

bit less efficient because of the need to communicate through an interface, as mentioned earlier. However, it avoids the addition of baggage to the VMM, a component that is central to the operation of the system and hence that must be efficient and verifiable.

The Livewire system ([Garfinkel and Rosenblum 2003](#)) is an example of a system that separates the IDS from the VMM. The changes needed to a VMM for supporting the interface required by an external IDS are minimal and need not compromise the robustness and efficiency of the VMM itself. The IDS configures the VMM to gather information about the activity of a virtual machine. To enable this, the system specifies additional events or instructions at which the virtual machine relinquishes control to the VMM so that the required information can be gathered. The VMM is also provided with a list of checks to be made on the gathered information. These checks may be a simple matter of matching the activity against templates of known malicious activity. Thus, once set up, the VMM performs all the needed activity without communicating back and forth with the IDS module. Once the signature of a malicious event is detected, the virtual machine is suspended, and the IDS is signaled and provided with the monitoring data gathered. The IDS examines the data and either responds with a command instructing the VMM to resume operation of the virtual machine or halts the virtual machine if it cannot guarantee that the system has not been compromised.

The system just described is simple because it is conceptually stateless. The decision about whether to halt a virtual machine or to allow it to continue is based only on the information provided by the VMM when suspicious activity is detected. This may be restrictive. Certain security measures require the incorporation of state in the intrusion-detection mechanism. Often, a suspicious event may not be directly identifiable as malicious but may require further examination of specific other events in order to be identified as such. For

example, monitoring the system call activity immediately after a series of attempts to log in could give greater confidence in identifying the user logging in as an intruder. To achieve this, the IDS can send a command to the VMM immediately after a series of unsuccessful login attempts, instructing it to monitor all traps and interrupts intended for the virtual machine being protected.

Unlike the VMM, an IDS needs to have information about the type of operating system running on the virtual machine. The characteristics of intrusions often tend to be specific to the operating systems being attacked. As a result, an IDS tends to have several operating system—specific mechanisms. The Livewire system, for example, includes an OS interface library that allows the IDS to perform an analysis on a crash dump generated by a virtual machine, in almost an identical way as it would have been done by an application running on the operating system.

We mentioned earlier that the VMM is provided with a list of checks and templates to be used during the monitoring process. This is the function of the *policy engine*, the part of the IDS that determines how the monitoring should progress and whether the observed behavior matches the signature of malicious activity. The interface to the VMM, along with the mentioned OS interface library, provides a framework on which to build the policy modules of the policy engine. Livewire (see [Figure 10.4](#)) used this technique to build several policy modules, each of which was only between 30 and 130 lines of code. More details about the implementation of these for a Linux-based system are presented in [Garfinkel and Rosenblum \(2003\)](#); following are a few examples.

Lie detector module: Attackers often want to mask their activity by lying about the system. They do this by getting control of the system, modifying it, and then masking their activity by providing false responses to system queries. A VM-based intrusion-detection system can thwart such attempts by sending an enquiry to the attacked sys-

tem about the state of the system and comparing the results it provides with the actual state of the hardware of the system, which is completely known to the VMM.

