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 necessary libraries available.
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 compute 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. We return to secure code migration extensively in Chapter 9.

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.
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 Figure 3.18.

![Figure 3.18: Alternatives for code migration.](image)

### 3.5.2 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] describe 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 URL to refer to a specific Web site or when it refers to an FTP server by means of that server’s Internet address. In the same line of reasoning, references to local communication end points 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 Figure 3.19.
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 compute 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 compute 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 end point. 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, commu-
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.

Fastened resources that are referred to by their value, are typically runtime libraries. Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place. Establishing a global reference is a better alternative when huge amounts of data are to be copied, as may be the case with dictionaries and thesauruses in text processing systems.

The easiest case is when dealing with unattached resources. The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes. In the latter case, establishing a global reference is the only option.

The last case deals with bindings by type. Irrespective of the resource-to-machine binding, the obvious solution is to rebind the process to a locally available resource of the same type. Only when such a resource is not available, will we need to copy or move the original one to the new destination, or establish a global reference.

3.5.3 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