of migration, the parent is terminated. In the case of cloning, both parent and child may continue to access the same state, introducing distributed shared state, which is typically complex and costly to maintain. Many systems use remote forking [Goldberg and Jefferson, 1987; Smith and Ioannidis, 1989; Zajcew et al., 1993].

Programming language support for mobility enables a wide variety of options, since such systems have almost complete control over the runtime implementation of an application. Such systems can enable self-checkpointing (and hence migratable) applications. They are suitable for entire processes, but also for objects as small as a few bytes, such as in Emerald [Jul et al., 1988; Jul, 1989] or Ellie [Andersen, 1992]. Finer granularity incurs lower transfer costs. The complexity

of maintaining communication channels poses different kinds of problems. In Emerald, for example, the pointers have to be updated to the source object. Programming language support allows a programmer to introduce more information on object behavior, such as hints about communication and concurrency patterns.

Object migration at the middleware level is also possible. Because of the increasing costs of operating system development and the lack of standard solutions for distributed systems and heterogeneity, middleware level solutions have become of more interest [Bernstein, 1996]. Distributed objects are supported in middleware systems such as DCE [Rosenberry et al., 1992] and CORBA [OMG, 1996]. Object migration at the middleware level has not attracted as much research as process migration in operating systems. One of the reasons is that the early heterogeneity of these systems did not adequately support mobility. Nevertheless, a couple of systems do support mobility at the middleware level, such as DC++ [Schill and Mock, 1993] and the OMG MASIF specification for mobile agents [Milojičić et al., 1998b] based on OMG CORBA.

Mobile agents are becoming increasingly popular. The mobility of agents on the Web emphasizes safety and security issues more than complexity, performance, transparency and heterogeneity. Mobile agents are implemented on top of safe languages, such as Java [Gosling et al., 1996], Telescript [White, 1996] and Tcl/Tk [Ousterhout, 1994]. Compared to process migration, mobile agents have reduced implementation complexity because they do not have to support OS semantics. Performance requirements are different due to the wide-area network communication cost, which is the dominant factor. Heterogeneity is abstracted away at the language level. The early results and opportunities for deployment, as well as the wide interest in the area of mobile agents, indicate a promising future for this form of mobility. However, the issues of security, social acceptance, and commercializable applications have been significantly increased and they represent the main focus of research in the mobile agent community. Mobile agents are described in more detail in Section 4.8.

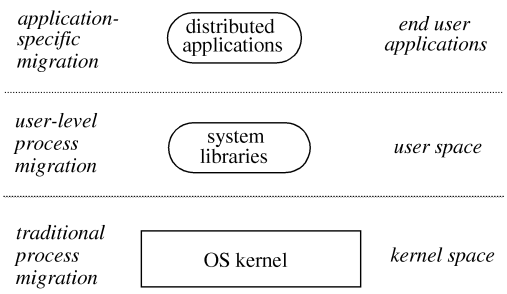

Fig. 5. Migration levels differ in implementation complexity, performance, transparency, and reusability.

3. CHARACTERISTICS

This section addresses issues in process migration, such as complexity, performance, transparency, fault resilience, scalability and heterogeneity. These characteristics have a major impact on the effectiveness and deployment of process migration.

3.1. Complexity and Operating System Support

The complexity of implementation and dependency on an operating system are among the obstacles to the wider use of process migration. This is especially true for fully-transparent migration implementations. Migration can be classified according to the level at which it is applied. It can be applied as part of the operating system kernel, in user space, as part of a system environment, or as a part of the application (see Figure 5). Implementations at different levels result in different performance, complexity, transparency and reusability.

User-level migration typically yields simpler implementations, but suffers too much from reduced performance and transparency to be of general use for load distribution. User-space implementations are usually provided for the support of long-running computations [Litzkow and Solomon, 1992]. Migration implemented as part of an application can have poor

reusability if modifications are required to the application, as was done in the work by Freedman [1991] and Skordos [1995]. This requires familiarity with applications and duplicating some of the mechanisms for each subsequent application, frequently involving effort beyond re-linking the migration part with the application code. It could be somewhat improved if parts of migration support is organized in a reusable run-time library. Lower-level migration is more complex to implement, but has better performance, transparency and reusability.

Despite high migration costs, user-level implementations have some benefits with regard to policy. The layers closer to an application typically have more knowledge about its behavior. This knowledge can be used to derive better policies and hence, better overall performance. Similar motivations led to the development of microkernels, such as Mach [Accetta et al., 1986], Chorus [Rozier, 1992], and Amoeba [Tanenbaum, 1990], which have moved much of their functionality from the kernel into user space. For example, file servers and networking may be implemented in user space, leaving only a minimal subset of functionality provided in the microkernel, such as virtual memory management, scheduling and interprocess communication.

Extensible kernels, such as Spin [Bershad et al., 1995], Exokernel [Engler et al., 1995], and Synthetix [Pu et al., 1995], have taken an alternative approach by allowing user implemented parts to be imported into the kernel. Both microkernels and extensible kernels provide opportunities for extracting a process's state from the operating system.

There have been many implementations of migration for various operating systems and hardware architectures; many of them required a significant implementation effort and modifications to the underlying kernel [Barak and Shiloh, 1985; Theimer et al., 1985; Zayas, 1987a; Douglis and Ousterhout, 1991]. This complexity is due to the underlying operating system architecture and specifically its lack of support for the complex interactions resulting from process migration. In the early days, migration required additional OS support, such as extensions for communications forwarding [Artsy et al., 1987], or for data transfer strategies [Theimer et al., 1985; Zayas, 1987a]. In the case of some subsequent migration implementations, this support already existed in the OS, such as in the case of Mach [Milojičić et al., 1993a].

In UNIX-like operating systems, support for opened files and signals requires significant interaction with various kernel subsystems [Douglis, 1989; Welch, 1990]. Process migration in message-passing kernels requires significant effort to support message handling [Theimer et al., 1985; Artsy et al., 1987; Artsy and Finkel, 1989]. Recent operating systems provide much of this support, such as transparent distributed IPC with message forwarding, and external distributed pagers, which allow easier optimizations and customizing [Black et al., 1992; Rozier, 1992]. Nevertheless, migration still challenges these mechanisms and frequently breaks them [Douglis and Ousterhout, 1991; Milojičić, 1993c].

3.2. Performance

Performance is the second important factor that affects the deployment of process migration. Migration performance depends on initial and run-time costs introduced by the act of migration. The initial costs stem from state transfer. Instead of at migration time, some of the state may be transferred *lazily* (on-demand), thereby incurring run-time costs. Both types of cost may be significant, depending on the application characteristics, as well as on the ratio of state transferred eagerly/lazily.

If only part of the task state is transferred to another node, the task can start executing sooner, and the initial migration costs are lower. This principle is called *lazy evaluation*: actions are not taken before they are really needed with the hope that they will never be needed. However, when this is not true, penalties are paid for postponed access. For example, it is

convenient to migrate a huge address space on demand instead of eagerly. In the lazy case, part of the space may never be trans-ferred if it is not accessed. However, the source node needs to retain lazily evaluated state throughout the life-time of the migrated process.

A process's address space usually constitutes by far the largest unit of process state; not surprisingly, the performance of process migration largely depends on the performance of the address space transfer. Various data transfer strategies have been invented in order to avoid the high cost of address space transfer.

- The **eager (all)** strategy copies all of the address space at the migration time. Initial costs may be in the range of minutes. Checkpoint/restart implementations typically use this strategy, such as Condor [Litzkow and Solomon, 1992] or LSF [Zhou et al., 1994].
- The **eager (dirty)** strategy can be deployed if there is remote paging support. This is a variant of the eager (all) strategy that transfers only modified (dirty) pages. Unmodified pages are paged in on request from a backing store. Eager (dirty) significantly reduces the initial transfer costs when a process has a large address space. Systems supporting eager (dirty) strategy include MOSIX [Barak and Litman, 1985] and Locus [Popek and Walker, 1985].
- The **Copy-On-Reference (COR)** strategy is a network version of demand paging: pages are transferred only upon reference. While dirty pages are brought from the source node, clean pages can be brought either from the source node or from the backing store. The COR strategy has the lowest initial costs, ranging from a few tens to a few hundred microseconds. However, it increases the run-time costs, and it also requires substantial changes to the underlying operating system and to the paging support [Zayas, 1987a].
- The **flushing** strategy consists of flushing dirty pages to disk and then accessing them on demand from disk instead of from memory on the source node as in copy-on-reference [Douglis and Ousterhout, 1991]. The flushing strategy is like the eager (dirty) transfer strategy from the perspective of the source, and like copy-on-reference from the target's viewpoint. It leaves dependencies on the server, but not on the source node.
- The **precopy** strategy reduces the "freeze" time of the process, the time that process is neither executed on the source nor on the destination node. While the process is executed on the source node, the address space is being transferred to the remote node until the number of dirty pages is smaller than a fixed limit. Pages dirtied during precopy have to be copied a second time. The precopy strategy cuts down the freeze time below the costs of the COR technique [Theimer et al., 1985].

There are also variations of the above strategies. The most notable example is migration in the **Choices** operating system [Roush and Campbell, 1996]. It uses a variation of the eager (dirty) strategy which transfers minimal state to the remote node at the time of migration. The remote instance is started while the remainder of the state is transferred in parallel. The initial migration time is reduced to 13.9ms running on a SparcStation II connected by a 10Mb Ethernet, which is an order of magnitude better than all other reported results, even if results are normalized (see work by Rousch [1995] for more details on normalized performance results).

Leaving some part of the process state on the source or intermediate nodes of the migrated instance results in a *residual dependency.* Residual dependencies typically occur as a consequence of two implementation techniques: either using lazy evaluation (see definition below), or as a means for achieving transparency in communication, by forwarding subsequent messages to a migrated process.

A particular case of residual dependency is the *home dependency,* which is a dependency on the (home) node where

a process was created [Douglis and Ousterhout, 1991]. An example of a home dependency is redirecting systems calls to the home node: for example, local host-dependent calls, calls related to the file system (in the absence of a distributed file system), or operations on local devices. A home dependency can simplify migration, because it is easier to redirect requests to the home node than to support services on all nodes. However, it also adversely affects reliability, because a migrated foreign process will always depend on its home node. The notion of the home dependency is further elaborated upon below in Section 5.1 (MOSIX) and Section 5.2 (Sprite).

Redirecting communication through the previously established links represents another kind of residual dependency. In general, dependencies left at multiple nodes should be avoided, since they require complex support, and degrade performance and fault resilience. Therefore, some form of periodic or lazy removal of residual dependencies is desirable. For example, the system could flush remaining pages to the backing store, or update residual information on migrated communication channels.

3.3. Transparency

Transparency requires that neither the migrated task nor other tasks in the system can notice migration, with the possible exception of performance effects. Communication with a migrated process could be delayed during migration, but no message can be lost. After migration, the process should continue to communicate through previously opened I/O channels, for example printing to the same console or reading from the same files.

Transparency is supported in a variety of ways, depending on the underlying operating system. Sprite and NOW MOSIX maintain a notion of a home machine that executes all host-specific code [Douglis and Ousterhout, 1991; Barak et al., 1995]. Charlotte supports IPC through links, which provide for remapping after migration [Finkel et al., 1989].

Transparency also assumes that the migrated instance can execute all system calls as if it were not migrated. Some user-space migrations do not allow system calls that generate internode signals or file access [Mandelberg and Sunderam, 1988; Freedman, 1991].

Single System Image (SSI) represents a complete form of transparency. It provides a unique view of a system composed of a number of nodes as if there were just one node. A process can be started and communicated with without knowing where it is physically executing. Resources can be transparently accessed from any node in the system as if they were attached to the local node. The underlying system typically decides where to instantiate new processes or where to allocate and access resources.

SSI can be applied at different levels of the system. At the user-level, SSI consists of providing transparent access to objects and resources that comprise a particular programming environment. Examples include Amber [Chase et al., 1989] and Emerald [Jul, 1989]. At the traditional operating system level, SSI typically consists of a distributed file system and distributed process management, such as in MOSIX [Barak and Litman, 1985], Sprite [Ousterhout et al., 1988] and OSF/1 AD TNC [Zajcew et al., 1993]. At the microkernel level, SSI is comprised of mechanisms, such as distributed IPC, distributed memory management, and remote tasking. A near-SSI is implemented for Mach [Black et al., 1992] based on these transparent mechanisms, but the policies are supported at the OSF/1 AD server running on top of it. At the microkernel level the programmer needs to specify where to create remote tasks.

SSI supports transparent access to a process, as well as to its resources, which simplifies migration. On the other hand, the migration mechanism exercises functionality provided at the SSI level, posing a more stressful workload than normally experienced in systems without migration [Milojičić et al., 1993a]. Therefore, although a migration implementation on top of SSI may seem less complex, this

complexity is pushed down into the SSI implementation.

Some location dependencies on another host may be inevitable, such as accessing local devices or accessing kernel-dependent state that is managed by the other host. It is not possible transparently to support such dependencies on the newly visited nodes, other than by forwarding the calls back to the *home* node, as was done in Sprite [Douglis and Ousterhout, 1991].

3.4. Fault Resilience

Fault resilience is frequently mentioned as a benefit of process migration. However, this claim has never been substantiated with a practical implementation, although some projects have specifically addressed fault resilience [Chou and Abraham, 1983; Lu et al., 1987]. So far the major contribution of process migration for fault resilience is through combination with checkpointing, such as in Condor [Litzkow and Solomon, 1992], LSF Zhou et al., 1994] and in work by Skordos [1995]. Migration was also suggested as a means of fault containment [Chapin et al., 1995].

Failures play an important role in the implementation of process migration. They can happen on a source or target machine or on the communication medium. Various migration schemes are more or less sensitive to each type of failure. Residual dependencies have a particularly negative impact on fault resilience. Using them is a trade-off between efficiency and reliability.

Fault resilience can be improved in several ways. The impact of failures during migration can be reduced by maintaining process state on both the source and destination sites until the destination site instance is successfully promoted to a regular process and the source node is informed about this. A source node failure can be overcome by completely detaching the instance from the source node once it is migrated, though this prevents lazy evaluation techniques from being employed. One way to remove communication residual dependencies is to deploy locating techniques, such as multicasting (as used in V kernel [Theimer et al., 1985), reliance on the home node (as used in Sprite [Douglis and Ousterhout, 1991], and MOSIX [Barak and Litman, 1985]), or on a forwarding name server (as used in most distributed name services, such as DCE, as well as in mobile agents, such as MOA [Milojičić et al., 1999]). This way dependencies are singled out on dedicated nodes, as opposed to being scattered throughout all the nodes visited, as is the case for Charlotte [Artsy et al., 1987]. Shapiro, et al. [1992] propose so-called SSP Chains for periodically collapsing forwarding pointers (and thereby reducing residual dependencies) in the case of garbage collection.

3.5. Scalability

The scalability of a process migration mechanism is related to the scalability of its underlying environment. It can be measured with respect to the number of nodes in the system, to the number of migrations a process can perform during its lifetime, and to the type and complexity of the processes, such as the number of open channels or files, and memory size or fragmentation.

The number of nodes in the system affects the organization and management of structures that maintain residual process state and the naming of migrated processes. If these structures are not part of the existing operating system, then they need to be added.

Depending on the migration algorithm and the techniques employed, some systems are not scalable in the number of migrations a process may perform. As we shall see in the case study on Mach (see Section 5.3), sometimes process state can grow with the number of migrations. This is acceptable for a small number of migrations, but in other cases the additional state can dominate migration costs and render the migration mechanism useless.

Migration algorithms should avoid linear dependencies on the amount of state to be transferred. For example, the eager data transfer strategy has costs proportional to the address space size, incurring

significant costs for large address spaces. The costs for a lazily copied process are independent of the address space size, but they can depend on the granularity and type of the address space. For example, the transfer of a large sparse address space can have costs proportional to the number of contiguous address space regions, because each such region has metadata associated with it that must be transferred at migration time. This overhead can be exacerbated if the meta-data for each region is transferred as a separate operation, as was done in the initial implementation of Mach task migration [Milojičić et al., 1993b].

Communication channels can also affect scalability. Forwarding communication to a migrated process is acceptable after a small number of sequential migrations, but after a large number of migrations the forwarding costs can be significant. In that case, some other technique, such as updating communication links, must be employed.

3.6. Heterogeneity

Heterogeneity has not been addressed in most early migration implementations. Instead, homogeneity is considered as a requirement; migration is allowed only among the nodes with a compatible architecture and processor instruction set. This was not a significant limitation at the time since most of the work was conducted on clusters of homogeneous machines.

Some earlier work indicated the need as well as possible solutions for solving the heterogeneity problem, but no mature implementations resulted [Maguire and Smith, 1988; Dubach, 1989; Shub, 1990; Theimer and Hayes, 1991].

The deployment of world-wide computing has increased the interest in heterogeneous migration. In order to achieve heterogeneity, process state needs to be saved in a machine-independent representation. This permits the process to resume on nodes with different architectures. An application is usually compiled in advance on each architecture, instrumenting the code to know what procedures and variables exist at any time, and identifying points at which the application can be safely preempted and checkpointed. The checkpointing program sets a breakpoint at each preemption point and examines the state of the process when a breakpoint is encountered. Smith and Hutchinson note that not all programs can be safely checkpointed in this fashion, largely depending on what features of the language are used [Smith and Hutchinson, 1998]. Emerald [Steensgaard and Jul, 1995] is another example of a heterogeneous system.

In the most recent systems, heterogeneity is provided at the language level, as by using intermediate byte code representation in Java [Gosling et al., 1996], or by relying on scripting languages such as Telescript [White, 1996] or Tcl/Tk [Ousterhout, 1994].

3.7. Summary

This subsection evaluates the trade-offs between various characteristics of process migration, and who should be concerned with it.

Complexity is much more of a concern to the implementors of a process migration facility than to its users. Complexity depends on the level where migration is implemented. Kernel-level implementations require significantly more complexity than user-level implementations. Users of process migration are impacted only in the case of user-level implementations where certain modifications of the application code are required or where migration is not fully transparent.

Long-running applications are not concerned with performance as are those applications whose lifetimes are comparable to their migration time. Short-running applications are generally not good candidates for migration. Migration-time performance can be traded off against execution-time (by leaving residual dependencies, or by lazily resolving communication channels). Residual dependencies are of concern for long-running applications and for network applications. Applications with real-time requirements generally are not suitable candidates for residual dependency because of the unpredictable costs

of bringing in additional state. On the other hand, real-time requirements can be more easily fulfilled with strategies, such as precopy.

Legacy applications are concerned with transparency in order to avoid any changes to existing code. Scientific applications typically do not have transparency requirements. Frequently, one is allowed to make modifications to the code of these applications, and even support migration at the application level (e.g. by checkpointing state at the application level). Transparency typically incurs complexity. However, transparency is not related to migration exclusively, rather it is inherent to remote access. Transparent remote execution can require support that is as complex as transparent process migration [Douglis and Ousterhout, 1991].

Scientific applications (typically long-running), as well as network applications are concerned with failure tolerance. In most cases periodic checkpointing of the state suffices.

Scalability requires additional complexity for efficient support. It is of concern for scientific applications because they may require a large number of processes, large address spaces, and a large number of communication channels. It is also important for network applications, especially those at the Internet scale.

Heterogeneity introduces performance penalties and additional complexity. It is of most concern to network applications which typically run on inhomogeneous systems.

