The American Heritage Dictionary defines an agent as “one that acts or has the power or authority to act... or represent another” or the “means by which something is done or caused; instrument.” The term derives from the present participle of the Latin verb agere: to drive, lead, act, or do.

The basic concept used in defining the objective is Process Migration. The background, terminology, algorithm, characteristics and case studies on
Process Migration is presented in this chapter. This literature survey is supported by the paper[46].

2.1 PROCESS MIGRATION

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 super computers 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[46].

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[50], including MOSIX[5],V[19], Accent[86], Sprite ,Mach[1] and OSF/1.In addition, some systems provide mechanisms that checkpoint active processes and resume their execution in essentially the same state on another machine, including Condor and Load Sharing Facility (LSF).

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. It is believed 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. 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.

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: Sprite and Mach.

![High Level View of Process Migration](image)

**Fig. 2.1:** 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.

2.1.1 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.

2.1.1.1 Terminology

A process is a key concept in operating systems[94]. 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[25] 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 Fig. 2.1. 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 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.

2.1.1.2 Target Architectures

Process Migration research started with the appearance of distributed processing among multiple processors[85]. Process Migration introduces opportunities for sharing processing power and other resources, such as memory and communication channels. It is addressed in early multiprocessor systems[12][93].

Current multiprocessor systems, especially symmetric multiprocessors, are scheduled using traditional scheduling methods. They are not used as an environment for Process Migration research.

2.1.1.3 Migration Algorithm

Although there are many different migration implementations and designs, most of them can be summarized in the following steps:

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.

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[84] 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.

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, by searching for the migrated process, or by forwarding messages across all visited nodes. 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.

### 2.1.1.4 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 has not occurred.

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[3].

- **A receiver-initiated policy** is activated on under loaded 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 under loaded 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[14].

- **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[90].

- **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[4].

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[92]. 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.

- **Stability** is defined as the ability to detect when the effects of further actions (e.g. load scheduling or paging) do not improve the
system state as defined by a user’s objective[17]. 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[23].

- **Approximate and heuristic scheduling** is necessary since optimal solutions are hard to achieve. Sub optimal solutions are reached either by approximating the search space with its subset or by using heuristics[102].

- **Hierarchical scheduling** integrates distributed and centralized scheduling. It supports distributed scheduling within a group of nodes and centralized scheduling among the groups[28].

### 2.1.1.5 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.

**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. 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.
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. Hence, it is decided to use this user-level migration for the current project.

**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.

**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[20]. So far the major contribution of Process Migration for fault resilience is through combination with check pointing, such as in Condor.

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.

**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. 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.

**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 that 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[27][70][91]. 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.

**2.1.1.6 Case Studies**

This section presents two case studies of Process Migration: Sprite and Mach.

**2.1.1.6.1 Sprite**
The Sprite Network Operating System was developed at U.C. Berkeley between 1984 and 1994. 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. 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[99]. Virtual memory was supported by paging a process’s heap and stack segments to a file on the local disk or a file server.

**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[79][80].

- *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[98]. 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[26]:
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. The centralized load information management system, 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.

**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.

### 2.1.1.6.2 Mach

Mach is a micro kernel developed at the Carnegie Mellon University[1][10], and later at the OSF Research Institute[13]. A migration mechanism on top of the Mach micro kernel was developed at the University of Kaiserslautern, from 1991 to 1993.

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. 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 ‘personality server’ emulating UNIX on top of the Mach micro kernel) 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.

**Goals.** The first goal was to provide a transparent task migration at user-level with minimal changes to the micro kernel. 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 micro kernel level, based on the three distinct parameters that characterize micro kernels: processing, VM and IPC.

**Design.** The design of task migration is affected by the underlying Mach micro kernel. 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.

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, pre copy and read-ahead. 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.
2. 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.
3. Interpose task/thread kernel ports on the source node.
4. Transfer the address space, capabilities, threads and the other task/thread state.
5. Interpose back task/thread kernel ports on the destination node.
6. 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.

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 micro kernel, which is up to a couple of thousand nodes on the Intel Paragon supercomputer.

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[73]. 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.
2.1.1.7 Why Process Migration has not caught on?

In this section, attempt is made to identify the barriers that have prevented a wider adoption of Process Migration and to explain how it may be possible to overcome them. The section starts with an analysis of each case study; followed by identifying misconceptions; identifying those barriers that are considered the true impediments to the adoption of migration; and in conclusion outlining the likelihood of overcoming these barriers.

**Misconceptions**

Frequently, Process Migration has been dismissed as an academic exercise with little chance for wide deployment [29][90][63]. 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.

**True Barriers to Migration Adoption**

It is strongly believed 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 represents 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[63].
• **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.

**2.1.1.8 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.

**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. 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 based 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.

### 2.2 SUMMARY

In this chapter, a survey on Process Migration mechanisms is done. A classification of a number of systems and then two case studies in detail are presented. Throughout the text attempt is made to assess some misconceptions about Process Migration, as well as to discover the true reasons for the lack of its wide acceptance.

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.
In summary, it cannot be believed that there is a need for any revolutionary development in Process Migration to make it widely used. It can be assumed that it is a matter of time, technology development, and the changing needs of users that will trigger a wider use of Process Migration.