- **Code Migration**

Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another (Milojicic et al., 2000). Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so.

That reason has always been performance. The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines. Load is often expressed in terms of the CPU queue length or CPU utilization, but other performance indicators are used as well.

Load distribution algorithms by which decisions are made concerning the allocation and redistribution of tasks with respect to a set of processors, play an important role in compute-intensive systems. However, in many modem distributed systems, optimizing computing capacity is less an issue than, for example, trying to minimize communication. Moreover, due to the heterogeneity of the underlying platforms and computer networks, performance improvement through code migration is often based on qualitative reasoning instead of mathematical models.

Consider, as an example, a client-server system in which the server manages a huge database. If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network. Otherwise, the network may be swamped with the transfer of data from the server to the client. In this case, code migration is based on the assumption that it generally makes sense to process data close to where those data reside.

This same reason can be used for migrating parts of the server to the client. For example, in many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations. Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network. The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.

Support for code migration can also help improve performance by exploiting parallelism,

but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance.

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed. However, if code can move between different machines, it becomes possible to dynamically configure distributed systems. For example, suppose a server implements a standardized interface to a file system. To allow remote clients to access the file system, the server makes use of a proprietary protocol. Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application. This approach requires that the software be readily available to the client at the time the client application is being developed.

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server. At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. This principle is shown in Fig. 3-17.

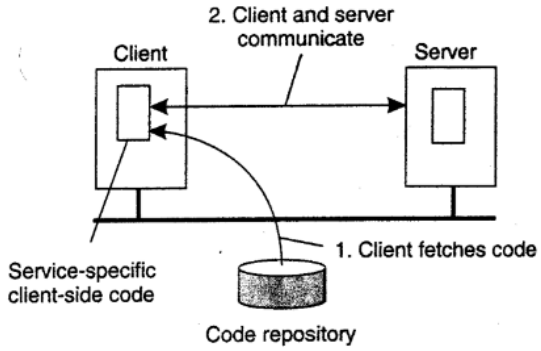

Figure 3-17. The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized. Also, it is necessary that the downloaded code can be executed on the client's machine. Different solutions are discussed below and in later chapters.

The important advantage of this model of dynamically downloading client-side software is that clients need not have all the software preinstalled to talk to servers. Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed. Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like. Changes will not affect existing client applications that rely on the server.

Models for Code Migration

Although code migration suggests that we move only code between machines, the term actually covers a much richer area. Traditionally, communication in distributed systems is concerned with exchanging data between processes. Code migration in the broadest sense deals with moving programs between machines,with the intention to have those programs be executed at the target. In some cases, as in process migration, the execution status of a program, pending signals, and other parts of the environment must be moved as well.

To get a better understanding of the different models for code migration, we use a framework described in Fuggetta et al. (1998). In this framework, a process consists of three segments. The code segment is the part that contains the set of instructions that make up the program that is being executed. The resource segment contains references to external resources needed. by the process, such as files, printers, devices, other processes, and so on. Finally, an execution segment is used to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

The bare minimum for code migration is to provide only weak mobility. In this model, it is possible to transfer only the code segment, along with perhaps some initialization data. A characteristic feature of weak mobility is that a transferred program is always started from one of several predefined starting positions.

This is what happens, for example, with Java applets, which always start execution from the beginning. The benefit of this approach is its simplicity. Weak mobility requires only

that the target machine can execute that code, which essentially boils down to making the code portable. We return to these matters when discussing migration in heterogeneous systems.

In contrast to weak mobility, in systems that support strong mobility the execution segment can be transferred as well. The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off. Clearly, strong mobility is much more general than weak mobility, but also much harder to implement.

Irrespective of whether mobility is weak or strong, a further distinction can be made between sender-initiated and receiver-initiated migration. In sender initiated migration, migration is initiated at the machine where the code currently resides or is being executed. Typically, sender-initiated migration is done when uploading programs to a computer server. Another example is sending a search program across the Internet to a Web database server to perform the queries at that server. In receiver-initiated migration, the initiative for code migration is taken by the target machine. Java applets are an example of this approach.

Receiver-initiated migration is simpler than sender-initiated migration. In many cases, code migration occurs between a client and a server, where the client takes the initiative for migration. Securely uploading code to a server, as is done in sender-initiated migration, often requires that the client has previously been registered and authenticated at that server. In other words, the server is required to know all its clients, the reason being is that the client will presumably want access to the server's resources such as its disk. Protecting such resources is essential. In contrast, downloading code as in the receiver-initiated case, can often be done anonymously. Moreover, the server is generally not interested in the client's resources. Instead, code migration to the client is done only for improving client side performance. To that end, only a limited number of resources need to be protected, such as memory and network connections.