4. EXAMPLES

This section classifies process migration implementations in the following categories: early work; UNIX-like systems supporting transparent migration; systems with message-passing interfaces; microkernels; user-space migration; and application-specific migration. In addition, we also give an overview of mobile objects and mobile agents. These last two classes do not represent process migration in the classic sense, but they are similar in sufficiently many ways to warrant their inclusion [Milojičić et al., 1998a]. For each class, an overview and some examples are presented. Finally, in the last subsection, we draw some conclusions. The next section expands upon four of these systems in substantial detail.

There are also other examples of process migration that can fit into one or more classes presented in this section. Examples include object migration in Eden [Lazowska, et al., 1981]; MINIX [Louboutin, 1991]; Galaxy [Sinha et al., 1991]; work by Dediu [1992]; EMPS [van Dijk and van Gils, 1992]; object migration for OSF DCE, DC++ [Schill and Mock, 1993]; work by Petri and Langendorfer [1995]; MDX [Schrimpf, 1995]; and many more. A description of these systems is beyond the scope of this paper. In addition to other surveys of process migration already mentioned in the introduction [Smith, 1988; Eskicioglu, 1990; Nuttal, 1994], Borghoff provides a catalogue of distributed operating systems with many examples of migration mechanisms [Borghoff, 1991].

4.1. Early Work

Early work is characterized by specialized, *ad hoc* solutions, often optimized for the underlying hardware architecture. In this subsection we briefly mention XOS, Worm, DEMOS/MP and Butler.

Migration in **XOS** is intended as a tool for minimizing the communication between the nodes in an experimental multiprocessor system, organized in a tree fashion [Miller and Presotto, 1981]. The representation of the process and its state are designed in a such a way as to facilitate migration. The Process Work Object (PWO) encapsulates process related state including stack pointers and registers. Migration is achieved by moving PWO objects between the XOS nodes. The process location is treated as a hint, and the current location is found by following hints.

The **Worm** idea has its background in the nature of real worms [Shoch and Hupp, 1982]. A worm is a computation that can live on one or more machines. Parts of the worm residing on a single

machine are called segments. If a segment fails, other segments cooperatively reinstantiate it by locating a free machine, rebooting it from the network, and migrating the failed worm segment to it. A worm can move from one machine to another, occupying needed resources, and replicating itself. As opposed to other migration systems, a worm is aware of the underlying network topology. Communication among worm segments is maintained through multicasting.

The original **Butler** system supports remote execution and process migration [Dannenberg, 1982]. Migration occurs when the guest process needs to be "deported" from the remote node, e.g. in case when it exceeds resources it negotiated before arrival. In such a case, the complete state of the guest process is packaged and transferred to a new node. The state consists of the address space, registers, as well as the state contained in the servers collocated at the same node. Migration does not break the communication paths because the underlying operating system (Accent [Rashid and Robertson, 1981]) allows for port migration. The Butler design also deals with the issues of protection, security, and autonomy [Dannenberg and Hibbard, 1985]. In particular, the system protects the client program, the Butler daemons on the source and destination nodes, the visiting process, and the remote node. In its later incarnation, Butler supports only remote invocation [Nichols, 1987].

DEMOS/MP [Miller et al., 1987] is a successor of the earlier version of the DEMOS operating system [Baskett et al., 1977]. Process migration is fully transparent: a process can be migrated during execution without limitations on resource access. The implementation of migration has been simplified and its impact to other services limited by the message-passing, location-independent communication, and by the fact that the kernel can participate in the communication in the same manner as any process [Powell and Miller, 1983]. Most of the support for process migration already existed in the DEMOS kernel. Extending it with migration required mechanisms for forwarding messages and updating links. The transferred state includes program code and data (most of the state), swappable and nonswappable state, and messages in the incoming queue of the process.

4.2. Transparent Migration in UNIX-like Systems

UNIX-like systems have proven to be relatively hard to extend for transparent migration and have required significant modifications and extensions to the underlying kernel (see Subsections 4.3 and 4.4 for comparisons with other types of OSes). There are two approaches to addressing distribution and migration for these systems. One is to provide for distribution at the lower levels of a system, as in MOSIX or Sprite, and the other is by providing distribution at a higher-level, as in Locus and its derivatives. In this section, we shall describe process migration for Locus, MOSIX and Sprite. All of these systems also happened to be RPC-based, as opposed to the message-passing systems described in Section 4.3.

Locus is a UNIX-compatible operating system that provides transparent access to remote resources, and enhanced reliability and availability [Popek et al., 1981; Popek and Walker, 1985]. It supports process migration [Walker et al., 1983] and initial placement [Butterfield and Popek, 1984]. Locus is one of the rare systems that achieved product stage. It has been ported to the AIX operating system on the IBM 370 and PS/2 computers under the name of the Transparent Computing Facility (TCF) [Walker and Mathews, 1989]. Locus migration has a high level of functionality and transparency. However, this required significant kernel modifications.

Locus has subsequently been ported to the **OSF/1 AD** operating system, under the name of **TNC** [Zajcew et al., 1993]. OSF/1 AD is a distributed operating system running on top of the Mach microkernel on Intel x86 and Paragon architectures (see Section 5.3). TNC is only partially concerned with task migration issues of the underlying Mach microkernel, because in

the OSF/1 AD environment the Mach interface is not exposed to the user, and therefore the atomicity of process migration is not affected. Locus was also used as a testbed for a distributed shared memory implementation, Mirage [Fleisch and Popek, 1989]. Distributed shared memory was not combined with process migration as was done in the case of Mach (see Section 5.3).

The **MOSIX** distributed operating system is an ongoing project that began in 1981. It supports process migration on top of a single system image base [Barak and Litman, 1985] and in a Network of Workstations environment [Barak et al., 1995]. The process migration mechanism is used to support dynamic load balancing. MOSIX employs a probabilistic algorithm in its load information management that allows it to transmit partial load information between pairs of nodes [Barak and Shiloh, 1985; Barak and Wheeler, 1989]. A case study of the MOSIX system is presented in Section 5.1.

The **Sprite** network operating system [Ousterhout et al., 1988] was developed from 1984–1994. Its process migration facility [Douglis and Ousterhout, 1991] was transparent both to users and to applications, by making processes appear to execute on one host throughout their execution. Processes could access remote resources, including files, devices, and network connections, from different locations over time. When a user returned to a workstation onto which processes had been offloaded, the processes were immediately migrated back to their home machines and could execute there, migrate else-where, or suspend execution. A case study of the Sprite system is presented in Section 5.2.

4.3. OS with Message-Passing Interface

Process migration for message-passing operating systems seems easier to design and implement. Message passing is convenient for interposing, forwarding and encapsulating state. For example, a new receiver may be interposed between the existing receiver and the sender, without the knowledge of the latter, and messages sent to a migrated process can be forwarded after its migration to a new destination. However, much of the simplicity that seems to be inherent for message-passing systems is hidden inside the complex message-passing mechanisms [Douglis and Ousterhout, 1991].

In this section we describe Charlotte, Accent and the V kernel. The V kernel can be classified both as a microker-nel and as a message passing kernel; we chose to present it in the message-passing section.

Charlotte is a message-passing operating system designed for the Crystal multicomputer composed of 20 VAX-11/750 computers [Artsy and Finkel, 1989]. The Charlotte migration mechanism extensively relies on the underlying operating system and its communication mechanisms which were modified in order to support transparent network communication [Artsy et al., 1987]. Its process migration is well insulated from other system modules. Migration is designed to be fault resilient: processes leave no residual dependency on the source machine. The act of migration is committed in the final phase of the state transfer; it is possible to undo the migration before committing it.

Accent is a distributed operating system developed at CMU [Rashid and Robertson, 1981; Rashid, 1986]. Its process migration scheme was the first one to use the *"Copy-On-Reference" (COR)* technique to lazily copy pages [Zayas, 1987a]. Instead of eagerly copying pages, virtual segments are created on the destination node. When a page fault occurs, the virtual segment provides a link to the page on the source node. The duration of the initial address space transfer is independent of the address space size, but rather depends on the number of contiguous memory regions. The subsequent costs for lazily copied pages are proportional to the number of pages referenced. The basic assumption is that the program would not access all of its address space, thereby saving the cost of a useless transfer. Besides failure vulnerability, the drawback of lazy evaluation is the increased complexity of in-kernel memory management [Zayas, 1987b].

The **V Kernel** is a microkernel developed at Stanford University [Cheriton, 1988]. It introduces a "*precopying*" technique for the process address space transfer [Theimer et al., 1985]. The address space of the process to be migrated is copied to the remote node prior to its migration, while the process is still executing on the source node. Dirty pages referenced during the precopying phase are copied again. It has been shown that only two or three iterations generally suffice to reach an acceptably small number of dirty pages. At that point of time the process is frozen and migrated. This technique shortens the process freeze time, but otherwise negatively influences the execution time, since overhead is incurred in iterative copying. Migration benefits from a communications protocol that dynamically rebinds to alternate destination hosts as part of its implementation of reliable message delivery. Instead of maintaining process communication endpoints after migration, V relies on multicast to find the new process location.

4.4. Microkernels

The microkernel approach separates the classical notion of a monolithic kernel into a microkernel and an operating system personality running on top of it in a separate module. A microkernel supports tasks, threads, IPC and VM management, while other functionality, such as networking, file system and process management, is imple-mented in the OS personality. Various OS personalities have been implemented, such as BSD UNIX [Golub et al., 1990], AT&T UNIX System V [Rozier, 1992; Cheriton, 1990], MS DOS [Malan et al., 1991], VMS [Wiecek, 1992], OS/2 [Phelan and Arendt, 1993] and Linux [Barbou des Places et al., 1996].

In the late eighties and early nineties, there was a flurry of research into microkernels, including systems, such as Mach [Accetta et al., 1986], Chorus [Rozier, 1992], Amoeba [Mullender et al., 1990], QNX [Hildebrand, 1992], Spring [Hamilton and Kougiouris, 1993] and L3 [Liedtke, 1993], which eventually reached commercial implementations, and many more research microkernels, such as Arcade [Cohn et al., 1989], Birlix [Haertig et al., 1993], KeyKOS [Bomberger et al., 1992] and RHODOS [Gerrity et al., 1991].

The microkernel approach, combined with message passing, allows for transparent, straightforward extensions to distributed systems. Not surprisingly, microkernels are a suitable environment for various migration experiments. The task migration mechanism can be reused by different OS personalities, as a common denominator for different OS-specific process migration mechanisms. In this subsection we describe process migrations for RHODOS, Arcade, Chorus, Amoeba, Birlix and Mach.

RHODOS consists of a nucleus that supports trap and interrupt handling, context switching, and local message passing. The kernel runs on top of the nucleus and supports IPC, memory, process, and migration managers [Gerrity et al., 1991]. The migration mechanism is similar to that in Sprite, with some modifications specific to the RHODOS kernel [Zhu, 1992].

Arcade considers groups of tasks for migration [Cohn et al., 1989]. It is used as a framework for investigating sharing policies related to task grouping [Tracey, 1991]. The group management software ensures that members of the group execute on different machines, thereby exploiting parallelism.

The **Chorus** microkernel was extended to support process migration [Philippe, 1993]. The migration mechanism is similar to task migration on top of Mach (cf. Section 5.3), however it is applied at the process level, instead of the Actor level. Actors in Chorus correspond to Mach tasks. Chorus migration is biased toward the hypercube implementation (fast and reliable links). Some limitations were introduced because Chorus did not support port migration.

Steketee et al. implemented process migration for the **Amoeba** operating system [Steketee et al., 1994]. Communication transparency relies on the location independence of the FLIP protocol [Kaashoek

et al., 1993]. Since Amoeba does not support virtual memory, the memory transfer for process migration is achieved by physical copying [Zhu et al., 1995].

Birlix supports adaptable object migration [Lux, 1995]. It is possible to specify a migration policy on a per-object basis. A meta-object encapsulates data for the migration mechanism and information collection. An example of the use of an adaptable migration mechanism is to extend migration for improved reliability or performance [Lux et al., 1993].

Mach [Accetta et al., 1986] was used as a base for supporting task migration [Milojičić et al., 1993b], developed at the University of Kaiserslautern. The goals were to demonstrate that microkernels are a suitable substrate for migration mechanisms and for load distribution in general. The task migration implementation significantly benefited from the near SSI provided by Mach, in particular from distributed IPC and distributed memory management. Process migration was built for the OSF/1 AD 1 server using Mach task migration [Paindaveine and Milojičić, 1996]. Task and process migration on top of Mach are discussed in more detail in Section 5.3.

4.5. User-space Migrations

While it is relatively straightforward to provide process migration for distributed operating systems, such as the V kernel, Accent, or Sprite, it is much harder to support transparent process migration on industry standard operating systems, which are typically non-distributed. Most workstations in the 1980s and 1990s run proprietary versions of UNIX, which makes them a more challenging base for process migration than distributed operating systems. Source code is not widely available for a proprietary OS; therefore, the only way to achieve a viable and widespread migration is to implement it in user space.

User-space migration is targeted to long-running processes that do not pose significant OS requirements, do not need transparency, and use only a limited set of system calls. The migration time is typically a function of the address space size, since the eager (all) data transfer scheme is deployed. This subsection presents a few such implementations: Condor, the work by Alonso and Kyrimis, the work by Mandelberg and Sunderam, the work by Petri and Langendoerfer, MPVM, and LSF.

Condor is a software package that supports user-space checkpointing and process migration in locally distributed systems [Litzkow, 1987; Litzkow et al., 1988; Litzkow and Solomon, 1992]. Its checkpointing support is particularly useful for long-running computations, but is too expensive for short processes. Migration involves generating a *core* file for a process, combining this file with the executable and then sending this on to the target machine. System calls are redirected to a "shadow" process on the source machine. This requires a special version of the C library to be linked with the migrated programs.

Condor does not support processes that use signals, memory mapped files, timers, shared libraries, or IPC. The scheduler activation period is 10 minutes, which demonstrates the "heaviness" of migration. Nevertheless, Condor is often used for long-running computations. It has been ported to a variety of operating systems. Condor was a starting point for a few industry products, such as LSF from Platform Computing [Zhou et al., 1994] and Loadleveler from IBM.

Alonso and Kyrimis perform minor modifications to the UNIX kernel in order to support process migration in user space [Alonso and Kyrimis, 1988]. A new signal for dumping process state and a new system call for restarting a process are introduced. This implementation is limited to processes that do not communicate and are not location- or process-dependent. The work by Alonso and Kyrimis was done in parallel with the early Condor system.

Mandelberg and Sunderam present a process migration scheme for UNIX that does not support tasks that perform I/O on non-NFS files, spawn subprocesses, or utilize pipes and sockets [Mandelberg and Sunderam, 1988]. A new terminal

interface supports detaching a process from its terminal and monitors requests for I/O on the process migration port.

Migratory Parallel Virtual Machine (MPVM) extends the PVM system [Beguelin et al., 1993] to support process migration among homogeneous machines [Casas et al., 1995]. Its primary goals are transparency, compatibility with PVM, and portability. It is implemented entirely as a user-level mechanism. It supports communication among migrating processes by limiting TCP communication to other MPVM processes.

Load Sharing Facility (LSF) supports migration indirectly through process checkpointing and restart [Platform Computing, 1996]. LSF can work with checkpointing at three possible levels: kernel, user, and application. The technique used for user-level check-pointing is based on the Condor approach [Litzkow and Solomon, 1992], but no *core* file is required, thereby improving performance, and signals can be used across checkpoints, thereby improving transparency. LSF is described in more detail in Section 5.4.

4.6. Application-specific Migration

Migration can also be implemented as a part of an application. Such an approach deliberately sacrifices transparency and reusability. A migrating process is typically limited in functionality and migration has to be adjusted for each new application. Nevertheless, the implementation can be significantly simplified and optimized for one particular application. In this subsection we describe work by Freedman, Skordos, and Bharat and Cardelli.

Freedman reports a process migration scheme involving cooperation between the migrated process and the migration module [Freedman, 1991]. The author observes that long-running computations typically use operating system services in the beginning and ending phases of execution, while most of their time is spent in number-crunching. Therefore, little attention is paid to supporting files, sockets, and devices, since it is not expected that they will be used in the predominant phase of execution. This ad hoc process migration considers only memory contents.

Skordos integrates migration with parallel simulation of subsonic fluid dynamics on a cluster of workstations [Skordos, 1995]. Skordos statically allocates problem sizes and uses migration when a workstation becomes overloaded. Upon migration, the process is restarted after synchronization with processes participating in the application on other nodes. At the same time, it is possible to conduct multiple migrations. On a cluster of 20 HP-Apollo workstations connected by 10 Mbps Ethernet, Skordos notices approximately one migration every 45 minutes. Each migration lasts 30 seconds on average. Despite the high costs, its relative impact is very low. Migrations happen infrequently, and do not last long relative to the overall execution time.

Bharat and Cardelli describe Migratory Applications, an environment for migrating applications along with the user interface and the application context, thereby retaining the same "look and feel" across different platforms [Bharat and Cardelli, 1995]. This type of migration is particularly suitable for mobile applications, where a user may be travelling from one environment to another. Migratory applications are closely related to the underlying programming language Oblique [Cardelli, 1995].

4.7. Mobile Objects

In this paper we are primarily concerned with process and task migration. Object migration and mobile agents are two other forms of migration that we mention briefly in this and the following subsection. Although used in different settings, these forms of migration serve a similar purpose and solve some of the same problems as process migration does. In this subsection, we give an overview of object migration for Emerald, SOS and COOL.

Emerald is a programming language and environment for the support of distributed systems [Black et al., 1987]. It supports mobile objects as small as a

couple of bytes, or as large as a UNIX process [Jul, 1988; Jul et al., 1988]. Objects have a global, single name space. In addition to traditional process migration benefits, Emerald improves data movement, object invocation and garbage collection.

In Emerald, communication links are pointers to other objects. Upon each object migration, all object pointers need to be updated. The Emerald compiler generates the templates that are associated with the object data areas describing its layout. The templates contain information about which objects are associated with the given object, including the pointers to other objects. These pointers are changed if the referenced object moves. Pointers are optimized for local invocation because mobility is a relatively infrequent case compared to local invocation. Objects that become unreachable are garbage collected. Moving a small passive object on a cluster of 4 MicroVax II workstations connected by a 10 megabit/second Ethernet takes about 12 ms while moving a small process takes about 40 ms. Some modest experiments demonstrated the benefits of Emerald for load distribution [Jul, 1989].

Shapiro investigates object migration and persistence in **SOS** [Shapiro et al., 1989]. The objects under consideration are small to medium size (a few hundred bytes). Of particular concern are intra-object references and how they are preserved across object migrations. References are expressed through a new type, called a *permanent pointer*. After migration, permanent pointers are lazily evaluated, based on the proxy principle [Shapiro, 1986]. A proxy is a new object that represents the original object, maintains a reference to it at the new location, and provides a way to access it. Proxies, and the term proxy principle describing its use, are extensively used in distributed systems with or without migration (e.g. for distributed IPC [Barrera, 1991], distributed memory management [Black et al. 1998], and proxy servers on the Web [Brooks, et al. 1995]). Functionality can be arbitrarily distributed between a proxy and its principal object.