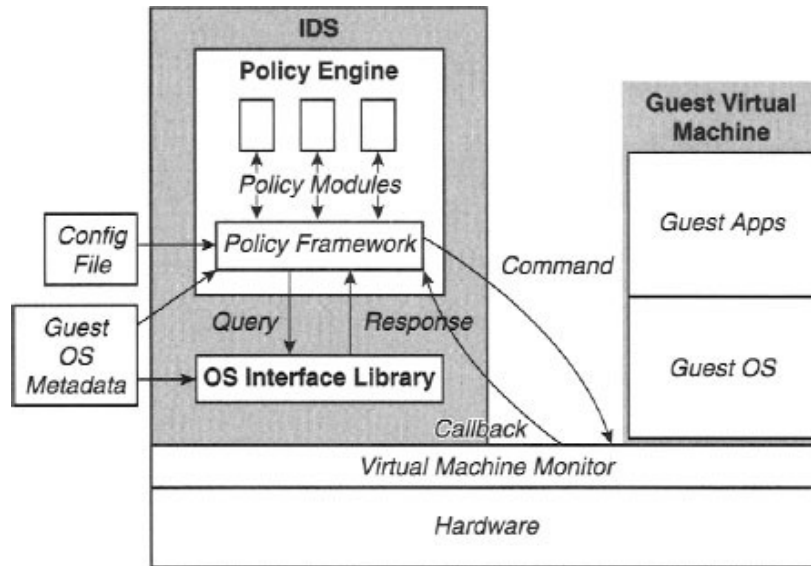


Figure 10.4 Livewire Intrusion-Detection System (Garfinkel and Rosenblum 2003). The IDS may be viewed as another virtual machine built on top of the VMM but with special privileges. The major components of the IDS are (a) the OS interface library, which provides an OS-level view of the monitored virtual machine, obtained by interpreting the metadata returned by the guest OS through the VMM, and (b) the policy engine, which provides a framework to implement common policies as well as a set of policy modules each of which implements an intrusion-detection scheme. The VMM isolates the IDS from the monitored virtual machine.

User program integrity detector module: Checking the binary stored on disk is not helpful after an attacker has modified the contents of a long-running program in memory. Livewire uses the VMM to check every code page of a program in memory and generates a signature (e.g., checksum) that is then compared with known good signatures for the same pages saved elsewhere. Any discrepancy is flagged as a potential violation of security.

Signature detector module: Antivirus programs have traditionally relied on the fact that most new viruses and Trojan horse programs simply reuse known techniques—often just copying the basic kernel code of an old virus or Trojan horse. Thus it is often effective to create a signature of these kernels and to check for this signature in files that are likely to be attacked. Virtual machine technology allows such a detection technique to be extended to the entire mem-

allows such a detection technique to be extended to the entire memory of the system. The VMM has access to the virtual hardware of the virtual machine, and hence an intrusion-detection system can use the interface of the VMM to schedule a periodic scan of the memory for such signatures.

Memory access enforcer module: Traditional operating systems protect code sections and other important parts of memory by rendering them read-only. Integrity check techniques are sometimes employed to ensure that certain critical sections of memory have not been tampered with and are still protected as read-only. But these techniques can detect such attacks only after they have occurred. By running the system inside a virtual machine, it is possible to get better and more immediate protection from such attacks, by rendering the same critical pages as read-only. Any attempt to change the access privilege of these pages is intercepted by the VMM, which can then halt the system.

The implemented policy modules generally fall into two categories: *polling modules* and *event-driven modules*. Polling modules poll the system at regular intervals to look for malicious activity, while event-driven modules report that some undesirable event has occurred on the system. The memory-access enforcer is an example of an event-driven policy module; the others are examples of polling modules.

Virtual machine technology helps eliminate some of the disadvantages that a HIDS has in comparison to the NIDS, with its better isolation characteristics. While it is natural to extend existing virtual machine systems to support intrusion detection, it is also possible that intrusion detection may one day be the reason for deployment of virtual machines.

Secure and Complete Logging Using Virtual Machines

Logging of activity on a system is an important part in the fight

Logging of activity on a system is an important part in the fight against malicious attacks on a computer system. As mentioned earlier in this chapter, a log of the activity on a system enables analysis of the events associated with an attack, especially if such data is available from the period both immediately prior to the attack and as the attack progresses. Such an analysis could lead to techniques that (a) anticipate an attack and (b) know when an attack has occurred.

A common method of logging is to record all accesses to important and critical parts of the system, e.g., login attempts, network-related events, access to the system registry, and commands invoked at the superuser or system-administrator level. Unfortunately this information is not trustworthy when under attack—an attacker gaining control of the operating system can change the logging information to mask the attack. Moreover, while evidence of the intrusion may be available from the log, there may be insufficient information to determine the exact vulnerability that led to the success of the attack.