In the case of weak mobility, it also makes a difference if the migrated code is executed by the target process, or whether a separate process is started. For example, Java applets are simply downloaded by a Web browser and are executed in the browser's address space. The benefit of this approach is that there is no need to start a separate process, thereby avoiding communication at the target machine.

The main drawback is that the target process needs to be protected against malicious or inadvertent code executions. A simple solution is to let the operating system take care of that by creating a separate process to execute the migrated code.

Note that this solution does not solve the resource-access problems mentioned above. They still have to be dealt with. Instead of moving a running process, also referred to as process migration, strong mobility can also be supported by remote cloning. In contrast to process migration, cloning yields an exact copy of the original process, but now running on a different machine. The cloned process is executed in parallel to the original process. In UNIX systems, remote cloning takes place by forking off a child process and letting that child continue on a remote machine. The benefit of cloning is that the model closely resembles the one that is already used in many applications.

The only difference is that the cloned process is executed on a different machine. In this sense, migration by cloning is a simple way to improve distribution transparency.

The various alternatives for code migration are summarized in Fig. 3-18.

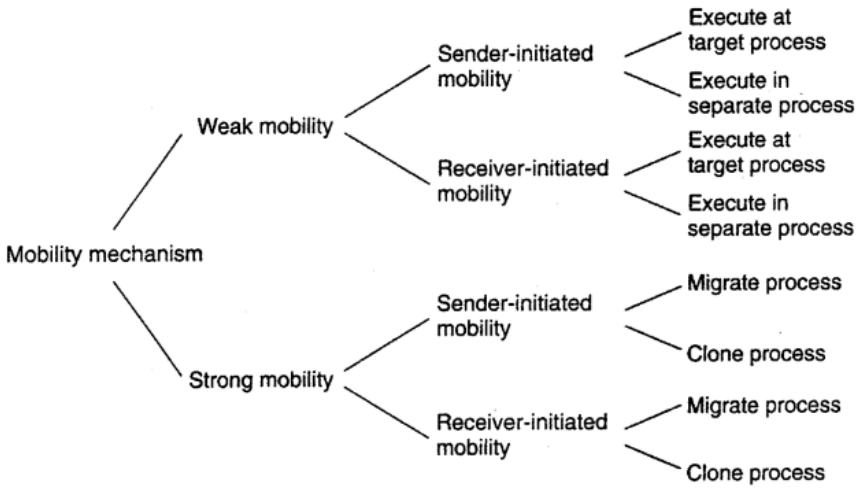

Figure 3-18. Alternatives for code migration.

Migration and Local Resources

So far, only the migration of the code and execution segment has been taken into account. The resource segment requires some special attention. What often makes code migration

so difficult is that the resource segment cannot always be simply transferred along with the other segments without being changed. For example, suppose a process holds a reference to a specific TCP port through which it was communicating with other (remote) processes. Such a reference is held in its resource segment. When the process moves to another location, it will have to give up the port and request a new one at the destination. In other cases, transferring a reference need not be a problem. For example, a reference to a file by means of an absolute URL will remain valid irrespective of the machine where

the process that holds the URL resides.

To understand the implications that code migration has on the resource segment, Fuggetta et al. (1998) distinguish three types of process-to-resource bindings. The strongest binding is when a process refers to a resource by its identifier. In that case, the process requires precisely the referenced resource, and nothing else. An example of such a binding by identifier is when a process uses a VRL to refer to a specific Web site or when it refers to an FrP server by means of that server's Internet address. In the same line of reasoning, references to local communication endpoints also lead to a binding by identifier.

A weaker form of process-to-resource binding is when only the value of a resource is needed. In that case, the execution of the process would not be affected if another resource would provide that same value. A typical example of binding by value is when a program relies on standard libraries, such as those for programming in C or Java. Such libraries should always be locally available, but their exact location in the local file system may differ between sites. Not the specific files, but their content is important for the proper execution of the process.

Finally, the weakest form of binding is when a process indicates it needs only a resource of a specific type. This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding. If, and exactly how a reference should be changed, depends on whether that resource can be moved along with the code to the target machine. More specifically, we need to consider the resource-to-machine bindings, and distinguish the following cases. Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is

to be migrated. In contrast, moving or copying a fastened resource may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites. Although such resources are, in theory, not dependent on their current machine, it is often infeasible to move them to another environment.