COOL provides an object-oriented layer on top of Chorus [Amaral et al., 1992]. It supports DSM-based object sharing, persistent store, and object clustering. Transparent remote invocation is achieved with a simple communication model using the COOL base primitives. When re-mapped onto a new node, all internal references are updated depending on the new location by pointer swizzling [Lea et al., 1993], which is a technique for converting the persistent pointers or object identifiers into the main memory pointers (addresses). Conversion can be activated upon an access to the object (swizzling on discovery) or eagerly (all objects at once upon the discovery of the first persistent pointer). Pointer swizzling can also be used for supporting large and persistent address spaces [Dearle, et al., 1994] and in very large data bases [Kemper and Kossmann, 1995].

4.8. Mobile Agents

In the recent past, mobile agents have received significant attention. A number of products have appeared and many successful research systems have been developed (see description of these systems below). A patent has been approved for one of the first mobile agent systems, Telescript [White, et al. 1997] and a standard was adopted by OMG [Milojičić et al., 1998b].

Mobile agents derive from two fields: agents, as defined in the artificial intelligence community [Shoham, 1997], and distributed systems, including mobile objects and process migration [Milojičić et al., 1999]. However, their popularity started with the appearance of the Web and Java. The former opened vast opportunities for applications suited for mobile agents and the latter became a driving programming language for mobile agents.

In a Web environment, programming languages focus on platform independence and safety. Innovations in OS services take place at the middleware level rather than in kernels [Bernstein, 1996]. Research in distributed systems has largely refocused from local to wide-area networks. Security is a dominant

requirement for applications and systems connected to the Web. In this environment, mobile agents are a very promising mechanism. Typical uses include electronic commerce and support for mobile, sporadically-connected computing for which agents overcome limitations posed by short on-line time, reduced bandwidth, and limited storage.

Java has proven to be a suitable programming language for mobile agents because it supports mobile code and mobile objects, remote object model and language and run-time safety, and it is operating system independent.

While a large amount of the OS-level support for migration concentrated on transparency issues, the agent approach has demonstrated less concern for transparency.

We provide an overview a few commercial mobile agent systems, such as Telescript, IBM Aglets, and Concordia, and a few academic systems, such as Agent Tcl, TACOMA and Mole.

Telescript first introduced the mobile agent concepts [White, 1996]. It is targeted for the MagicCap, a small handheld device. Telescript first introduced mobile agent concepts *place* and *permit* and mechanisms *meet* and *go*. **IBM Aglets** is one of the first commercially available mobile agent systems based on Java [Lange and Oshima, 1998]. It is developed by IBM Tokyo Research Lab IBM. Aglets has a large community of users and applications, even a few commercial ones. **Concordia** is a mobile agent system developed at the Mitsubishi Electric ITA Laboratory [Wong, et al., 1997]. It is a Java-based system that addresses security (by extending the Java security manager) and reliability (using message queuing based on two-phase-commit protocol). Concordia is used for many in-house applications.

Agent Tcl started as a Tcl/Tk-based transportable agent, but it has been extended to support Java, Scheme and C/C++ [Kotz, et al., 1997]. It is used for the development of the DAIS system for information retrieval and dissemination in military intelligence [Hoffman, et al., 1998]. Agent Tcl is optimized for mobile computers, e.g. by minimizing connection time and communication. The **TACOMA** project is a joint effort by Tromso and Cornell Universities [Johansen et al., 1995]. Compared to other mobile agent research, which addresses programming languages aspects, TACOMA addresses operating system aspects. The main research topics include security and reliability. **Mole** is one of the first academic agent systems written in Java [Baumann, et al., 1998]. It has been used by industry (Siemens, Tandem, and Daimler Benz), and academia (University of Geneva). Mole addresses groups of agents, agent termination, and security for protecting agents against malicious hosts.

There are also many other mobile agent systems, such as Ara [Peine and Stolpmann, 1997], Messenger [Tschudin, 1997], MOA [Milojičić et al., 1998a], and Sumatra [Ranganathan, et al., 1997]. A lot of effort has been invested in security of mobile agents, such as in the work by Farmer, et al. [1996], Hohl [1998], Tardo and Valente [1996], Vigna [1998], and Vitek, et al. [1997]. A paper by Chess et al. [1995] is a good introduction to mobile agents.

5. CASE STUDIES

This section presents four case studies of process migration: MOSIX, Sprite, Mach, and LSF. At least one of the authors of this survey directly participated in the design and implementation of each of these systems. Because it is difficult to choose a representative set of case studies, the selection of systems was guided by the authors' personal experience with the chosen systems.

5.1. MOSIX

MOSIX is a distributed operating system from the Hebrew University of Jerusalem. MOSIX is an ongoing project which began in 1981 and released its most recent version in 1996. Automatic load balancing between MOSIX nodes is done by process migration. Other interesting features

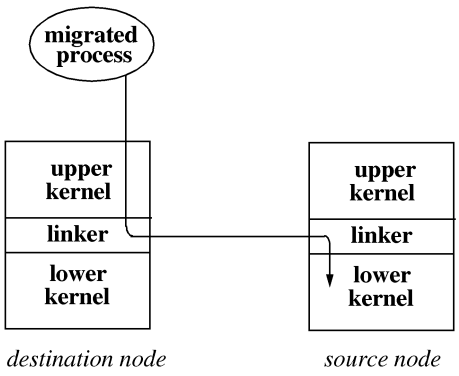

Fig. 6. The MOSIX Architecture.

include full autonomy of each node in the system, fully-decentralized control, single system image, dynamic configuration and scalability.

Various versions of MOSIX have been in active use at the Hebrew University since 1983. The original version of MOSIX was derived from UNIX Version 7 and ran on a cluster of PDP-11/45 nodes connected by a token passing ring [Barak and Litman, 1985]. The version of MOSIX documented in the MOSIX book is a cluster of multiprocessor workstations which used a UNIX System V.2 code base [Barak and Wheeler, 1989; Barak et al., 1993]. The most recent version, developed in 1993, is called NOW MOSIX [Barak et al., 1995]. This version enhances BSDI UNIX by providing process migration on a cluster of Intel Pentium processor based workstations.

Goals of the MOSIX system include:

- *Dynamic process migration*. At context switch time, a MOSIX node may elect to migrate any process to another node. The migrated process is not aware of the migration.
- *Single system image*. MOSIX presents a process with a uniform view of the file system, devices and networking facilities regardless of the process's current location.
- *Autonomy of each node*. Each node in the system is independent of all other nodes and may selectively participate in the MOSIX cluster or deny services to other nodes. Diskless nodes in MOSIX rely on a specific node for file services.
- *Dynamic configuration*. MOSIX nodes may join or leave a MOSIX cluster at any time. Processes that are not running on a node or using some node specific resource, are not affected by the loss of that node.
- *Scalability*. System algorithms avoid using any global state. By avoiding dependence on global state or centralized control, the system enhances its ability to scale to a large number of nodes.

Design. The system architecture separates the UNIX kernel into a *lower* and an *upper kernel*. Each object in MOSIX, like an open file, has a universal object pointer that is unique across the MOSIX domain. Universal objects in MOSIX are kernel objects (e.g. a file descriptor entry) that can reference an object anywhere in the cluster. For example, the upper kernel holds a universal object for an open file; the universal object migrates with the process while only the host of the file has the local, non-universal file information. The upper kernel provides a traditional UNIX system interface. It runs on each node and handles only universal objects. The lower kernel provides normal services, such as device drivers, context switching, and so on without having any knowledge or dependence on other nodes. The third component of the MOSIX system is the *linker,* which maps universal objects into local objects on a specific node, and which provides internode communication, data transfer, process migration and load balancing algorithms. When the upper kernel needs to perform an operation on one of the universal objects that it is handling, it uses the linker to perform a remote kernel procedure call on the object's host node.

MOSIX transfers only the dirty pages and user area of the migrating process at the time of the migration, an *eager (dirty) transfer* strategy. Text and other clean pages are faulted in as needed once the process resumes execution on the target node.

Process migration in MOSIX is a common activity. A process has no explicit

knowledge about what node it is actually running on or any guarantees that it will continue to run on its current node. The migration algorithm is cooperative: for a process to migrate to a node, the target node must be willing to accept it. This allows individual nodes control over the extent of their own contribution to the MOSIX system. Individual nodes can also force all active processes to migrate away, a procedure that is used when shutting down an individual node.

Process migration in MOSIX relies on the fact that the upper kernel context of each process is site-independent: regardless of where the process physically runs, its local upper kernel and linker route each system call to the appropriate node. If the process decides to migrate to a new node, the migration algorithm queries the new node to ensure that it is willing to accept a new process. If so, the upper kernel invokes a series of remote kernel procedure calls that create an empty process frame on the new node, moves the upper kernel context and any dirty pages associated with the process and then resumes the process on the new node.

Fault Resilience. Failed nodes in MOSIX affect only processes running on the failed node or directly using resources provided by the node. Nodes dynamically join and leave a MOSIX cluster at will. Detection of stale objects—those that survive past the reboot of the object's server—is done by maintaining per object version numbers. (As an example of a stale object, a universal pointer to a file object must be reclaimed after the home node for the file reboots.) Migrated processes leave no traces on other nodes.

Transparency. Migration is completely transparent in MOSIX, except for processes that use shared memory and are not eligible for migration. Full single system image semantics are presented by MOSIX, making processes unaware of their actual physical node. A new system call, *migrate()*, was added to allow processes to determine the current location or to request migration to a specified node.

Scalability. MOSIX was designed as a scalable system. The system relies on no centralized servers and maintains no global information about the system state. Each MOSIX node is autonomous and can dynamically join or withdraw from the MOSIX system. No remote system operations involve more than two nodes: the initiating node and the node providing the service. The process migration and load balancing algorithms also support scalability: load information is totally decentralized. Currently, an 80-node MOSIX system is running at Hebrew University.

Load Information Management and Distributed Scheduling. Several types of information are managed by MOSIX in order to implement its dynamic load balancing policy: the load at each node, individual process profiling, and load information about other nodes in the system.

Each node computes a local load estimate that reflects the average length of its ready queue over a fixed time period. By selecting an appropriate interval, the impact of temporary local load fluctuations is reduced without presenting obsolete information.

For each process in the system, an execution profile is maintained which reflects its usage of remote resources like files or remote devices, communication patterns with other nodes, how long this process has run and how often it has created new child processes via the *fork()* system call. This information is useful in determining where a process should migrate to when selected for migration. For example, a small process that is making heavy use of a network interface or file on a specific node would be considered for migration to that node. This profiling information is discarded when a process terminates.

The MOSIX load balancing algorithm is decentralized. Each node in the system maintains a small load information vector about the load of a small subset of other nodes in the system [Barak et al., 1989]. On each iteration of the algorithm, each node randomly selects two other nodes, of which at least one node is known to have been recently alive. Each of the selected nodes is sent the most recent half of the

local load vector information. In addition, when a load information message is received, the receiving node acknowledges receipt of the message by returning its own load information back to the sending node.

During each iteration of the algorithm, the local load vector is updated by incorporating newly received information and by aging or replacing older load information. To discourage migration between nodes with small load variations, each node adjusts its exported local load information by a stability factor. For migration to take place, the difference in load values between two nodes must exceed this stability value.

The load balancing algorithm decides to migrate processes when it finds another node with a significantly reduced load. It selects a local process that has accumulated a certain minimum amount of runtime, giving preference to processes which have a history of forking off new subprocesses or have a history of communication with the selected node. This prevents short-lived processes from migrating.

Implementation and Performance. Porting the original version of MOSIX to a new operating system base required substantial modifications to the OS kernel in order to layer the code base into the three MOSIX components (linker, lower and upper kernels). Few changes took place at the low level operating system code [Barak and Wheeler, 1989].

In order to reduce the invasiveness of the porting effort, a completely redesigned version of NOW MOSIX was developed for the BSDI version of UNIX [Barak et al., 1995]. The NOW MOSIX provides process migration and load balancing. without a single system image. As in Sprite, system calls that are location sensitive are forwarded to the *home* node of a migrated process as required (cf. Section 5.2).

The performance of a migrated process in MOSIX depends on the nature of the process. One measurement of the effect that migration has on a process is the slower performance of remote system calls. Using the frequencies of system calls measured by Douglis and Ousterhout [1987], system calls were 2.8 times slower when executed on a remote 33MHz MOSIX node [Barak et al., 1989]. Table 1 shows the measured performance and slowdown of several commonly used system calls. Many system calls, for example *getpid ()*, are always performed on the processe's current node and have no remote performance degradation.

Table 1. MOSIX System Call Performance

System Call	Local	Remote	Slowdown
read (1K)	0.34	1.36	4.00
write (1K)	0.68	1.65	2.43
open/close	2.06	4.31	2.09
fork (256 Kb)	7.8	21.60	2.77
exec (256 KB)	25.30	51.50	2.04

The performance of the MOSIX migration algorithm depends directly on the performance of the linker's data transfer mechanism on a given network and the size of the dirty address space and user area of the migrating process. The measured performance of the VME-based MOSIX migration, from one node of the cluster to the bus master, was 1.2 MB/second. The maximum data transfer speed of the system's VME bus was 3 MB/second.

Some applications benefit significantly from executing in parallel on multiple nodes. In order to allow such applications to run on a system without negatively impacting everyone else, one needs process migration in order to be able to rebalance loads when necessary. Arguably the most important performance measurement is the measurement of an actual user-level application. Specific applications, for example an implementation of a graph coloring algorithm, show a near-linear speedup with increasing number of nodes [Barak et al., 1993]. Of course, this speedup does not apply to other types of applications (non-CPU-bound, such as network or I/O bound jobs). These applications may experience different speedups. No attempt has been conducted to measure an average speedup for such types of applications.

Lessons Learned. The MOSIX system demonstrated that dynamic load balancing implemented via dynamic process

migration is a viable technology for a cluster of workstations. The earlier MOSIX implementations required too many changes to the structure of the base operating system code in order to maintain the single system image nature of the system. Giving up the single system image while preserving process migration delivers most of the benefits of the earlier MOSIX systems without requiring invasive kernel changes.

5.2. Sprite

The Sprite Network Operating System was developed at U.C. Berkeley between 1984 and 1994 [Ousterhout et al., 1988]. Its primary goal was to treat a network of personal workstations as a time-shared computer, from the standpoint of sharing resources, but with the performance guarantees of individual workstations. It provided a shared network file system with a single-system image and a fully-consistent cache that ensured that all machines always read the most recently written data [Nelson et al., 1988]. The kernel implemented a UNIX-like procedural interface to applications; internally, kernels communicated with each other via a kernel-to-kernel RPC. User-level IPC was supported using the file system, with either pipes or a more general mechanism called *pseudo-devices* [Welch and Ousterhout, 1988]. Virtual memory was supported by paging a process's heap and stack segments to a file on the local disk or a file server.

An early implementation of migration in Sprite [Douglis and Ousterhout, 1987] suffered from some deficiencies [Douglis, 1989]:

- processes accessing some types of files, such as pseudo-devices, could not be migrated;
- there was no automatic host selection; and
- there was no automatic failure recovery.

After substantial modifications to the shared file system to support increased transparency and failure recovery [Welch, 1990], migration was ported to Sun-3 workstations, and later Sparcstation and DECstation machines. Automatic host selection went through multiple iterations as well, moving from a shared file to a server-based architecture. Migration was used regularly starting in the fall of 1988.

Goals:

- *Workstation autonomy*. Local users had priority over their workstation. Dynamic process migration, as opposed to merely remote invocation, was viewed primarily as a mechanism to evict other users' processes from a personal workstation when the owner returned. In fact, without the assurance of local autonomy through process migration, many users would not have allowed remote processes to start on their workstation in the first place.
- *Location transparency*. A process would appear to run on a single workstation throughout its lifetime.
- *Using idle cycles*. Migration was meant to take advantage of idle workstations, but not to support full load balancing.
- *Simplicity*. The migration system tried to reuse other support within the Sprite kernel, such as demand paging, even at the cost of some performance. For example, migrating an active process from one workstation to another would require modified pages in its address space to be written to a file server and faulted in on the destination, rather than sent directly to the destination.

Design. Transparent migration in Sprite was based on the concept of a *home machine*. A *foreign* process was one that was not executing on its home machine. Every process appeared to run on its home machine throughout its lifetime, and that machine was inherited by descendants of a foreign process as well. Some location-dependent system calls by a foreign process would be forwarded automatically, via kernel-to-kernel RPC, to its home; examples include calls dealing with the time-of-day clock and process groups. Numerous other calls, such as *fork* and *exec*, required cooperation between the remote and home machines. Finally, location-independent calls, which

included file system operations, could be handled locally or sent directly to the machine responsible for them, such as a file server.

Foreign processes were subject to eviction—being migrated back to their home machine—should a local user return to a previously idle machine. When a foreign process migrated home, it left no residual dependencies on its former host. When a process migrated away from its home, it left a shadow process there with some state that would be used to support transparency. This state included such things as process identifiers and the parent-child relationships involved in the UNIX *wait* call.

As a performance optimization, Sprite supported both full process migration, in which an entire executing process would migrate, and remote invocation, in which a new process would be created on a different host, as though a *fork* and *exec* were done together (like the Locus *run* call [Walker et al., 1983]). In the latter case, state that persists across an *exec* call, such as open files, would be encapsulated and transferred, but other state such as virtual memory would be created from an executable.

When migrating an active process, Sprite writes dirty pages and cached file blocks to their respective file server(s). The address space, including the executable, is paged in as necessary. Migration in the form of remote invocation would result in dirty cached file blocks being written, but would not require an address space to be flushed, since the old address space is being discarded.

The migration algorithm consists of the following steps [Douglis, 1989]:

1. The process is signaled, to cause it to trap into the kernel.
2. If the process is migrating away from its home machine, the source contacts the target to confirm its availability and suitability for migration.
3. A "pre-migration" procedure is invoked for each kernel module. This returns the size of the state that will be transferred and can also have side effects, such as queuing VM pages to be flushed to the file system.
4. The source kernel allocates a buffer and calls encapsulation routines for each module. These too can have side effects.
5. The source kernel sends the buffer via RPC, and on the receiving machine each module de-encapsulates its own state. The target may perform other operations as a side effect, such as communicating with file servers to arrange for the transfer of open files.
6. Each kernel module can execute a "post-migration" procedure to clean up state, such as freeing page tables.
7. The source sends an RPC to tell the target to resume the process, and frees the buffer.

Fault Resilience. Sprite process migration was rather intolerant of faults. During migration, the failure of the target anytime after step 5 could result in the termination of the migrating process, for example, once its open files have been moved to the target. After migration, the failure of either the home machine or the process's current host would result in the termination of the process. There was no facility to migrate away from a home machine that was about to be shut down, since there would always be some residual dependencies on that machine.

Transparency was achieved through a conspiracy between a foreign process's current and home workstations. Operations on the home machine that involved a foreign process, such as a *ps* listing of CPU time consumed, would contact its current machine via RPC. Operations on the current host involving transparency, including all process creations and terminations, contacted the home machine. Waiting for a child, even one co-resident on the foreign machine, would be handled on the home machine for simplicity.

All IPC in Sprite was through the file system, even TCP connections. (TCP was served through user-level daemons contacted via pseudo-devices.) The shared network file system provided transparent

access to files or processes from different locations over time.

As in MOSIX, processes that share memory could not be migrated. Also, processes that map hardware devices directly into memory, such as the X server, could not migrate.