Researchers at the University of Michigan ([Dunlap et al. 2002](#)) have proposed a virtual machine—based system called ReVirt that attempts to address both these problems, namely, the inability to guarantee the integrity of logging information as well as the incompleteness of traditional logging information. They use the VMM both to separate the logging process from the system being monitored and to collect all information needed to replay instruction-by-instruction system activity for a long period of time—potentially starting from a point before the start of the attack and continuing to well after the attack is under way.

Any computing system may be viewed as a finite-state machine. If the starting state is known, it is sufficient to know only the sequence of inputs to be able to replay its entire execution; the system moves through a set of states that can generally be determined simply by

re-executing the program. However, there are situations where the transition between the states of a system is time dependent—reproducing the effects of an earlier execution in such cases requires a simulation of the input at the time recorded during the earlier execution.

A better reference for recording nondeterministic events is the number of instructions that have been completed by the machine rather than the time on the system clock. It has been observed (**Bressoud and Schneider 1996**) that even when the inputs are completely deterministic, for example, in a program with no I/O, the time at which an instruction is executed varies from run to run, even though the count of instructions executed remains the same. This is due to a number of factors, the most obvious one being the variable latency to cache memory, depending on the interaction with other applications simultaneously running on the system. Modern processors often provide performance-monitoring hardware to count completed instructions, and hence it may be feasible to trigger events at specified instruction execution counts. However, a more efficient and an equally effective measure is the number of branches that have been executed by the machine. The location of an event can be completely characterized by the exact number of branches that have been executed from the program start, along with the value of the instruction program counter when the event occurs.

Counting events, e.g., the number of instructions executed or the number of branches executed, is accomplished through the setting of special performance counters visible at the ISA level. Each time the event occurs, the counter counts down by 1. When the counter reaches a value of zero, an interrupt can be scheduled to inform the application that the desired count has been reached. The designers of ReVirt observed that on IA-32 processors (and possibly on other processors too), the interrupt generated by the counter does not stop execution immediately and could in fact occur several tens of in-

structions later. To ensure the delivery of a nondeterministic event at a precise point in the computation, they use a two-phase technique, shown in **Figure 10.5**. At the start of the replay the precise number of branches to be executed before the occurrence of the event is known. The instruction address of the desired point within the basic block after the last branch is also known. In the first phase, the branch performance counter of the IA-32 processor is set so that 128 branches will still remain to be executed. The interrupt from the counter is received at some point where the number of branches remaining to be executed is less than 128. At this point, the second-phase strategy is invoked. In this phase, a breakpoint is set to the program counter value recorded for the event. Each time the execution arrives at the breakpoint, the number of branches is examined to determine whether the desired number of branches have been executed. The exact point is reached when the desired number of branches have been executed, and execution progresses to the instruction at the desired instruction address.