Finally, fixed resources are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices. Another example of a fixed resource is a local communication end point. Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code. These nine combinations are shown in Fig. 3-19.

		Resource-to-machine binding		
		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV,GR)	GR (or CP)	GR
	By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR Establish a global systemwide reference
MV Move the resource
CP Copy the value of the resource
RB Rebind process to locally-available resource

Figure 3-19. Actions to be taken with respect to the references to local resources when migrating code to another machine.

Let us first consider the possibilities when a process is bound to a resource by identifier. When the resource is unattached, it is generally best to move it along with the migrating code. However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries. An example of such a reference is a URL. When the resource is fastened or fixed, the best solution is also to create a global reference.

It is important to realize that establishing a global reference may be more than just making use of URLs, and that the use of such a reference is sometimes prohibitively expensive. Consider, for example, a program that generates high-quality images for a dedicated multimedia workstation. Fabricating high-quality images in real time is a compute-intensive task, for which reason the program may be moved to a high-performance compute server. Establishing a global reference to the multimedia workstation means setting up a communication path between the computer server and the

workstation. In addition, there is significant processing involved at both the server and the workstation to meet the bandwidth requirements of transferring the images. The net result may be that moving the program to the computer server is not such a good idea, only because the cost of the global reference is too high.

Another example of where establishing a global reference is not always that easy is when migrating a process that is making use of a local communication endpoint. In that case, we are dealing with a fixed resource to which the process is bound by the identifier. There are basically two solutions. One solution is to let the process set up a connection to the source machine after it has migrated and install a separate process at the source machine that simply forwards all incoming messages. The main drawback of this approach is that whenever the source machine malfunctions, communication with the migrated process may fail. The alternative solution is to have all processes that communicated with the migrating process, change their global reference, and send messages to the new communication endpoint at the target machine.

The situation is different when dealing with bindings by value. Consider first a fixed resource. The combination of a fixed resource and binding by value occurs, for example, when a process assumes that memory can be shared between processes. Establishing a global reference in this case would mean that we need to implement a distributed form of shared memory. In many cases, this is not really a viable or efficient solution.

Migration in Heterogeneous Systems

So far, we have tacitly assumed that the migrated code can be easily executed at the target machine. This assumption is in order when dealing with homogeneous systems. In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture. Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform. Also, we need to ensure that the execution segment can be properly represented at each platform. .

The problems coming from heterogeneity are in many respects the same as those of portability. Not surprisingly, solutions are also very similar. For example, at the end of the 1970s, a simple solution to alleviate many of the problems of porting Pascal to different machines was to generate machine-independent intermediate code for an abstract virtual machine (Barron, 1981). That machine, of course, would need to be implemented on

many platforms, but it would then allow

Pascal programs to be run anywhere. Although this simple idea was widely used for some years, it never really caught on as the general solution to portability problems for other languages, notably C. About 25 years later, code migration in heterogeneous systems is being attacked by scripting languages and highly portable languages such as Java. In essence, these solutions adopt the same approach as was done for porting Pascal. All such solutions have in common that they rely on a (process) virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java). Being in the right place at the right time is also important for language developers.

There are several reasons for wanting to migrate entire environments, but perhaps the most important one is that it allows continuation of operation while a machine needs to be shutdown. For example, in a server cluster, the systems administrator may decide to shut down or replace a machine, but will not have to stop all its running processes. Instead, it can temporarily freeze an environment, move it to another machine (where it sits next to other, existing environments), and simply unfreeze it again. Clearly, this is an extremely powerful way to manage long-running compute environments and their processes.

Let us consider one specific example of migrating virtual machines, as discussed in Clark et al. (2005). In this case, the authors concentrated on real-time migration of a virtualized operating system, typically something that would be convenient in a cluster of servers where a tight coupling is achieved through a single, shared local-area network. Under these circumstances, migration involves two major problems: migrating the entire memory image and migrating bindings to local resources.

As to the first problem, there are, in principle, three ways to handle migration (which can be combined):

1. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.

2. Stopping the current virtual machine; migrate memory, and start the new virtual machine.

3. Letting the new virtual machine pull in new pages as needed, that is, let processes start

on the new virtual machine immediately and copy memory pages on demand.

The second option may lead to unacceptable downtime if the migrating virtual machine is running a live service, that is, one that offers continuous service. On the other hand, a pure on-demand approach as represented by the third option may extensively prolong the migration period, but may also lead to poor performance because it takes a long time before the working set of the migrated processes has been moved to the new machine.