Scalability. Sprite was designed for a cluster of workstations on a local area network and did not particularly address the issue of scalability. As a result, neither did the migration system. The centralized load information management system, discussed next, could potentially be a bottleneck, although a variant based on the MOSIX probabilistic load dissemination algorithm was also implemented. In practice, the shared file servers proved to be the bottleneck for file-intensive operations such as kernel compilations with as few as 4-5 hosts, while cpu-intensive simulations scaled linearly with over ten hosts [Douglis, 1990].

Load Information Management. A separate, user-level process (*migd*) was responsible for maintaining the state of each host and allocating idle hosts to applications. This daemon would be started on a new host if it, or its host, should crash. It allocated idle hosts to requesting processes, up to one foreign "job" per available processor. (A "job" consisted of a foreign process and its descendants.) It supported a notion of *fairness*, in that one application could use all idle hosts of the same architecture but would have some of them reclaimed if another application requested hosts as well. Reclaiming due to fairness would look to the application just like reclaiming due to a workstation's local user returning: the foreign processes would be migrated home and either run locally, migrated elsewhere, or suspended, depending on their controlling task's behavior and host availability.

Migration was typically performed by *pmake*, a parallel *make* program like many others that eventually became commonplace (e.g., [Baalbergen, 1988]) *Pmake* would use remote invocation and then remigrate processes if *migd* notified it that any of its children were evicted. It would suspend any process that could not be remigrated.

Implementation and Performance. Sprite ran on Sun (Sun 2, Sun 3, Sun 4, SPARCstation 1, SPARCstation 2) and Digital (DECstation 3100 and 5100) workstations. The entire kernel consisted of approximately 200,000 lines of heavily commented code, of which approximate 10,000 dealt with migration.

The performance of migration in Sprite can be measured in three respects. All measurements in this subsection were taken on SPARCstation 1 workstations on a 10-Mbps Ethernet, as reported in [Douglis and Ousterhout, 1991].

1. The time to **migrate a process** was a function of the overhead of host selection (36ms to select a single host, amortized over multiple selections when migration is performed in parallel); the state for each open file (9.4ms/file); dirty file and VM blocks that must be flushed (480-660 Kbytes/second depending on whether they are flushed in parallel); process state such as *exec* arguments and environment variables during remote invocation (also 480 Kbytes/second); and a basic overhead of process creation and message traffic (76ms for the null process).
2. A process that had migrated away from its home machine incurred **run-time overhead** from forwarding location-dependent system calls. Applications of the sort that were typically migrated in Sprite, such as parallel compilation and LaTeX text processing, incurred only 1-3% degradation from running remotely, while other applications that invoked a higher fraction of location-dependent operations (such as accessing the TCP daemon on the home machine, or forking children repeatedly) incurred substantial overhead.
3. Since the purpose of migration in Sprite was to enable parallel use of many workstations, **application speedup** is an important metric. Speedup is affected by a number of factors, including the degree of parallelism, the load on central resources such as the

migd daemon, and inherently nonparallelizable operations. By comparing the parallel compilation of several source directories, ranging from 24 to 276 files and 1 to 3 independent link steps, one found that the speedup compared to the sequential case ranged from about 3 to 5.4 using up to 12 hosts, considerably below linear speedup. During a 12-way *pmake*, the processors on both the server storing the files being read and written, and the workstation running *pmake*, were saturated. Network utilization was not a significant problem, however.

Lessons Learned. Here we summarize the two most important lessons and experiences in Sprite process migration [Douglis, 1990; Douglis and Ousterhout, 1991].

- Migration provided a considerable source of processor cycles to the Sprite community. Over a one-month period, 30% of user processor activity came from migrated (foreign) processes. The host that accounted for the greatest total usage (nearly twice as many cpu-seconds as the next greatest) ran over 70% of its cycles on other hosts.
- Evictions accounted for 6% of all migrations, with about half of these evictions due to fairness considerations and the other half due to users reclaiming their machines. About 1% of all host allocations were revoked for one of these two reasons. (Evictions counted for a relatively higher fraction of all migrations because one host revocation could result in many processes being migrated.)

5.3. Mach

Mach is a microkernel developed at the Carnegie Mellon University [Accetta et al., 1986; Black et al., 1992], and later at the OSF Research Institute [Bryant, 1995]. A migration mechanism on top of the Mach microkernel was developed at the University of Kaiserslautern, from 1991 to 1993 [Milojičić et al., 1993b].

Task migration was used for experiments with load distribution. In this phase, only tasks were addressed, while UNIX processes were left on the source machine, as described in Figure 7. This means that only Mach task state was migrated, whereas the UNIX process state that was not already migrated as a part of the Mach task state (e.g. state in the UNIX "personailty server" emulating UNIX on top of the Mach microkernel) remained on the source machine. Therefore, most of the UNIX system calls were forwarded back to the source machine, while only Mach system calls were executed on the destination machine.

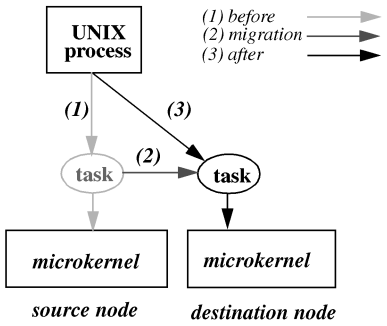

Fig. 7. Task Migration Design. Only task abstraction is migrated, while process abstraction remains on the source node.

Process migration for the OSF/1 AD 1 server [Paindaveine and Milojičić, 1996] was developed during 1994 at the Universite Catholique de Louvain, Belgium, as a part of a project on load-leveling policies in a distributed system [Jacqmot, 1996]. OSF/1 AD 1 is a version of the OSF/1 operating system which provides a scalable, high-performance single-system image version of UNIX. It is composed of servers distributed across the different nodes running the Mach microkernel. Process migration relies on the Mach task migration to migrate microkernel-dependent process state between nodes.

Mach task migration was also used at the University of Utah, for the Schizo project [Swanson et al., 1993]. Task and process migration on top of Mach were designed and implemented for clusters of workstations.

Goals. The first goal was to provide a transparent task migration at user-level

with minimal changes to the microkernel. This was possible by relying on Mach OS mechanisms, such as (distributed) memory management and (distributed) IPC. The second goal was to demonstrate that it is possible to perform load distribution at the microkernel level, based on the three distinct parameters that characterize microkernels: processing, VM and IPC.

Design. The design of task migration is affected by the underlying Mach microkernel. Mach supported various powerful OS mechanisms for purposes other than task and process migration. Examples include Distributed Memory Management (DMM) and Distributed IPC (DIPC). DIPC and DMM simplified the design and implementation of task migration. DIPC takes care of forwarding messages to migrated process, and DMM supports remote paging and distributed shared memory. The underlying complexity of message redirection and distributed memory management are heavily exercised by task migration, exposing problems otherwise not encountered. This is in accordance with earlier observations about message-passing [Douglis and Ousterhout, 1991].

In order to improve robustness and performance of DIPC, it was subsequently redesigned and reimplemented [Milojičić et al., 1997]. Migration experiments have not been performed with the improved DIPC. However, extensive experiments have been conducted with Concurrent Remote Task Creation (CRTC), an in-kernel service for concurrent creation of remote tasks in a hierarchical fashion [Milojičić et al., 1997]. The CRTC experiments are similar to task migration, because a remote *fork* of a task address space is performed.

DMM enables cross-node transparency at the Mach VM interface in support of a distributed file system, distributed file system, distributes processes, and distributed shared memory [Black, et al., 1998]. The DMM support resulted in simplified design and implementation of the functionality built on top of it, such as SSI UNIX and remote tasking, and it avoided pager modifications by interposing between the VM system and the pager. However, the DMM became too complex, and had performance and scalability problems. The particular design mistakes include the interactions between DSM support and virtual copies in a distributed system; transparent extension of Mach *copy-on-write* VM optimization to distributed systems; and limitations imposed by Mach's external memory management while transparently extending it to distributed systems. (*Copy-on-write* is an optimization introduced to avoid copying pages until it is absolutely needed, and otherwise sharing the same copy. It has also been used in Chorus [Rozier, 1992] and Sprite [Nelson and Ousterhout, 1988].)

DMM had too many goals to be successful; it failed on many general principles, such as "do one thing, but do it right," and "optimize the common case" [Lampson, 1983]. Some of the experiments with task migration reflect these problems. Variations of forking an address space and migrating a task significantly suffered in performance. While some of these cases could be improved by optimizing the algorithm (as was done in the case of CRTC [Milojičić et al., 1997]), it would only add to an already complex and fragile XMM design and implementation. Some of the DMM features are not useful for task migration, even though they were motivated by task migration support. Examples include DSM and distributed copy-on-write optimizations. DSM is introduced in order to support the transparent remote forking of address spaces (as a consequence of remote fork or migration) that locally share memory. Distributed copy-on-write is motivated by transparently forking address spaces that are already created as a consequence of local copy-on-write, as well as in order to support caching in distributed case.

Even though the DIPC and DMM interfaces support an implementation of user-level task migration, there are two exceptions. Most of the task state is accessible from user space except for the capabilities that represent tasks and threads and capabilities for internal memory state. Two new interfaces are provided for exporting

the aforementioned capabilities into user space.

A goal of one of the user-space migration servers is to demonstrate different data transfer strategies. An external memory manager was used for implementation of this task migration server. The following strategies were implemented: eager copy, flushing, copy-on-reference, precopy and read-ahead [Milojičić et al., 1993b]. For most of the experiments, a simplified migration server was used that relied on the default in-kernel data transfer strategy, copy-on-reference.

The task migration algorithm steps are:

1. Suspend the task and abort the threads in order to clean the kernel state.[1]
2. Interpose task/thread kernel ports on the source node.
3. Transfer the address space, capabilities, threads and the other task/thread state.
4. Interpose back task/thread kernel ports on the destination node.
5. Resume the task on the destination node.

Process state is divided into several categories: the Mach task state; the UNIX process local state; and the process-relationship state. The local process state corresponds to the typical UNIX *proc* and *user* structures. Open file descriptors, although part of the UNIX process state, are migrated as part of the Mach task state.

Fault Resilience of Mach task migration was limited by the default transfer strategy, but even more by the DIPC and DMM modules. Both modules heavily employ the lazy evaluation principle, leaving residual dependencies throughout the nodes of a distributed system. For example, in the case of DIPC, proxies of the receive capabilities remain on the source node after receive capability is migrated to a remote node. In the case of DMM, the established paging paths remain bound to the source node even after eager copying of pages is performed to the destination node.

Transparency was achieved by delaying access or providing concurrent access to a migrating task and its state during migration. The other tasks in the system can access the migrating task either by sending messages to the task kernel port or by accessing its memory. Sending messages is delayed by interposing the task kernel port with an interpose port. The messages sent to the interpose port are queued on the source node and then restarted on the destination node. The messages sent to other task ports are transferred as a part of migration of the receive capabilities for these ports. Access to the task address space is supported by DMM even during migration. Locally shared memory between two tasks becomes distributed shared memory after migration of either task.

In OSF/1 AD, a virtual process (*Vprocs*) framework supports transparent operations on the processes independently of the actual process's location [Zajcew et al., 1993]. By analogy, *vprocs* are to processes what *vnodes* are to files, both providing location and heterogeneity transparency at the system call interface. Distributed process management and the single system image of Mach and OSF/1 AD eased the process migration implementation.

A single system image is preserved by retaining the process identifier and by providing transparent access to all UNIX resources. There are no forwarding stub processes or chains. No restrictions are imposed on the processes considered for migration: for example, using pipes or signals does not prevent a process from being migrated.

Scalability. The largest system that Mach task migration ran on at University of Kaiserslautern consisted of five nodes. However, it would have been possible to scale it closer towards the limits of the scalability of the underlying Mach microkernel, which is up to a couple of thousand nodes on the Intel Paragon supercomputer.

[1] Aborting is necessary for threads that can wait in the kernel arbitrarily long, such as in the case of waiting for a message to arrive. The wait operation is restartable on the destination node.

Table 2. Processing, IPC and VM Intensive Applications

Type	Application	User/Total Time	IPC (msg/s)	VM ((pagin + out)/s)
Processing	Dhrystone	**1.00**	3.49	0.35 + 0
IPC	find	0.03	**512.3**	2.75 + 0
VM	WPI Jigsaw	0.09	2.46	**28.5 + 38.2**

Migration of the address space relies heavily on the Mach copy-on-write VM optimization, which linearly grows the internal VM state for each migration [Milojičić et al., 1997]. In practice, when there are just few migrations, this anomaly is not noticeable. However for many consecutive migrations it can reduce performance.

Load Information and Distributed Scheduling. Mach was profiled to reflect remote IPC and remote paging activity in addition to processing information. This information was used to improve load distribution decisions. [Milojičić, 1993c]. Profiling was performed inside of the microkernel by collecting statistics for remote communication and for remote paging and in user space, by interposing application ports with profiler ports.

A number of applications were profiled and classified in three categories: processing, communication and paging intensive. Table 2 gives representatives of each class.

Extended load information is used for applying more appropriate distributed scheduling decisions [Milojičić et al., 1993a]. An application that causes a significant amount of remote paging, or communicates with another node, is considered for migration to the appropriate node. CPU-bound applications have no such preference and can be migrated based only on the processing load criteria. For applications consisting of a number of processes that communicate among themselves, improvements achieved by considering IPC/VM information in addition to CPU load is proportional to the load and it can reach up to 20-50% for distributed scheduling strategies [Milojičić et al., 1993a]. Improvements of the performance of a simple application due to locality of reference can be multifold [Milojičić et al., 1993b].

Implementation and Performance. Milojičić et al. built three implementations: two user-level migrations (an optimized and a simple migration server); and a kernel implementation. The size of the simplified migration server is approximately 400 lines of code that took about 3 months to design and implement. A lot of this time was spent in debugging the DIPC parts of code that were never before exercised. Task migration, especially load distribution experiments using task migration, turned out to be a very stressful test for DIPC.

The size of the in-kernel version is close to the simplified migration server, from which it was derived. These two implementations relied on the in-kernel support for address space transfer. However, the size of the DIPC and DMM modules was significantly higher. One of the latest versions of optimized DIPC (nmk21b1) consisted of over 32,000 lines of code. It took over 10 engineer-years to release the second version of DIPC. The DMM, which was never optimized, consists of 24,000 lines of code.

The optimized migration server is largest in size with a few thousand lines of code. Most of this implemented a pager supporting different data transfer strategies. The optimized migration server did not rely on in-kernel data transfer strategy, except for the support of distributed shared memory.

Although there is an underlying distributed state in the microkernel, no distributed state is involved in the process migration facility at the server level, rendering the design of the migration mechanism simple. The process migration code consists of approximately 800 lines of code. However, adding distributed process management requires about 5000 lines of additional code. The main (initial and runtime)

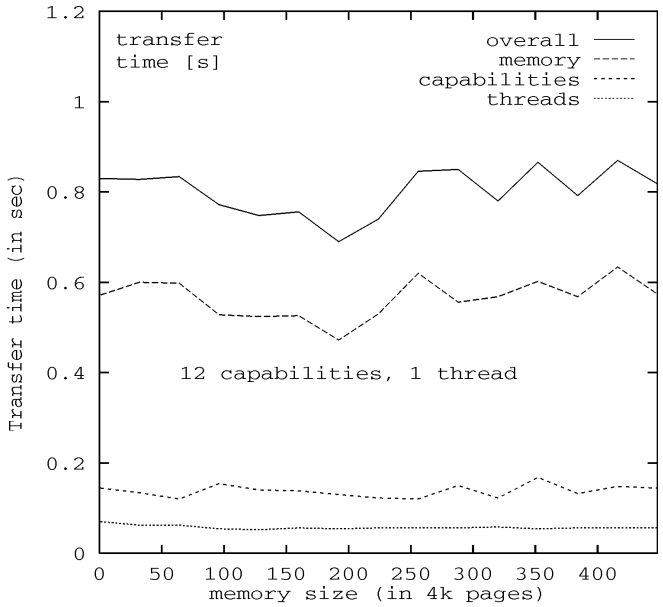

Fig. 8. Task migration performance as a function of VM size: *initial costs are independent of task address space size (aside of variations due to other side effects).*

costs of migration are due to task migration. Process migration has very little overhead in addition to task migration.

Performance measurements were conducted on a testbed consisting of three Intel 33MHz 80486 PCs with 8MB RAM. The NORMA14 Mach and UNIX server UX28 were used. Performance is independent of the address space size (see Figure 8), and is a linear function of the number of capabilities. It was significantly improved in subsequent work [Milojičić et al., 1997].

Lessons Learned

- Relying on DIPC and DMM is crucial for the easy design and implementation of transparent task migration, but these modules also entail most of the complexity and they limit performance and fault resilience.
- Task migration is sufficient for microkernel applications. In contrast, as mentioned above, UNIX applications would forward most system calls back to the source node, resulting in an order-of-magnitude performance degradation. Migrating the full UNIX process state would presumably have alleviated this overhead, similar to the evolution in Sprite toward distinguishing between location-dependent and location-independent calls [Douglis, 1989].
- Applications on microkernels can be profiled as a function of processing, IPC and VM and this information can be used for improved load distribution. Improvement ranges from 20-55% for collaborative types of applications.

5.4. LSF

LSF (Load Sharing Facility) is a load sharing and batch scheduling product from Platform Computing Corporation [Platform Computing, 1996]. LSF is based on the Utopia system developed at the University of Toronto [Zhou et al., 1994], which is in turn based on the earlier Ph.D. thesis work of Zhou at UC Berkeley [Zhou, 1987; Zhou and Ferrari, 1988].

LSF provides some distributed operating system facilities, such as distributed process scheduling and transparent remote execution, on top of various

operating system kernels without change. LSF primarily relies on initial process placement to achieve load balancing, but also uses process migration via checkpointing as a complement. LSF currently runs on most UNIX-based operating systems.

Checkpointing and Migration Mechanisms. LSF's support for process user-level process migration is based on Condor's approach [Litzkow and Solomon, 1992]. A checkpoint library is provided that must be linked with application code. Part of this library is a signal handler that can create a checkpoint file of the process so that it can be restarted on a machine of compatible architecture and operating system. Several improvements have been made to the original Condor checkpoint design, such as:

- No *core* dump is required in order to generate the checkpoint file. The running state of the process is directly extracted and saved in the checkpoint file together with the executable in a format that can be used to restart the process. This not only is more efficient, but also preserves the original process and its ID across the checkpoint.
- UNIX signals can be used by the checkpointed process. The state information concerning the signals used by the process is recorded in the checkpoint file and restored at restart time.

In addition to user-level transparent process checkpointing, LSF can also take advantage of checkpointing already supported in the OS kernel (such as in Cray Unicos and ConvexOS), and application-level checkpointing. The latter is achievable in classes of applications by the programmer writing additional code to save the data structures and execution state information in a file that can be interpreted by the user program at restart time in order to restore its state. This approach, when feasible, often has the advantage of a much smaller checkpoint file because it is often unnecessary to save all the dirty virtual memory pages as must be done in user-level transparent checkpointing. Application-level checkpointing may also allow migration to work across heterogeneous nodes.

The checkpoint file is stored in a user-specified directory and, if the directory is shared among the nodes, the process may be restarted on another node by accessing this file.

Load Information Exchange. Similar to Sprite, LSF employs a centralized algorithm for collecting load information. One of the nodes acts as the master, and every other node reports its local load to the master periodically. If the master node fails, another node immediately assumes the role of the master. The scheduling requests are directed to the master node, which uses the load information of all the nodes to select the one that is likely to provide the best performance.

Although many of the load information updates may be wasted if no process need to be scheduled between load information updates, this algorithm has the advantage of making (reasonably up-to-date) load information of all nodes readily available, thus reducing the scheduling delay and considering all nodes in scheduling. Zhou et al. [1994] argue that the network and CPU overhead of this approach is negligible in modern computers and networks. Measurements and operational experience in clusters of several hundred hosts confirm this observation. Such a centralized algorithm also makes it possible to coordinate all process placements - once a process is scheduled to run on a node, this node is less likely to be considered for other processes for a while to avoid overloading it. For systems with thousands of nodes, several clusters can be formed, with selective load information exchange among them.

Scheduling Algorithms. LSF uses checkpoint and restart to achieve process migration, which in turn is used to achieve load balancing. If a node is overloaded or needed by some higher priority processes, a process running on it may be migrated to another node. The load conditions that trigger process migration can be configured to be different for various types of jobs. To avoid undesirable migration due

to temporary load spikes and to control migration frequency, LSF allows users to specify a time period for which a process is suspended on its execution node. Only if the local load conditions remain unfavorable after this period would the suspended process be migrated to another node.

The target node is selected based on the dynamic load conditions and the resource requirements of the process. Recognizing that different processes may require different types of resources, LSF collects a variety of load information for each node, such as average CPU run queue length, available memory and swap space, disk paging and I/O rate, and the duration of idle period with no keyboard and mouse activities. Correspondingly, a process may be associated with resource requirement expressions such as

```
select[sparc && swap >= 120 &&
    mem >= 64] order[cpu : mem]
```

which indicates that the selected node should have a resource called "sparc," and should have at least 120 MB of swap space and 64 MB of main memory. Among the eligible nodes, the one with the fastest, lightly loaded CPU, as well as large memory space, should be selected. A heuristic sorting algorithm is employed by LSF to consider all the (potentially conflicting) resource preferences and select a suitable host. Clearly, good host allocation can only be achieved if the load condition of all nodes is known to the scheduler.

The resource requirements of a process may be specified by the user when submitting the process to LSF, or may be configured in a system process file along with the process name. This process file is automatically consulted by LSF to determine the resource requirement of each type of process. This process file also stores information on the eligibility of each type of process for remote execution and migration. If the name of a process is not found in this file, either it is excluded from migration consideration, or only nodes of the same type as the local node would be considered.

Process Migration vs. Initial Placement. Although LSF makes use of process migration to balance the load, it is used more as an exception rather than the rule, for three reasons. First, transparent user-level checkpointing and migration are usable by only those processes linked with the checkpoint library, unless the OS kernel can be modified; in either case, their applicability is limited. Secondly, intelligent initial process placement has been found to be effective in balancing the load in many cases, reducing the need for migration [Eager et al., 1988]. Finally, and perhaps most importantly, the same load balancing effect can often be achieved by process placement with much less overhead. The remote process execution mechanism in LSF maintains the connection between the application and the Remote Execution Server on the execution node and caches the application's execution context for the duration of the application execution, so that repeated remote process executions would incur low overhead (0.1 seconds as measured by Zhou et al. on a network of UNIX workstations [1994]).

In contrast, it is not desirable to maintain per-application connections in a kernel implementation of process migration to keep the kernel simple, thus every process migration to a remote node is "cold". Per-application connections and cached application state are rather "heavyweight" for kernel-level migration mechanisms, and the kernel-level systems surveyed in this paper treat each migration separately (though the underlying communication systems, such as kernel-to-kernel RPC, may cache connection state). The benefits of optimizing remote execution are evident by comparing LSF to an earlier system such as Sprite. In the case of Sprite, the overhead of exec time migration was measured to be approximately 330ms on Sparcstation 1 workstations over the course of one month [Douglis and Ousterhout, 1991]. Even taking differences in processor speed into account as well as underlying overheads such as file system cache flushing, LSF shows a marked improvement in remote invocation performance.

6. COMPARISON

In this section, we compare the various migration implementations described in the paper. We cover the case studies, as well as some other systems mentioned in Section 4.

Table 3 summarizes the process migration classification provided in Section 4. We mention **examples** of each class of migration, followed by the **main characteristic** of each class. These columns are self-explanatory. The **OS v. Application Modification** column describes where the majority of modifications to support migration are performed. Migration in the early work, UNIX-like systems, message passing and microkernels require modifications to the underlying OS. User-space and application-specific systems require modifications to the application, typically relinking and in certain cases also recompiling. Mobile objects and agents require modifications to the underlying programming environment. However, they also have the least transparency, as described below.

The **Migration Complexity** column describes the amount of effort required to design and implement migration. Complexity is high for kernel implementations. Exceptions are message-passing kernels, which already provide much of the required functionality in their support for message passing. This results in a simpler migration mechanism. Microkernels also support migration more easily because of simpler abstractions and reduced functionality (for example, no UNIX compatibility). However, extensive complexity is introduced for supporting distributed IPC and distributed memory management. The least complex implementations are those done at user level and those done as part of an application.

The "other complexity" subfield describes where the complexity in the system exists. Early work incurred complexity in infrastructure support for the underlying hardware and software, such as Alto computers in the case of Worms, and the X-Tree architecture in the case of XOS. Transparent migration on UNIX-like systems incurs a lot of complexity for the support of Single System Image and extending UNIX semantics to a distributed system. As already pointed out, message passing typically requires a lot of complexity; examples include Charlotte and the V kernel, as well as some of the microkernels, such as Mach and Chorus. In addition, some of the microkernels (e.g. Mach) also support distributed memory management, which is even harder to support. User-space migrations trade off the simplicity of the underlying support for redirecting system calls or imposing limits on them. Application-specific migrations require knowledge of the application semantics in order to integrate migration calls at appropriate places.

Extensibility describes how easy it is to extend a process migration implementation. Examples include support for multiple data transfer and location strategies. In most cases, extensibility is inversely proportional to complexity. An exception to this rule are message-passing kernels, which have simple migration implementations, but are not as extensible. Extensions to a migration mechanism for performance and improved fault resilience typically require complex changes to the underlying mechanism for message passing. **Portability** describes how easy it is to port the migration mechanism to another operating system or computer. User-space and application-specific implementations have superior portability. Condor and LSF run on numerous versions of operating systems and computers. Kernel-level implementations are typically closely related to the underlying system and consequently their portability is limited to the portability of the operating system. For example Mach and MOSIX were ported to a number of computer architectures.

It is hard to compare the **performance** of various migration mechanisms because the implementations were done on a number of different architectures. It is also hard and inaccurate to normalize performance (some attempts toward normalizations were done by Roush [1995]). Therefore, we have not provided a column describing performance. Nevertheless,

Table 3. Summary of the Different Migration Implementations

Migration/ Characteristics	Examples	Main Characteristics	OS v. Appl. Modification	Migration Complexity (Other Complexity)	Extensibility & Portability	Transparency
Early Work	XOS, Worm, DEMOS, Butler	ad-hoc solutions, HW dependent	OS	low (lack of infrastructure)	poor	limited
Transp. Migration in UNIX-like OS	Locus, MOSIX, Sprite	major changes to the underlying env.	OS	high (Supporting SSI)	fair (OS depend.)	full
Message-Passing OS	Charlotte, Accent, V Kernel	complex OS support easy PM implement.	OS	low (Message Passing)	fair (OS depend.)	full
Microkernels	Amoeba, Arcade, BirliX, Chorus, Mach, RHODOS	no UNIX semantics complex OS support	OS	low (DMM and DIPC)	good (OS depend.)	full
User Space	Condor, Alonso & Kyrimis, Mandelberg, LSF	less transparency	application (relinked)	low (forwarding system calls)	very good (appl. dep.)	limited
Application	Freedman, Skordos, Bharat & Cardelli	min. transparency, more appl. knowledge	application (recompiled)	lowest (app migration awareness)	very good	minimal
Mobile objects	Emerald, SOS, COOL	object oriented	programming environment	moderate (communication)	good	full
Mobile Agents	Agent-TCL, Aglets TACOMA, Telescript	heterogeneity	programming environment	lowest (security & safety)	good	fair

Table 4. Transparency "Checklist"

Migration/ Supported	Open Files	Fork Children	Communication Channels	Need to Relink Application	Changes to Kernel	Shared Memory
MOSIX	yes	yes	yes	no	yes	no
Sprite	yes	yes	yes	no	yes	no
Mach & OSF/1 AD	yes	yes	yes	no	yes	yes
LSF	some	no	no	yes	no	no

we note that the performance of user- and application-level migrations typically fall in the range of seconds, even minutes, when migrating processes with large address spaces. The kernel supported migrations, especially the newer implementations, fall in the range of tens of milliseconds. The most optimized kernel implemented migration (Choices) has initial costs of only 14ms [Roush and Campbell, 1996], and it is better even if some rough normalization is accounted for (see [Roush, 1995]).

As mentioned earlier, the dominant performance element is the cost to transfer the address space. Kernel-level optimizations can cut down this cost, whereas user-level implementations do not have access to the relevant data structures and cannot apply these optimizations.

Recently, trends are emerging that allow users more access to kernel data, mechanism, and policies [Bomberger et al., 1992]. For example, microkernels export most of the kernel state needed for user-level implementations of migration [Milojičić, 1993c]. Extensible kernels provide even more support in this direction [Bershad et al., 1995; Engler et al., 1995]. These trends decrease the relevance of user versus kernel implementations.

Transparency describes the extent to which a migrated process can continue execution after migration as if migration has not happened. It also determines whether a migrated process is allowed to invoke all system functions. Many user- or application-level implementations do not allow a process to invoke all system calls. Migration that is implemented inside the kernel typically supports full functionality. In general, the higher the level of the implementation, the less transparency is provided. User-space implementations are aware of migration and they can invoke migration only at predefined places in the code. Kernel-supported implementations typically have higher levels of transparency. Single system image supports transparent migration at any point of application code; migration can transparently be initiated either by the migrating process or by another process. Most mobile agent implementations do not allow transparent migration invocation by other applications; only the migrating agent can initiate it. Even though less transparent, this approach simplifies implementation.

More specifics on transparency in the case studies are presented in Table 4. Migration for each case study is categorized by whether it transparently supports open files, forking children, communication channels, and shared memory. If migration requires changes to the kernel or relinking the application, that is also listed.

Support for shared memory of migrated tasks in Mach is unique. In practice, it was problematic due to a number of design and implementation issues [Black et al. 1998]. Other systems that supported both shared memory and migration either chose not to provide transparent access to shared memory after migration (e.g. Locus [Walker and Mathews, 1989; Fleisch and Popek, 1989]), or disallowed migration of processes using shared memory (e.g., Sprite [Ousterhout et al., 1988]).

Kernel-level migration typically supports all features transparently, whereas user-level migrations may limit access to NFS files and may not support communication channels or interprocess communication. In addition, a user-level migration typically requires relinking applications with special libraries. Migration done as

Table 5. Summary of Various Data Transfer Strategies

Data Transfer Strategy	Example	Freeze Time	Freeze Costs	Residual Time & Costs	Residual Dependency	Initial Migration Time
eager (all)	most user-level and early migrations	high	high	none	none	high
eager (dirty)	MOSIX, Locus	moderate	moderate	none	none	moderate
precopy	V kernel	very low	high	none	none	high
copy on reference	Accent, Mach	low	small	high	yes	low
flushing	Sprite	moderate	moderate	moderate	none	moderate

part of an application requires additional re-compilation.

In Table 5, we compare different data transfer strategies with respect to **freeze time, freeze costs, residual time and costs, residual dependencies**, and **initial migration time** (time passed since request for migration until process started on remote node).

We can see that different strategies have different goals and introduce different costs. At one end of the spectrum, systems that implement an *eager (all)* strategy in user space eliminate residual dependencies and residual costs, but suffer from high freeze time and freeze costs.

Modifying the operating system allows an *eager (dirty)* strategy to reduce the amount of the address space that needs to be copied to the subset of its dirty pages. This increases residual costs and dependencies while reducing freeze time and costs.

Using a *precopy* strategy further improves freeze time, but has higher freeze costs than other strategies. Applications with real-time requirements can benefit from this. However, it has very high migration time because it may require additional copying of already transferred pages.

Copy on reference requires the most kernel changes in order to provide sophisticated virtual mappings between nodes. It also has more residual dependencies than other strategies, but it has the lowest freeze time and costs, and migration time is low, because processes can promptly start on the remote node.

Finally, the *flushing* strategy also requires some amount of change to the kernel, and has somewhat higher freeze time than copy-on-reference, but improves residual time and costs by leaving residual dependencies only on a server, but not on the source node. Process migration in the Choices system, not listed in the table, represents a highly optimized version of *eager (dirty)* strategy.

The data transfer strategy dominates process migration characteristics such as performance, complexity, and fault resilience. The costs, implementation details and residual dependencies of other process elements (e.g. communication channels, and naming) are also important but have less impact on process migration.

In the Mach case study, we saw that most strategies can be implemented in user space. However, this requires a pager-like architecture that increases the complexity of OS and migration design and implementation.

Table 6 summarizes load information database characteristics. **Database type** indicates whether the information is maintained as a distributed or a centralized database. Centralized databases have shown surprising scalability for some systems, in particular LSF. Nevertheless, achieving the highest level of scalability requires distributed information management.

Maximum nodes deployed is defined as the number of nodes that were actually used. It is hard to make predictions about the scalability of migration and load information management. An

Table 6. Load Information Database

Migration/ Charact.	Database type	Maximum Nodes Deployed	Database Scope	Fault Tolerance	Knowledge Relevance
MOSIX	distributed	64	partial	yes	aging
Sprite	centralized	30	global	limited	verification, update on state change or periodic
Mach	distributed	5	global	no	negotiation
LSF	centralized	500	global	yes	none

approximate prediction is that centralized load information management could scale up to 500 nodes without hierarchical organization, such as in Sprite. With hierarchical organization, such as in LSF, it could scale beyond 2000 nodes. Decentralized information management, such as in MOSIX, can scale to an even larger number of nodes. Even though Mach task migration has not been used on larger systems than a 5-node Ethernet cluster, most of its components that can impact scalability (distributed IPC, distributed memory management, and remote address space creation) have been demonstrated to scale well. The Intel Paragon computer, the largest MMP machine that runs Mach, has over 2000 nodes [Zajcew et al., 1993]. However, in order to use migration for load distribution some decentralized information management algorithm would have to be deployed, similar to the one used for TNC.

Database scope defines the amount of information that is considered. Some systems, like MOSIX, maintain partial system information in order to enhance scalability. Large systems need to address **fault tolerance**. One drawback of centralized databases is that storing the data on one node introduces a single point of failure. This problem can be alleviated by replicating the data.

Once knowledge about the state of a system is collected and disseminated, it starts to lose its relevance as the state of the system changes. This is an intrinsic characteristic of distributed systems. The last column, **knowledge relevance**, lists the methods used by the load information management modules to account for this.

Table 7 describes the type of information managed by load information collection and dissemination. Load information is maintained for each process in a system, as well as for each machine in a system. **Process parameters** lists the information gathered about individual processes while **system parameters** shows the information gathered per node. All systems use processor utilization information while some of them also consider system communication patterns and access to off-node devices.

Disseminated parameters describes how much of the information is passed on to other nodes in a system. In most cases, only system information is disseminated, and the average ready queue length is of most interest. In some systems, not all available information is retained, as described in the **retained information** column. For example, MOSIX retains only a subset of collected information during dissemination phase. **Negotiation parameters** details the information exchanged at the time of an attempted migration. Process parameters are used during negotiation phase. Finally, the **collection** and **dissemination** columns detail the frequency of collection and dissemination in four case studies. In all cases both collection and dissemination are periodic, with the exception of Sprite—it also disseminates upon a state change.

Table 8 summarizes the characteristics of distributed scheduling. The **migration** class column indicates the type of migration mechanism employed. The **considered costs** column indicates whether and how systems weigh the actual cost of migration. The migration costs are considered in the case of MOSIX and LSF, and not in the case of Sprite and Mach. In addition to CPU costs, MOSIX also accounts for communication costs.

Migration trigger summarizes the reasons for migration activation.

Table 7. Load Information Collection and Dissemination

Migration/ Charact.	Per Process Parameters	System Parameters (also disseminated)	Retained Information	Negotiation Parameters	Collection -periodic (freq.) -event driv. (event)	Dissemination -periodic (freq.) -event driv. (event)
MOSIX	age, I/O patterns, file access	average ready queue	partial (random subset)	migrating process may be refused	periodic	periodic (1-60s) (worm-like)
Sprite	none	time since last local user input, ready queue length	all info retained	migration version	periodic (5s)	periodic (1min) and upon a state change
Mach	age, remote IPC, and remote paging	average ready queue, remote IPC, remote paging	all info retained	destination load, free paging space	periodic (1s)	periodic (1s)
LSF	none	arbitrary configurable	all info retained	system parameters of all nodes	periodic	periodic

Table 8. Distributed Scheduling

System/ Characteristics	Migration Class	Considered Costs	Migration Trigger	A Priori Knowledge	Learning from the Past	Stability
MOSIX	process migration (UNIX-like OS)	CPU & communication	threshold cross + load difference	non eligible processes	aging load vector process residency	minimum residency node refusal info weighting
Sprite	process migration (UNIX-like OS)	no	*pmake*, migratory shell, eviction (due to user activity or fairness)	non eligible processes or list of eligible ones	none	bias toward long-idle machines
Mach	task migration (microkernel)	no	threshold cross	predefined non-eligible tasks	limit consecutive migration	high threshold
LSF	process migration (user-level migr.)	CPU overhead	configurable thresholds	predefined non-eligible commands	lowering standard deviation	high thresholds

Examples include crossing a load threshold on a single node or on demand after an application-specific request, or only for specific events like process eviction. Sprite process migration can be initiated as a part of the *pmake* program or a migratory shell, or as a consequence of the eviction of a remotely executed process.

Some of the systems use a **priori knowledge**, typically in the form of specifying which processes are not allowed to migrate. These are for example well known system processes, such as in the case of MOSIX, Sprite and Mach, or commands in the case of LSF. The **learning from the past** column indicates how some systems adapt to changing loads. Examples include aging load vectors and process residency in MOSIX, and limiting consecutive migrations in Mach. **Stability** is achieved by requiring a minimum residency for migrated processes after a migration (such as in MOSIX), by introducing a high threshold per node (such as in Mach and LSF), or by favoring long-idle machines (such as in Sprite). It can also be achieved by manipulating load information as was investigated in MOSIX. For example, dissemination policies can be changed, information can be weighed subject to current load, and processes can be refused.

7. WHY PROCESS MIGRATION HAS NOT CAUGHT ON

In this section, we attempt to identify the barriers that have prevented a wider adoption of process migration and to explain how it may be possible to overcome them. We start with an analysis of each case study; we identify misconceptions; we identify those barriers that we consider the true impediments to the adoption of migration; and we conclude by outlining the likelihood of overcoming these barriers.

7.1. Case Analysis

MOSIX. The MOSIX distributed operating system is an exception to most other systems supporting transparent process migration in that it is still in general use. Several things worked against the wider adoption of the MOSIX system: the implementation was done on a commercial operating system which prevented wide-spread distribution of the code. One commercial backer of MOSIX withdrew from the operating system business.

The current outlook is much brighter. The latest versions of MOSIX support process migration on BSDI's version of UNIX and Linux. The Linux port eliminates the legal barriers that prevented the distribution of early versions of the system.

Sprite. Sprite as a whole did not achieve a long-lasting success, so its process migration facility suffered with it. Sprite's failure to expand significantly beyond U.C. Berkeley was due to a conscious decision among its designers not to invest the enormous effort that would have been required to support a large external community. Instead, individual ideas from Sprite, particularly in the areas of file systems and virtual memory, have found their way into commercial systems over time.

The failure of Sprite's process migration facility to similarly influence the commercial marketplace has come as a surprise. Ten years ago we would have predicted that process migration in UNIX would be commonplace today, despite the difficulties in supporting it. Instead, user-level load distribution is commonplace, but it is commonly limited to applications that can run on different hosts without ill effects, and relies either on explicit checkpointing or the ability to run to completion.

Mach and OSF/1. Compared to other systems, Mach has gone the furthest in technology transfer. Digital UNIX has been directly derived from OSF/1, NT internals resemble the Mach design, and a lot of research was impacted by Mach. However, almost no distributed support was transferred elsewhere. The distributed memory management and distributed IPC were extremely complex, requiring significant effort to develop and to maintain. The redesign of its distributed IPC was accomplished within the OSF RI [Milojičić et al., 1997], but distributed memory management has never been redesigned and was instead abandoned

[Black et al. 1998]. Consequently, task and process migration have never been transferred elsewhere except to Universities and Labs.

LSF. Platform Computing has not aggressively addressed process migration because the broad market is still not ready—partially due to an immature distributed system structure, and partially due to a lack of cooperation from OS and application vendors. But most importantly there was no significant customer demand.

Since a vast majority of users run Unix and Windows NT, for which dynamic process migration is not supported by the OS kernel, Platform Computing has been using user-level job checkpointing and migration as an indirect way to achieve process migration for the users of LSF. A checkpoint library based on that of Condor is provided that can be linked with Unix application programs to enable transparent process migration. This has been integrated into a number of important commercial applications. For example, a leading circuit simulation tool from Cadence, called Verilog, can be checkpointed on one workstation and resumed on another.

It is often advantageous to have checkpointing built into the applications and have LSF manage the migration process. The checkpoint file is usually smaller compared to user-level, because only certain data structures need to be saved, rather than all dirty memory pages. With more wide-spread use of workstations and servers on the network, Platform Computing is experiencing a rapidly increasing demand for process migration.

7.2. Misconceptions

Frequently, process migration has been dismissed as an academic exercise with little chance for wide deployment [Eager et al., 1988; Kremien and Kramer, 1992; Shivaratri et al., 1992]. Many rationales have been presented for this position, such as:

- significant complexity,
- unacceptable costs,
- the lack of support for transparency, and
- the lack of support for heterogeneity.

Some implementations, even successful ones, indeed have reinforced such beliefs. Despite the absence of wide spread deployment, work on process migration has persisted. In fact, recently we have seen more and more attempts to provide migration and other forms of mobility [Steensgaard and Jul, 1995; Roush and Campbell, 1996; Smith and Hutchinson, 1998]. Checkpoint/restart systems are being deployed for the support of long-running processes [Platform Computing, 1996]. Finally, mobile agents are being investigated on the Web.

If we analyze implementations, we see that technical solutions exist for each of these problems (complexity, cost, nontransparency and homogeneity). Migration has been supported with various degrees of complexity: as part of kernel mechanisms; as user-level mechanisms; and even as a part of an application (see Sections 4.2–4.6). The time needed to migrate has been reduced from the range of seconds or minutes [Mandelberg and Sunderam, 1988; Litzkow and Solomon, 1992] to as low as 14ms [Roush and Campbell, 1996]. Various techniques have been introduced to optimize state transfer [Theimer et al., 1985; Zayas, 1987a; Roush and Campbell, 1996] (see Section 3.2). Transparency has been achieved to different degrees, from limited to complete (see Section 3.3). Finally, recent work demonstrates improvements in supporting heterogeneous systems, as done in Emerald [Steensgaard and Jul, 1995], Tui [Smith and Hutchinson, 1998] and Legion [Grimshaw and Wulf, 1996] (see Section 3.6).

7.3. True Barriers to Migration Adoption

We believe that the true impediments to deploying migration include the following:

- **A lack of applications**. Scientific applications and academic loads (e.g. *pmake* and simulations) represent a small percentage of today's applications. The largest percentage of applications

today represent standard PC applications, such as word-processing, and desktop publishing. Such applications do not significantly benefit from migration.

- **A lack of infrastructure**. There has not been a widely-used distributed operating system. Few of the distributed features of academically successful research operating systems, such as Mach, Sprite, or the V kernel, have been transferred to the marketplace despite initial enthusiasm. This lack increases the effort needed to implement process migration.
- **Migration is not a requirement for users**. Viable alternatives, such as remote invocation and remote data access, might not perform as uniformly as process migration but they are able to meet user expectations with a simpler and well understood approach [Eager et al., 1986a, Kremien and Kramer, 1992].
- **Sociological factors** have been important in limiting the deployment of process migration. In the workstation model, each node belongs to a user. Users are not inclined to allow remote processes to visit their machines. A lot of research has addressed this problem, such as process eviction in Sprite [Douglis and Ousterhout, 1991], or lowering the priority of foreign processes in the Stealth scheduler [Krueger and Chawla, 1991]. However, the psychological effects of workstation ownership still play a role today.

7.4. How these Barriers Might be Overcome

It often takes a long time for good research ideas to become widely adopted in the commercial arena. Examples include object-orientation, multi-threading, and the Internet. It may be the case that process mobility is not ripe enough to be adopted by the commercial market.

We address each of the barriers identified in previous section and try to predict how migration might fit the future needs. The rest of the section is highly speculative because of the attempts to extrapolate market needs and technology.

Applications. To become popular in the marketplace, migration needs a "killer application" that will provide a compelling reason for commercial operating system vendors to devote the resources needed to implement and support process migration. The types of application that are well-suited for process migration include processor-intensive tasks such as parallel compilation and simulation, and I/O-intensive tasks that would benefit from the movement of a process closer to some data or another process (see also Section 2.4). These applications are exceedingly rare by comparison to the typical uses of today's computers in the home and workplace, such as word processing, spreadsheets, and games. However, applications are becoming more distributed, modular, and dependent on external data. In the near future, because of the exceeding difference in network performance, it will be more and more relevant to execute (migrate) applications close to the source of data. Modularity will make parallelization easier (e.g. various component models, such as Java Beans and Microsoft DCOM).

Infrastructure. The NT operating system is becoming a *de facto* standard, leading to a common environment. UNIX is also consolidating into fewer versions. All these systems start to address the needs for clustering, and large-scale multicomputers. Both environments are suitable for process migration. These operating systems are becoming more and more distributed. A lot of missing infrastructure is becoming part of the standard commercial operating systems or its programming environments.

Convenience vs. requirement (impact of hardware technology). The following hardware technology trends may impact process migration in the future: high speed networks, large scale systems, and the popularity of hardware mobile gadgets. With the increasing difference in network speeds (e.g. between a mobile computer and a fiber-channel), the difference between remote execution and migration becomes greater. Being able to move processes during execution (e.g. because it

was realized that there is a lot of remote communication) can improve performance significantly. Secondly, with the larger scale of systems, the failures are more frequent, thereby increasing the relevance of being able to continue program execution at another node. For long-running or critical applications (those that should not stop executing) migration becomes a more attractive solution. Finally, the increasing popularity of hardware mobile gadgets will require mobility support in software. Examples include migrating applications from a desktop, to a laptop, and eventually to a gadget (e.g. future versions of cellular phones or palmtops).

Sociology. There are a few factors related to sociology. The meaning and relevance of someone's own workstation is blurring. There are so many computers in use today that the issue of computing cycles becomes less relevant. Many computers are simply servers that do not belong to any single user, and at the same time the processing power is becoming increasingly cheap. A second aspect is that as the world becomes more and more connected, the idea of someone else's code arriving on one's workstation is not unfamiliar anymore. Many security issues remain, but they are being actively addressed by the mobile code and agents community.

In summary, we do not believe that there is a need for any revolutionary development in process migration to make it widely used. We believe that it is a matter of time, technology development, and the changing needs of users that will trigger a wider use of process migration.

8. SUMMARY AND FURTHER RESEARCH

In this paper we have surveyed process migration mechanisms. We have classified and presented an overview of a number of systems, and then discussed four case studies in more detail. Based on this material, we have summarized various migration characteristics. Throughout the text we tried to assess some misconceptions about process migration, as well as to discover the true reasons for the lack of its wide acceptance.

We believe there is a future for process migration. Different streams of development may well lead to a wider deployment of process migration. Below we include some possible paths.

One path is in the direction of LSF, a user-level facility that provides much of the functionality of full-fledged process migration systems, but with fewer headaches and complications. The checkpoint/restart model of process migration has already been relatively widely deployed. Packages such as Condor, LSF and Loadleveler are used for scientific and batch applications in production environments. Those environments have high demands on their computer resources and can take advantage of load sharing in a simple manner.

A second path concerns clusters of workstations. Recent advances in high speed networking (e.g. ATM [Partridge, 1994] and Myrinet [Boden et al., 1995]) have reduced the cost of migrating processes, allowing even costly migration implementations to be deployed.

A third path, one closer to the consumers of the vast majority of today's computers (Windows systems on Intel-based platforms), would put process migration right in the home or office. Sun recently announced their Jini architecture for home electronics [Sun Microsystems, 1998] and other similar systems are sure to follow. One can imagine a process starting on a personal computer, and migrating its flow of control into another device in the same domain. Such activity would be similar to the migratory agents approach currently being developed for the Web [Rothermel and Hohl, 1998].

Still another possible argument for process migration, or another Worm-like facility for using vast processing capability across a wide range of machines, would be any sort of national or international computational effort. Several years ago, Quisquater and Desmedt [1991] suggested that the Chinese government could solve complex problems (such as factoring large numbers) by permitting people to use the processing power in their television sets, and offering a prize for a correct answer

as an incentive to encourage television owners to participate. In case of extraordinary need, process migration could provide the underlying mechanism for large-scale computation across an ever-changing set of computers.

Finally, the most promising new opportunity is the use of mobile agents in the Web. In this setting, both technical and sociological conditions differ from the typical distributed system where process migration has been de-ployed (see the analysis in Section 7.2). Instead of the processor pool and workstation models, the Web environment connects computers as interfaces to the "network-is-computer" model. The requirements for transparency are relaxed, and user-specific solutions are preferred. Performance is dominated by network latency and therefore state transfer is not as dominant as it is on a local area network; remote access and remote invocation are not competitive with solutions based on mobility. Users are ready to allow foreign code to be downloaded to their computer if this code is executed within a safe environment. In addition, there are plenty of dedicated servers where foreign code can execute. Heterogeneity is supported at the language level. Generally speaking, the use of mobile agents in a Web environment overcomes each of the real impediments to deploying process migration, and will be a growing application of the technology (albeit with new problems, such as security, that are currently being addressed by the mobile agents community). Mobile agents bear a lot of similarity and deploy similar techniques as process migration.

Process migration will continue to attract research independently of its success in market deployment. It is deemed an interesting, hard, and unsolved problem, and as such is ideal for research. However, reducing the amount of transparency and the OS-level emphasis is common for each scenario we outlined above. Eventually this may result in a less transparent OS support for migration, reflecting the lack of transparency to the application level while still providing certain guarantees about connectivity.

ACKNOWLEDGMENTS

We would like to thank Amnon Barak, David Black, Don Bolinger, Bill Bryant, Luca Cardelli, Steve Chapin, Alan Downey, Shai Guday, Mor Harchol-Balter, Dag Johansen, and Dan Teodosiu for providing many useful suggestions that significantly improved the paper. The anonymous reviewers provided an extensive list of general, as well as very detailed, suggestions that have strengthened our focus, presentation and correctness of the paper. We are indebted to them and to the ACM *Computing Surveys* editor, Fred Schneider.

REFERENCES

ACCETTA, M., BARON, R., BOLOSKY, W., GOLUB, D., RASHID, R., TEVANIAN, A., AND YOUNG, M. 1986. Mach: A New Kernel Foundation for UNIX Development. *Proceedings of the Summer USENIX Conference*, 93–112.

AGRAWAL, R. AND EZZAT, A. 1987. Location Independent Remote Execution in NEST. *IEEE Transactions on Software Engineering 13*, 8, 905–912.

ALON, N., BARAK, A., AND MANBER, U. 1987. On Disseminating Information Reliably without Broadcasting. *Proceedings of the 7th International Conference on Distributed Computing Systems*, 74–81.

ALONSO, R. AND KYRIMIS, K. 1988. A Process Migration Implementation for a UNIX System. *Proceedings of the USENIX Winter Conference*, 365–372.

AMARAL, P., JACQEMOT, C., JENSEN, P., LEA, R., AND MIROWSKI, A. 1992. Transparent Object Migration in COOL-2. *Proceedings of the ECOOP*.

ANDERSEN, B. 1992. Load Balancing in the Fine-Grained Object-Oriented Language Ellie. *Proceedings of the Workshop on Dynamic Object Placement and Load Balancing in Parallel and Distributed Systems Programs*, 97–102.

ANDERSON, T. E., CULLER, D. E., AND PATTERSON, D. A. 1995. A Case for NOW (Networks of Workstations). *IEEE Micro 15*, 1, 54–64.

ARTSY, Y., CHANG, Y., AND FINKEL, R. 1987. Interprocess Communication in Charlotte. *IEEE Software*, 22–28.

ARTSY, Y. AND FINKEL, R. 1989. Designing a Process Migration Facility: The Charlotte Experience. *IEEE Computer*, 47–56.

BAALBERGEN, E. H. 1988. Design and Implementation of Parallel Make. *Computing Systems 1*, 135–158.

BANAWAN, S. A. AND ZAHORJAN, J. 1989. Load Sharing in Hierarchical Distributed Systems. *Proceedings of the 1989 Winter Simulation Conference*, 963–970.

BARAK, A. AND LITMAN, A. 1985. MOS: a Multicomputer Distributed Operating System. *Software-Practice and Experience 15*, 8, 725–737.

BARAK, A. AND SHILOH, A. 1985. A Distributed Load-Balancing Policy for a Multicomputer. *Software-Practice and Experience 15*, 9, 901–913.

BARAK, A. AND WHEELER, R. 1989. MOSIX: An Inte-grated Multiprocessor UNIX. *Proceedings of the Winter 1989 USENIX Conference*, 101–112.

BARAK, A., SHILOH, A., AND WHEELER, R. 1989. Flood Prevention in the MOSIX Load-Balancing Scheme. *IEEE Technical Committee on Operating Systems Newsletter 3*, 1, 24–27.

BARAK, A., GUDAY, S., AND WHEELER, R. G. 1993. The MOSIX Distributed Operating System. *Springer Verlag*.

BARAK, A., LADEN, O., AND BRAVERMAN, A. 1995. The NOW MOSIX and its Preemptive Process Migration Scheme. *Bulletin of the IEEE Technical Committee on Operating Systems and Application Environments 7*, 2, 5–11.

BARBOU DES PLACES, F. B., STEPHEN, N., AND REYNOLDS, F. D. 1996. Linux on the OSF Mach3 Microkernel. *Proceedings of the First Conference on Freely Redistributable Software*, 33–46.

BARRERA, J. 1991. A Fast Mach Network IPC Implementation. *Proceedings of the Second USENIX Mach Symposium*, 1–12.

BASKETT, F., HOWARD, J., AND MONTAGUE, T. 1977. Task Communication in DEMOS. *Proceedings of the 6th Symposium on OS Principles*, 23–31.

BAUMANN, J., HOHL, F., ROTHERMEL, K., AND STRAßER, M. 1998. Mole—Concepts of a Mobile Agent System. *World Wide Web 1*, 3, 123–137.

BEGUELIN, A., DONGARRA, J., GEIST, A., MANCHEK, R., OTTO, S., AND WALPOLE, J. 1993. PVM: Experiences, Current Status and Future Directions. *Proceedings of Supercomputing 1993*, 765–766.

BERNSTEIN, P. A. 1996. Middleware: A Model for Distributed System Services. *Communications of the ACM 39*, 2, 86–98.

BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZINSKI, M., BECKER, D., CHAMBERS, C., AND EGGERS, S. 1995. Extensibility, Safety and Performance in the SPIN Operating System. *Proceedings of the 15th Symposium on Operating Systems Principles*, 267–284.

BHARAT, K. A. AND CARDELLI, L. 1995. Migratory Applications. *Proceedings of the Eight Annual ACM Symposium on User Interface Software Technology*.

BLACK, A., HUTCHINSON, N., JUL, E., LEVY, H., AND CARTER, L. 1987. Distributed and Abstract Types in Emerald. *IEEE Transactions on Software Engineering, SE-13*, 1, 65–76.

BLACK, D., GOLUB, D., JULIN, D., RASHID, R., DRAVES, R., DEAN, R., FORIN, A., BARRERA, J., TOKUDA, H., MALAN, G., AND BOHMAN, D. 1992. Microkernel Operating System Architecture and Mach. *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 11–30.

BLACK, D., MILOJIČIĆ, D., DEAN, R., DOMINIJANNI, M., LANGERMAN, A., SEARS, S. 1998. Extended Memory Management (XMM): Lessons Learned. *Software-Practice and Experience 28*, 9, 1011–1031.

BODEN, N., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. 1995. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro 15*, 1, 29–38.

BOKHARI, S. H. 1979. Dual Processor Scheduling with Dynamic Reassignment. *IEEE Transactions on Software Engineering, SE-5*, 4, 326–334.

BOMBERGER, A. C., FRANTZ, W. S., HARDY, A. C., HARDY, N., LANDAU, C. R., AND SHAPIRO, J. S. 1992. The Key-KOS (R) Nanokernel Architecture. *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 95–112.

BOND, A. M. 1993. Adaptive Task Allocation in a Distributed Workstation Environment. *Ph.D. Thesis*, Victoria University at Wellington.

BONOMI, F. AND KUMAR, A. 1988. Adaptive Optimal Load Balancing in a Heterogeneous Multiserver System with a Central Job Scheduler. *Proceedings of the 8th International Conference on Distributed Computing Systems*, 500–508.

BORGHOFF U. M. 1991. *Catalogue of Distributed File/Operating Systems*. Springer Verlag.

BOWEN, N. S., NIKOLAOU, C. N., AND GHAFOOR, A. 1988. Hierarchical Workload Allocation for Distributed Systems. *Proceedings of the 1988 International Conference on Parallel Processing*, II:102–109.

BROOKS, C., MAZER, M. S., MEEKS, S., AND MILLER, J. 1995. Application-Specific Proxy Servers as HTTP Stream Transducers. *Proceedings of the Fourth International World Wide Web Conference*, 539–548.

BRYANT, B. 1995. Design of AD 2, a Distributed UNIX Operating System. *OSF Research Institute*.

BRYANT, R. M. AND FINKEL, R. A. 1981. A Stable Distributed Scheduling Algorithm. *Proceedings of the 2nd International Conference on Distributed Computing Systems*, 314–323.

BUGNION, E., DEVINE, S., GOVIL, K., AND ROSENBLUM, M. 1997. Disco: running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems 15*, 4, 412–447.

BUTTERFIELD, D. A. AND POPEK, G. J. 1984. Network Tasking in the Locus Distributed UNIX System. *Proceedings of the Summer USENIX Conference*, 62–71.

CABRERA, L. 1986. The Influence of Workload on Load Balancing Strategies. *Proceedings of the Winter USENIX Conference*, 446–458.

CARDELLI, L. 1995. A Language with Distributed Scope. *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*, 286–297.

CASAS, J., CLARK, D. L., CONURU, R., OTTO, S. W., PROUTY, R. M., AND WALPOLE, J. 1995. MPVM: A Migration Transparent Version of PVM. *Computing Systems 8*, 2, 171–216.

CASAVANT, T. L. AND KUHL, J. 1988a. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, SE-14(2), 141–152.

CASAVANT, T. L. AND KUHL, J. 1988b. Effects of Response and Stability on Scheduling in Distributed Computing systems. *IEEE Transactions on Software Engineering*, SE-14(11), 1578–1588.

CHAPIN, J., ROSENBLUM, M., DEVINE, S., LAHIRI, T., TEODOSIU, D., AND GUPTA, A. 1995. Hive: Fault Containment for Shared-Memory Multiprocessors. *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 12–25.

CHAPIN, S. J. 1995. Distributed Scheduling Support in the Presence of Autonomy. *Proceedings of the 4th Heterogeneous Computing Workshop, IPPS*, 22–29.

CHAPIN, S. J. 1993. Scheduling Support Mechanisms for Autonomous, Heterogeneous, Distributed Systems. *Ph.D. Thesis, Technical Report CSD-TR-93-087*, Purdue University.

CHAPIN, S. J. AND SPAFFORD, E. H. 1994. Support for Implementing Scheduling Algorithms Using MESSIAHS. *Scientific Programming*, 3, 325–340.

CHAPIN, S. J. 1996. Distributed and Multiprocessor Scheduling. *ACM Computing Surveys 28*, 1, 233–235.

CHASE, J. S., AMADOR, F. G., LAZOWSKA, E. D., LEVY, H. M., AND LITTLEFIELD, R. J. 1989. The Amber System: Parallel Programming on a Network of Multiprocessors. *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 147–158.

CHERITON, D. R. 1988. The V Distributed System. *Communications of the ACM 31*, 3, 314–333.

CHERITON, D. 1990. Binary Emulation of UNIX Using the V Kernel. *Proceedings of the Summer USENIX Conference*, 73–86.

CHESS, D., B., G., HARRISON, C., LEVINE, D., PARRIS, C., AND TSUDIK, G. 1995. Itinerant Agents for Mobile Computing. *IEEE Personal Communications Magazine*.

CHOU, T. C. K. AND ABRAHAM, J. A. 1982. Load Balancing in Distributed Systems. *IEEE Transactions on Software Engineering, SE-8*, 4, 401–419.

CHOU, T. C. K. AND ABRAHAM, J. A. 1983. Load Redistribution under Failure in Distributed Systems. *IEEE Transactions on Computers, C-32*, 9, 799–808.

CHOW, Y.-C. AND KOHLER, W. H. 1979. Models for Dynamic Load Balancing in a Heterogeneous Multiple Processor System. *IEEE Transactions on Computers, C-28*, 5, 354–361.

COHN, D. L., DELANEY, W. P., AND TRACEY, K. M. 1989. Arcade: A Platform for Distributed Operating Systems. *Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems (WEBDMS)*, 373–390.

CONCEPCION, A. I. AND ELEAZAR, W. M. 1988. A Testbed for Comparative Studies of Adaptive Load Balancing Algorithms. *Proceedings of the Distributed Simulation Conference*, 131–135.

DANNENBERG, R. B. 1982. Resource Sharing in a Network of Personal Computers. *Ph.D. Thesis, Technical Report CMU-CS-82-152*, Carnegie Mellon University.

DANNENBERG, R. B. AND HIBBARD, P. G. 1985. A Butler Process for Resource Sharing on a Spice Machine. *IEEE Transactions on Office Information Systems 3*, 3, 234–252.

DEARLE A. DI BONA R., FARROW J., HENSKENS F., LINDSTROM A., ROSENBERG J. AND VAUGHAN F. 1994. Grasshopper: An Orthogonally Persistent Operating System. *Computer Systems 7*, 3, 289–312.

DEDIU, H., CHANG, C. H., AND AZZAM, H. 1992. Heavyweight Process Migration. *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems*, 221–225.

DENNING, P. J. 1980. Working Sets Past and Present. *IEEE Transactions on Software Engineering, SE-6*, 1, 64–84.

DIKSHIT, P., TRIPATHI, S. K., AND JALOTE, P. 1989. SAHAYOG: A Test Bed for Evaluating Dynamic Load-Sharing Policies. *Software-Practice and Experience*, 19, 411–435.

DOUGLIS, F. AND OUSTERHOUT, J. 1987. Process Migration in the Sprite Operating System. *Proceedings of the Seventh International Conference on Distributed Computing Systems*, 18–25.

DOUGLIS, F. 1989. Experience with Process Migration in Sprite. *Proceedings of the USENIX Workshop on Experiences with Distributed and Multiprocessor Systems (WEBDMS)*, 59–72.

DOUGLIS, F. 1990. Transparent Process Migration in the Sprite Operating System. *Ph.D. Thesis, Technical Report UCB/CSD 90/598, CSD (EECS)*, University of California, Berkeley.

DOUGLIS, F. AND OUSTERHOUT, J. 1991. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software-Practice and Experience 21*, 8, 757–785.

DUBACH, B. 1989. Process-Originated Migration in a Heterogeneous Environment. *Proceedings of the 17th ACM Annual Computer Science Conference*, 98–102.

EAGER, D., LAZOWSKA, E., AND ZAHORJAN, J. 1986a. A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing. *Performance Evaluation 6*, 1, 53–68.

EAGER, D., LAZOWSKA, E., AND ZAHORJAN, J. 1986b. Dynamic Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering 12*, 5, 662–675.

EAGER, D., LAZOWSKA, E., AND ZAHORJAN, J. 1988. The Limited Performance Benefits of Migrating Active Processes for Load Sharing. *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer*

Systems, Performance Evaluation Review 16, 1, 63–72.

EFE, K. 1982. Heuristic Models of Task Assignment Scheduling in Distributed Systems. *IEEE Computer, 15*, 6, 50–56.

EFE, K. AND GROSELJ, B. 1989. Minimizing Control Overheads in Adaptive Load Sharing. *Proceedings of the 9th International Conference on Distributed Computing Systems*, 307–315.

ENGLER, D. R., KAASHOEK, M. F., AND O'TOOLE, J. J. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proceedings of the 15th Symposium on Operating Systems Principles*, 267–284.

ESKICIOGLU, M. R. 1990. Design Issues of Process Migration Facilities in Distributed Systems. *IEEE Technical Committee on Operating Systems Newsletter 4*, 2, 3–13.

EZZAT, A., BERGERON, D., AND POKOSKI, J. 1986. Task Allocation Heuristics for Distributed Computing Systems. *Proceedings of the 6th International Conference on Distributed Computing Systems*.

FARMER, W. M., GUTTMAN, J. D., AND SWARUP, V. 1996. Security for Mobile Agents: Issues and Requirements. *Proceedings of the National Information Systems Security Conference*, 591–597.

FEITELSON, D. G. AND RUDOLPH, L. 1990. Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control. *Proceedings of the 1990 International Conference on Parallel Processing, I*: 1–8.

FERRARI, D. AND ZHOU., S. 1986. A Load Index for Dynamic Load Balancing. *Proceedings of the 1986 Fall Joint Computer Conference*, 684–690.

FINKEL, R., SCOTT, M., ARTSY, Y., AND CHANG, H. 1989. Experience with Charlotte: Simplicity and Function in a Distributed Operating system. *IEEE Transactions on Software Engineering, SE-15*, 6, 676–685.

FLEISCH, B. D. AND POPEK, G. J. 1989. Mirage: A Coherent Distributed Shared Memory Design. *Proceedings of the 12th ACM Symposium on Operating System Principles*, 211–223.

FREEDMAN, D. 1991. Experience Building a Process Migration Subsystem for UNIX. *Proceedings of the Winter USENIX Conference*, 349–355.

GAIT, J. 1990. Scheduling and Process Migration in Partitioned Multiprocessors. *Journal of Parallel and Distributed Computing 8*, 3, 274–279.

GAO, C., LIU, J. W. S., AND RAILEY, M. 1984. Load Balancing Algorithms in Homogeneous Distributed Systems. *Proceedings of the 1984 International Conference on Parallel Processing*, 302–306.

GERRITY, G. W., GOSCINSKI, A., INDULSKA, J., TOOMEY, W., AND ZHU, W. 1991. Can We Study Design Issues of Distributed Operating Systems in a Generalized Way? *Proceedings of the Second USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 301–320.

GOLDBERG, A. AND JEFFERSON, D. 1987. Transparent Process Cloning: A Tool for Load Management of Distributed Systems. *Proceedings of the 8th International Conference on Parallel Processing*, 728–734.

GOLUB, D., DEAN, R., FORIN, A., AND RASHID, R. 1990. UNIX as an Application Program. *Proceedings of the Summer USENIX Conference*, 87–95.

GOPINATH, P. AND GUPTA, R. 1991. A Hybrid Approach to Load Balancing in Distributed Systems. *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 133–148.

GOSCINSKI, A. 1991. *Distributed Operating Systems: The Logical Design*. Addison Wesley.

GOSLING, J., JOY, B., AND STEELE, G. 1996. *The Java Language Specification*. Addison Wesley.

GRAY, R. 1995. Agent Tcl: A flexible and secure mobileagent system. *Ph.D. thesis, Technical Report TR98-327*, Department of Computer Science, Dartmouth College, June 1997.

GRIMSHAW, A. AND WULF, W. 1997. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM 40*, 1, 39–45.

GUPTA, R. AND GOPINATH, P. 1990. A Hierarchical Approach to Load Balancing in Distributed Systems. *Proceedings of the Fifth Distributed Memory Computing Conference, II*, 1000–1005.

HAC, A. 1989a. A Distributed Algorithm for Performance Improvement Through File Replication, File Migration, and Process Migration. *IEEE Transactions on Software Engineering 15*, 11, 1459–1470.

HAC, A. 1989b. Load Balancing in Distributed Systems: A Summary. *Performance Evaluation Review*, 16, 17–25.

HAERTIG, H., KOWALSKI, O. C., AND KUEHNHAUSER, W. E. 1993. The BirliX Security Architecture.

HAGMANN, R. 1986. Process Server: Sharing Processing Power in a Workstation Environment. *Proceedings of the 6th International Conference on Distributed Computing Systems*, 260–267.

HAMILTON, G. AND KOUGIOURIS, P. 1993. The Spring Nucleus: A Microkernel for Objects. *Proceedings of the 1993 Summer USENIX Conference*, 147–160.

HAN, Y. AND FINKEL, R. 1988. An Optimal Scheme for Disseminating Information. *Proceedings of the 1988 International Conference on Parallel Processing, II*, 198–203.

HARCHOL-BALTER, M. AND DOWNEY, A. 1997. Exploiting Process Lifetime Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems 15*, 3, 253–285. Previously appeared in the *Proceedings of ACM Sigmetrics 1996 Conference on Measurement and Modeling of Computer Systems*, 13–24, May 1996.

HILDEBRAND, D. 1992. An Architectural Overview of QNX. *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 113–126.

HOFMANN, M.O., MCGOVERN, A., AND WHITEBREAD, K. 1998. Mobile Agents on the Digital Battlefield.

Proceedings of the Autonomous Agents '98, 219–225.

HOHL, F. 1998. A Model of Attacks of Malicious Hosts Against Mobile Agents. *Proceedings of the 4th Workshop on Mobile Objects Systems, INRIA Technical Report*, 105–120.

HWANG, K., CROFT, W., WAH, B., BRIGGS, F., SIMONS, W., AND COATES, C. 1982. A UNIX-Based Local Computer Network with Load Balancing. *IEEE Computer*, 15, 55–66.

JACQMOT, C. 1996. Load Management in Distributed Computing Systems: Towards Adaptive Strategies. *Technical Report, Ph.D. Thesis, Departement d'Ingenierie Informatique, Universite catholique de Louvain*.

JOHANSEN, D., VAN RENESSE, R., AND SCHNEIDER, F. 1995. Operating System Support for Mobile Agents. *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, 42–45.

JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. 1988. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems 6*, 1, 109–133.

JUL, E. 1988. Object Mobility in a Distributed Object-Oriented System. *Technical Report 88-12-06, Ph.D. Thesis, Department of Computer Science*, University of Washington, Also *Technical Report no. 98/1*, University of Copenhagen DIKU.

JUL, E. 1989. Migration of Light-weight Processes in Emerald. *IEEE Technical Committee on Operating Systems Newsletter*, 3(1)(1):20–23.

KAASHOEK, M. F., VAN RENESSE, R., VAN STAVEREN, H., AND TANENBAUM, A. S. 1993. FLIP: An Internetwork Protocol for Supporting Distributed Systems. *ACM Transactions on Computer Systems*, 11(1).

KEMPER, A., KOSSMANN, D. 1995. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal* 4(3): 519-566(1995).

KHALIDI, Y. A., BERNABEU, J. M., MATENA, V., SHIRIFF, K., AND THADANI, M. 1996. Solaris MC: A Multi-Computer OS. *Proceedings of the USENIX 1996 Annual Technical Conference*, 191–204.

KLEINROCK, L. 1976. *Queueing Systems vol. 2: Computer Applications*. Wiley, New York.

KNABE, F. C. 1995. Language Support for Mobile Agents. *Technical Report CMU-CS-95-223, Ph.D. Thesis, School of Computer Science*, Carnegie Mellon University, Also Technical Report ECRC-95-36, European Computer Industry Research Centre.

KOTZ, D., GRAY, R., NOG, S., RUS, D., CHAWLA, S., AND CYBENKO., G. 1997. Agent Tcl: Targeting the needs of mobile computers. *IEEE Internet Computing 1*, 4, 58–67.

KREMIEN, O. AND KRAMER, J. 1992. Methodical Analysis of Adaptive Load Sharing Algorithms. *IEEE Transactions on Parallel and Distributed Systems 3*, 6, 747–760.

KRUEGER, P. AND LIVNY, M. 1987. The Diverse Objectives of Distributed Scheduling Policies. *Proceedings of the 7th International Conference on Distributed Computing Systems*, 242–249.

KRUEGER, P. AND LIVNY, M. 1988. A Comparison of Preemptive and Non-Preemptive Load Balancing. *Proceedings of the 8th International Conference on Distributed Computing Systems*, 123–130.

KRUEGER, P. AND CHAWLA, R. 1991. The Stealth Distributed Scheduler. *Proceedings of the 11th International Conference on Distributed Computing Systems*, 336–343.

KUNZ, T. 1991. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Transactions on Software Engineering 17*, 7, 725–730.

LAMPSON, B. 1983. Hints for Computer System Design. *Proceedings of the Ninth Symposium on Operating System Principles*, 33–48.

LANGE, D. AND OSHIMA, M. 1998. *Programming Mobile Agents in Java^{TM}-With the Java Aglet API*. Addison Wesley Longman.

LAZOWSKA, E. D., LEVY, H. M., ALMES, G. T., FISHER, M. J., FOWLER, R. J., AND VESTAL, S. C. 1981. The Architecture of the Eden System. *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, 148–159.

LEA, R., JACQUEMOT, C., AND PILLVESSE, E. 1993. COOL: System Support for Distributed Programming. *Communications of the ACM 36*, 9, 37–47.

LELAND, W. AND OTT, T. 1986. Load Balancing Heuristics and Process Behavior. *Proceedings of the SIGMETRICS Conference*, 54–69.

LIEDTKE, J. 1993. Improving IPC by Kernel Design. *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, 175–188.

LITZKOW, M 1987. Remote UNIX-Turning Idle Work-stations into Cycle Servers. *Proceedings of the Summer USENIX Conference*, 381–384.

LITZKOW, M., LIVNY, M., AND MUTKA, M. 1988. Condor A Hunter of Idle Workstations. *Proceedings of the 8th International Conference on Distributed Computing Systems*, 104–111.

LITZKOW, M. AND SOLOMON, M. 1992. Supporting Checkpointing and Process Migration outside the UNIX Kernel. *Proceedings of the USENIX Winter Conference*, 283–290.

LIVNY, M. AND MELMAN, M. 1982. Load Balancing in Homogeneous Broadcast Distributed Systems. *Proceedings of the ACM Computer Network Performance Symposium*, 47–55.

LO, V. 1984. Heuristic Algorithms for Task Assignments in Distributed Systems. *Proceedings of the 4th International Conference on Distributed Computing Systems*, 30–39.

LO, V. 1989. Process Migration for Communication Performance. *IEEE Technical Committee on Operating Systems Newsletter 3*, 1, 28–30.

LO, V. 1988. Algorithms for Task Assignment and Contraction in Distributed Computing Systems. *Proceedings of the 1988 International Conference on Parallel Processing, II*, 239–244.

LOUBOUTIN, S. 1991. An Implementation of a Process Migration Mechanism using Minix. *Proceedings of 1991 European Autumn Conference, Budapest, Hungary*, 213–224.

LU, C., CHEN, A., AND LIU, J. 1987. Protocols for Reliable Process Migration. *INFOCOM 1987, The 6th Annual Joint Conference of IEEE Computer and Communication Societies*.

LU, C. 1988. Process Migration in Distributed Systems. *Ph.D. Thesis, Technical Report*, University of Illinois at Urbana-Champaign.

LUX, W., HAERTIG, H., AND KUEHNHAUSER, W. E. 1993. Migrating Multi-Threaded, Shared Objects. *Proceedings of 26th Hawaii International Conference on Systems Sciences, II*, 642–649.

LUX, W. 1995. Adaptable Object Migration: Concept and Implementation. *Operating Systems Review 29*, 2, 54–69.

MA, P. AND LEE, E. 1982. A Task Allocation Model for Distributed Computing Systems. *IEEE Transactions on Computers, C-31*, 1, 41–47.

MAGUIRE, G. AND SMITH, J. 1988. Process Migrations: Effects on Scientific Computation. *ACM SIGPLAN Notices, 23*, 2, 102–106.

MALAN, G., RASHID, R., GOLUB, D., AND BARON, R. 1991. DOS as a Mach 3.0 Application. *Proceedings of the Second USENIX Mach Symposium*, 27–40.

MANDELBERG, K. AND SUNDERAM, V. 1988. Process Migration in UNIX Networks. *Proceedings of USENIX Winter Conference*, 357–363.

MEHRA, P. AND WAH, B. W. 1992. Physical Level Synthetic Workload Generation for Load-Balancing Experiments. *Proceedings of the First Symposium on High Performance Distributed Computing*, 208–217.

MILLER, B. AND PRESOTTO, D. 1981. XOS: an Operating System for the XTREE Architecture. *Operating Systems Review, 2*, 15, 21–32.

MILLER, B., PRESOTTO, D., AND POWELL, M. 1987. DEMOS/MP: The Development of a Distributed Operating System. *Software-Practice and Experience 17*, 4, 277–290.

MILOJIČIĆ, D. S., BREUGST, B., BUSSE, I., CAMPBELL, J., COVACI, S., FRIEDMAN, B., KOSAKA, K., LANGE, D., ONO, K., OSHIMA, M., THAM, C., VIRDHAGRISWARAN, S., AND WHITE, J. 1998b. MASIF, The OMG Mobile Agent System Interoperability Facility. *Proceedings of the Second International Workshop on Mobile Agents*, pages 50–67. Also appeared in Springer *Journal on Personal Technologies, 2*, 117–129, 1998.

MILOJIČIĆ, D. S., CHAUHAN, D., AND LAFORGE, W. 1998a. Mobile Objects and Agents (MOA), Design, Implementation and Lessons Learned. *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies (COOTS)*, 179–194. Also appeared in *IEE Proceedings—Distributed Systems Engineering, 5*, 1–14, 1998.

MILOJIČIĆ, D., DOUGLIS, F., WHEELER, R. 1999. *Mobility: Processes, Computers, and Agents*. Addison-Wesley Longman and ACM Press.

MILOJIČIĆ, D., GIESE, P., AND ZINT, W. 1993a. Experiences with Load Distribution on Top of the Mach Microkernel. *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*.

MILOJIČIĆ, D., ZINT, W., DANGEL, A., AND GIESE, P. 1993b. Task Migration on the top of the Mach Microkernel. *Proceedings of the third USENIX Mach Symposium*, 273–290.fc

MILOJIČIĆ, D. 1993c. Load Distribution, Implementation for the Mach Microkernel. *Ph.D. Thesis, Technical Report*, University of Kaiserslautern. Also *Vieweg, Wiesbaden*, 1994.

MILOJIČIĆ, D., LANGERMAN, A., BLACK, D., SEARS, S., DOMINIJANNI, M., AND DEAN, D. 1997. Concurrency, a Case Study in Remote Tasking and Distributed IPC. *IEEE Concurrency 5*, 2, 39–49.

MIRCHANDANEY, R., TOWSLEY, D., AND STANKOVIC, J. 1989. Analysis of the Effects of Delays on Load Sharing. *IEEE Transactions on Computers 38*, 11, 1513–1525.

MIRCHANDANEY, R., TOWSLEY, D., AND STANKOVIC, J. 1990. Adaptive Load Sharing in Heterogeneous Distributed Systems. *Journal of Parallel and Distributed Computing*, 331–346.

MULLENDER, S. J., VAN ROSSUM, G., TANENBAUM, A. S., VAN RENESSE, R., AND VAN STAVEREN, H. 1990. Amoeba—A Distributed Operating System for the 1990s. *IEEE Computer 23*, 5, 44–53.

MUTKA, M. AND LIVNY, M. 1987. Scheduling Remote Processing Capacity in a Workstation Processor Bank Computing System. *Proceedings of the 7th International Conference on Distributed Computing Systems*, 2–7.

NELSON, M. N., AND OUSTERHOUT, J. K. 1988. Copy-on-Write for Sprite. *Proceedings of the Summer 1988 USENIX Conference*, 187–201.

NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. 1988. Caching in the Sprite Network File System. *ACM Transaction on Computer Systems 6*, 1, 134–54.

NELSON, R. AND SQUILLANTE, M. 1995. Stochastic Analysis of Affinity Scheduling and Load Balancing in Parallel Processing Systems. *IBM Research Report RC 20145*.

NI, L. M. AND HWANG, K. 1985. Optimal Load Balancing in a Multiple Processor System with Many Job Classes. *IEEE Transactions on Software Engineering, SE-11*, 5, 491–496.

NICHOLS, D. 1987. Using Idle Workstations in a Shared Computing Environment. *Proceedings of the 11th Symposium on OS Principles*, 5–12.

NICHOLS, D. 1990. Multiprocessing in a Network of Workstations. *Ph.D. Thesis, Technical Report CMU-CS-90-107*, Carnegie Mellon University.

NUTTAL, M. 1994. Survey of Systems Providing Process or Object Migration. *Operating System Review, 28*, 4, 64–79.

OMG, 1996. Common Object Request Broker Architecture and Specification. *Object Management Group Document Number 96.03.04*.

OUSTERHOUT, J., CHERENSON, A., DOUGLIS, F., NELSON, M., AND WELCH, B. 1988. The Sprite Network Operating System. *IEEE Computer*, 23–26.

OUSTERHOUT, J. 1994. *TcL and the Tk Toolkit*. Addison-Wesley Longman.

PAINDAVEINE, Y. AND MILOJIČIĆ, D. 1996. Process v. Task Migration. *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, 636–645.

PARTRIDGE, C. 1994. *Gigabit Networking*. Addison Wesley.

PEINE, H. AND STOLPMANN, T. 1997. The Architecture of the Ara Platform for Mobile Agents. *Proceedings of the First International Workshop on Mobile Agents (MA'97)*. LNCS 1219, Springer Verlag, 50–61.

PETRI, S. AND LANGENDORFER, H. 1995. Load Balancing and Fault Tolerance in Workstation Clusters Migrating Groups of Communicating Processes. *Operating Systems Review 29* 4, 25–36.

PHELAN, J. M. AND ARENDT, J. W. 1993. An OS/2 Personality on Mach. *Proceedings of the third USENIX Mach Symposium*, 191–202.

PHILIPPE, L. 1993. Contribution à l'étude et la réalisation d'un système d'exploitation à image unique pour multicalculateur. *Ph.D. Thesis, Technical Report 308*, Université de Franche-comté.

PIKE, R., PRESOTTO, D., THOMPSON, K., AND TRICKEY, H. 1990. Plan 9 from Bell Labs. *Proceedings of the UKUUG Summer 1990 Conference*, 1–9.

PLATFORM COMPUTING. 1996. LSF User's and Administrator's Guides, Version 2.2, Platform Computing Corporation.

POPEK, G., WALKER, B. J., CHOW, J., EDWARDS, D., KLINE, C., RUDISIN, G., AND THIEL, G. 1981. Locus: a Network-Transparent, High Reliability Distributed System. *Proceedings of the 8th Symposium on Operating System Principles*, 169–177.

POPEK, G. AND WALKER, B. 1985. The Locus Distributed System Architecture. *MIT Press*.

POWELL, M. AND MILLER, B. 1983. Process Migration in DEMOS/MP. *Proceedings of the 9th Symposium on Operating Systems Principles*, 110–119.

PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUYE, J., KETHANA, L., WALPOLE, J., AND ZHANG, K. 1995. Optimistic Incremental Specialization. *Proceedings of the 15th Symposium on Operating Systems Principles*, 314–324.

QUISQUATER, J.-J. AND DESMEDT, Y. G. 1991. Chinese Lotto as an Exhaustive Code-Breaking Machine. *IEEE Computer 24*, 11, 14–22.

RANGANATHAN, M., ACHARYA, A., SHARMA, S. D., AND SALTZ, J. 1997. Networkaware Mobile Programs. *Proceedings of the USENIX 1997 Annual Technical Conference*, 91–103.

RASHID, R. AND ROBERTSON, G. 1981. Accent: a Communication Oriented Network Operating System Kernel. *Proceedings of the 8th Symposium on Operating System Principles*, 64–75.

RASHID, R. 1986. From RIG to Accent to Mach: The Evolution of a Network Operating System. *Proceedings of the ACM/IEEE Computer Society Fall Joint Computer Conference*, 1128–1137.

ROSENBERRY, W., KENNEY, D., AND FISHER, G. 1992. *Understanding DCE*. O'Reilly & Associates, Inc.

ROTHERMEL, K., AND HOHL, F. 1998. Mobile Agents. *Proceedings of the Second International Workshop, MA'98*, Springer Verlag.

ROUSH, E. T. 1995. The Freeze Free Algorithm for process Migration. *Ph.D. Thesis, Technical Report*, University of Illinois at Urbana-Champaign.

ROUSH, E. T. AND CAMPBELL, R. 1996. Fast Dynamic Process Migration. *Proceedings of the International Conference on Distributed Computing Systems*, 637–645.

ROWE, L. AND BIRMAN, K. 1982. A Local Network Based on the UNIX Operating System. *IEEE Transactions on Software Engineering, SE-8*, 2, 137–146.

ROZIER, M. 1992. Chorus (Overview of the Chorus Distributed Operating System). *USENIX Workshop on Micro Kernels and Other Kernel Architectures*, 39–70.

SCHILL, A. AND MOCK, M. 1993. DC++: Distributed Object Oriented System Support on top of OSF DCE. *Distributed Systems Engineering 1*, 2, 112–125.

SCHRIMPF, H. 1995. Migration of Processes, Files and Virtual Devices in the MDX Operating System. *Operating Systems Review 29*, 2, 70–81.

SHAMIR, E. AND UPFAL, E. 1987. A Probabilistic Approach to the Load Sharing Problem in Distributed Systems. *Journal of Parallel and Distributed Computing, 4*, 5, 521–530.

SHAPIRO, M. 1986. Structure and Encapsulation in Distributed Systems: The PROXY Principle. *Proceedings of the 6th International Conference on Distributed Computing Systems*, 198–204.

SHAPIRO, M., DICKMAN, P., AND PLAINFOSSÉ, D. 1992. Robust, Distributed References and Acyclic Garbage Collection. *Proceedings of the Symposium on Principles of Distributed Computing*, 135–146.

SHAPIRO, M., GAUTRON, P., AND MOSSERI, L. 1989. Persistence and Migration for C++ Objects. *Proceedings of the ECOOP 1989-European Conference on Object-Oriented Programming*.

SHIVARATRI, N. G. AND KRUEGER, P. 1990. Two Adaptive Location Policies for Global Scheduling Algorithms. *Proceedings of the 10th International*

Conference on Distributed Computing Systems, 502–509.

SHIVARATRI, N., KRUEGER, P., AND SINGHAL, M. 1992. Load Distributing for Locally Distributed Systems. *IEEE Computer*, 33–44.

SHOHAM, Y. 1997. An Overview of Agent-oriented Programming. in J. M. Bradshaw, editor, *Software Agents*, 271–290. MIT Press.

SHOCH, J. AND HUPP, J. 1982. The Worm Programs—Early Experience with Distributed Computing. *Communications of the ACM 25*, 3, 172–180.

SHUB, C. 1990. Native Code Process-Originated Migration in a Heterogeneous Environment. *Proceedings of the 18th ACM Annual Computer Science Conference*, 266–270.

SINGHAL, M. AND SHIVARATRI, N. G. 1994. *Advanced Concepts in Operating Systems*. McGraw-Hill.

SINHA, P., MAEKAWA, M., SHIMUZU, K., JIA, X., ASHIHARA, UTSUNOMIYA, N., PARK, AND NAKANO, H. 1991. The Galaxy Distributed Operating System. *IEEE Computer 24*, 8, 34–40.

SKORDOS, P. 1995. Parallel Simulation of Subsonic Fluid Dynamics on a Cluster of Workstations. *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing*.

SMITH, J. M. 1988. A Survey of Process Migration Mechanisms. *Operating Systems Review 22*, 3, 28–40.

SMITH, J. M. AND IOANNIDIS, J. 1989. Implementing Remote *fork()* with Checkpoint-Restart. *IEEE Technical Committee on Operating Systems Newsletter 3*, 1, 15–19.

SMITH, P. AND HUTCHINSON, N. 1998. Heterogeneous Process Migration: The Tui System. *Software-Practice and Experience 28*, 6, 611–639.

SOH, J. AND THOMAS, V. 1987. Process Migration for Load Balancing in Distributed Systems. *TENCON*, 888–892.

SQUILLANTE, M. S. AND NELSON, R. D. 1991. Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling. *Proceedings of the ACM SIGMETRICS Conference 19*, 1, 143–155.

STANKOVIC, J. A. 1984. Simulation of the three Adaptive Decentralized Controlled Job Scheduling algorithms. *Computer Networks*, 199–217.

STEENSGAARD, B. AND JUL, E. 1995. Object and Native Code Thread Mobility. *Proceedings of the 15th Symposium on Operating Systems Principles*, 68–78.

STEKETEE, C., ZHU, W., AND MOSELEY, P. 1994. Implementation of Process Migration in Amoeba. *Proceedings of the 14th International Conference on Distributed Computer Systems*, 194–203.

STONE, H. 1978. Critical Load Factors in Two-Processor Distributed Systems. *IEEE Transactions on Software Engineering, SE-4*, 3, 254–258.

STONE, H. S. AND BOKHARI, S. H. 1978. Control of Distributed Processes. *IEEE Computer 11*, 7, 97–106.

STUMM, M. 1988. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. *Proceedings of the Second Conference on Computer Workstations*, 12–22.

SUN MICROSYSTEMS. 1998. Jini™ Software Simplifies Network Computing. http://www.sun.com/980713/jini/feature.jhtml.

SVENSSON, A. 1990. History, an Intelligent Load Sharing Filter. *Proceedings of the 10th International Conference on Distributed Computing Systems*, 546–553.

SWANSON, M., STOLLER, L., CRITCHLOW, T., AND KESSLER, R. 1993. The Design of the Schizophrenic Workstation System. *Proceedings of the third USENIX Mach Symposium*, 291–306.

TANENBAUM, A. S., RENESSE, R. VAN, STAVEREN, H. VAN., SHARP, G. J., MULLENDER, S. J., JANSEN, A. J., AND VAN ROSSUM, G. 1990. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM, 33*, 12, 46–63.

TANENBAUM, A. 1992. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, New Jersey.

TARDO, J. AND VALENTE, L. 1996. Mobile Agent Security and Telescript. *Proceedings of COMPCON'96*, 52–63.

TEODOSIU, D. 2000. End-to-End Fault Containment in Scalable Shared-Memory Multiprocessors. *Ph.D. Thesis, Technical Report*, Stanford University.

THEIMER, M. H. AND HAYES, B. 1991. Heterogeneous Process Migration by Recompilation. *Proceedings of the 11th International Conference on Distributed Computer Systems*, 18–25.

THEIMER, M. AND LANTZ, K. 1988. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering, SE-15*, 11, 1444–1458.

THEIMER, M., LANTZ, K., AND CHERITON, D. 1985. Preemptable Remote Execution Facilities for the V System. *Proceedings of the 10th ACM Symposium on OS Principles*, 2–12.

TRACEY, K. M. 1991. Processor Sharing for Cooperative Multi-task Applications. *Ph.D. Thesis, Technical Report*, Department of Electrical Engineering, Notre Dame, Indiana.

TRITSCHER, S. AND BEMMERL, T. 1992. Seitenorientierte Prozessmigration als Basis fuer Dynamischen Lastausgleich. *GI/ITG Pars Mitteilungen, no 9*, 58–62.

TSCHUDIN, C. 1997. The Messenger Environment M0 - A condensed description. In *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, *Springer Verlag*, 149–156.

VAN DIJK, G. J. W. AND VAN GILS, M. J. 1992. Efficient process migration in the EMPS multiprocessor system. *Proceedings 6th International Parallel Processing Symposium*, 58–66.

VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. 1996. Horus: A Flexible Group Communication System. *Communication of the ACM 39*, 4, 76–85.

VASWANI, R. AND ZAHORJAN, J. 1991. The implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared Memory Multiprocessors. *Proceedings of the Thirteenth Symposium on Operating Systems Principles*, 26–40.

VENKATESH, R. AND DATTATREYA, G. R. 1990. Adaptive Optimal Load Balancing of Loosely Coupled Processors with Arbitrary Service Time Distributions. *Proceedings of the 1990 International Conference on Parallel Processing, I*, 22–25.

VIGNA, G. 1998. *Mobile Agents Security*, LNCS 1419, Springer Verlag.

VITEK, I., SERRANO, M., AND THANOS, D. 1997. Security and Communication in Mobile Object Systems. In *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, Springer Verlag, 177–200.

WALKER, B., POPEK, G., ENGLISH, R., KLINE, C., AND THIEL, G. 1983. The LOCUS Distributed Operating System. *Proceedings of the 9th Symposium on Operating Systems Principles 17*, 5, 49–70.

WALKER, B. J. AND MATHEWS, R. M. 1989. Process Migration in AIX's Transparent Computing Facility (TCF). *IEEE Technical Committee on Operating Systems Newsletter, 3*, 1, (1) 5–7.

WANG, Y.-T. AND MORRIS, R. J. T. 1985. Load Sharing in Distributed Systems. *IEEE Transactions on Computers, C-34*, 3, 204–217.

WANG, C.-J., KRUEGER, P., AND LIU, M. T. 1993. Intelligent Job Selection for Distributed Scheduling. *Proceedings of the 13th International Conference on Distributed Computing Systems*, 288–295.

WELCH, B. B. AND OUSTERHOUT, J. K. 1988. Pseudo-Devices: User-Level Extensions to the Sprite File System. *Proceedings of the USENIX Summer Conference*, 7–49.

WELCH, B. 1990. Naming, State Management and User-Level Extensions in the Sprite Distributed File System. *Ph.D. Thesis, Technical Report UCB/CSD 90/567*, CSD (EECS), University of California, Berkeley.

WHITE, J. 1997. Telescript Technology: An Introduction to the Language. *White Paper, General Magic, Inc., Sunnyvale, CA*. Appeared in Bradshaw, J., Software Agents, *AAAI/MIT Press*.

WHITE, J. E., HELGESON, S., AND STEEDMAN, D. A. 1997. System and Method for Distributed Computation Based upon the Movement, Execution, and Interaction of Processes in a Network. *United States Patent* no. 5603031.

WIECEK, C. A. 1992. A Model and Prototype of VMS Using the Mach 3.0 Kernel. *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, 187–204.

WONG, R., WALSH, T., AND PACIOREK, N. 1997. Concordia: An Infrastructure for Collaborating Mobile Agents. *Proceedings of the First International Workshop on Mobile Agents*, LNCS 1219, *Springer Verlag*, 86–97.

XU, J. AND HWANG, K. 1990. Heuristic Methods for Dynamic Load Balancing in a Message-Passing Supercomputer. *Proceedings of the Supercomputing'90*, 888–897.

ZAJCEW, R., ROY, P., BLACK, D., PEAK, C., GUEDES, P., KEMP, B., LOVERSO, J., LEIBENSPERGER, M., BARNETT, M., RABII, F., AND NETTERWALA, D. 1993. An OSF/1 UNIX for Massively Parallel Multicomputers. *Proceedings of the Winter USENIX Conference*, 449–468.

ZAYAS, E. 1987a. Attacking the Process Migration Bottleneck. *Proceedings of the 11th Symposium on Operating Systems Principles*, 13–24.

ZAYAS, E. 1987b. The Use of Copy-on-Reference in a Process Migration System. *Ph.D. Thesis, Technical Report CMU-CS-87-121*, Carnegie Mellon University.

ZHOU, D. 1987. A Trace-Driven Simulation Study of Dynamic Load Balancing. *Ph.D. Thesis, Technical Report UCB/CSD 87/305*, CSD (EECS), University of California, Berkeley.

ZHOU, S. AND FERRARI, D. 1987. An Experimental Study of Load Balancing Performance. *Proceedings of the 7th IEEE International Conference on Distributed Computing Systems*, 490–497.

ZHOU, S. AND FERRARI, D. 1988. A Trace-Driven Simulation Study of Dynamic Load Balancing. *IEEE Transactions on Software Engineering 14*, 9, 1327–1341.

ZHOU, S., ZHENG, X., WANG, J., AND DELISLE, P. 1994. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software-Practice and Experience*.

ZHU, W. 1992. The Development of an Environment to Study Load Balancing Algorithms, Process migration and load data collection. *Ph.D. Thesis, Technical Report*, University of New South Wales.

ZHU, W., STEKETEE, C., AND MUILWIJK, B. 1995. Load Balancing and Workstation Autonomy on Amoeba. *Australian Computer Science Communications (ACSC'95) 17*, 1, 588–597.

Received October 1996; revised December 1998 and July 1999; accepted August 1999