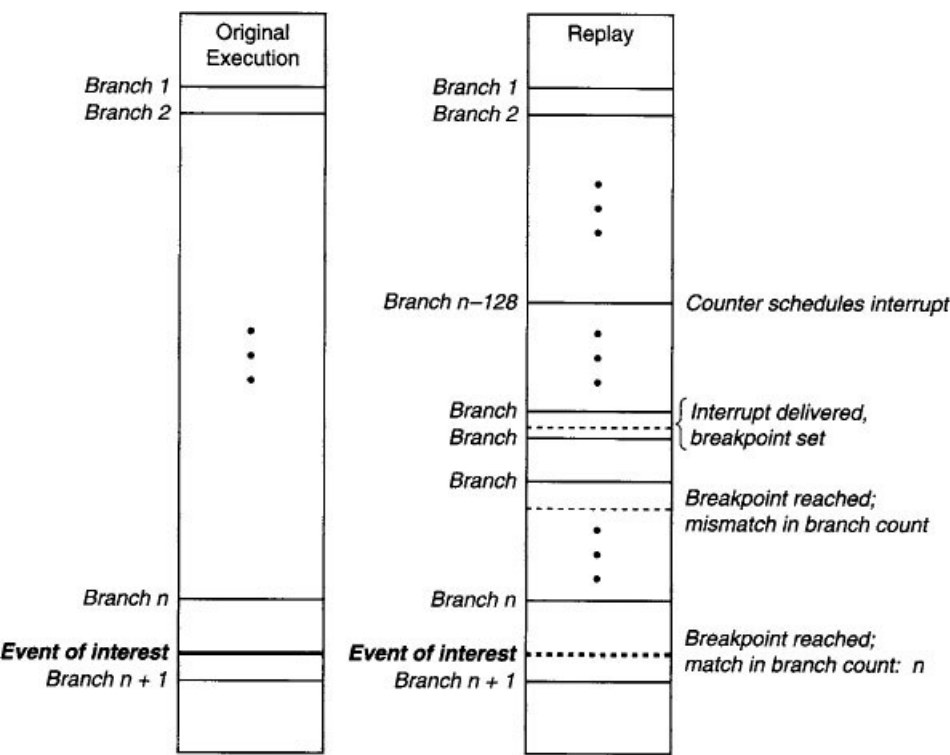


Figure 10.5 Two-Phase Method for Delivering a Nondeterministic Event Accurately During Replay.

The number of events to be recorded can grow to an unmanageable size unless one takes care to minimize the logged information. Note that only those events that can affect the execution of the monitored virtual machine need be logged; it is not necessary to log the events that arrive at the VMM but are not delivered to the virtual machine itself. Some of the common events that must be logged are timer and I/O interrupts. There are some cases where, in addition to logging the event itself, it is necessary to log the actual input value, for example, for inputs from the keyboard, mouse, network interface card, real time clock, and CD drive. Input from the hard drive need not be recorded if we ensure that the hard drive will be set to the original state at the start of the replay. Even in the case of a CD drive, the actual input values read from the media need not be recorded if the original CD is available to be replayed; in this case, even these events become deterministic.

It is important to have good knowledge of the characteristics of the specific system to ensure that all nondeterministic events are properly logged. As an example of system-specific aspects that need to be considered, the ReVirt paper ([Dunlap et al. 2002](#)) indicates the following three aspects specific to the IA-32.

1. The IA-32 architecture allows a long memory instruction (e.g., a string instruction) to be interruptible, meaning that an asynchronous interrupt could cause a context switch in the middle of execution of the instruction. By saving the value of a register, `ecx`, as part of the state, an implementation will have the necessary information to resume the instruction where it left off after handling the interrupt. This implies that the register value must also be saved when a nondeterministic event is recorded, in case the interrupted instruction happens to be a string instruction.
2. The read-timestamp-counter (`rdtsc`) instruction returns nondeterministic values, but this instruction does not trap when in user state unless the control register `CR4` is set in a particular way. One way to deal with this instruction is for the VMM to treat it the same way as other sensitive but nonprivileged instructions, as described in [Chapter 8](#), namely, to scan ahead for the instruction, insert a trap, and emulate the instruction. The ReVirt system sets the `CR4` so that the

trap, and emulate the instruction. The ReVirt system sets the CR4 so that the instruction traps, replaces the instruction with a library call routine **gettimeofday()**, which does not use the rdtsc instruction, and scales the value returned to account for the overhead.

3. A similar situation occurs with the read-performance-monitor-counter (rdpmc) instruction. This instruction also exhibits nondeterministic behavior and usually does not trap in user mode as in the rdtsc case. As with the rdtsc, the ReVirt system sets the CR4 so that the instruction traps in user mode, but then it simply disallows the instruction in the guest kernel and applications running in the monitored virtual machine.

The ReVirt system was built on a hosted virtual machine system based on Linux (UMLinux 2002). The proper functioning of the system—specifically the ability to detect the occurrence of an external interrupt event as well as to deliver it at the precise point during the computation—was tested using special kernels. The performance overhead due to logging was found to be less than 8% for a wide variety of benchmarks. The growth rate of the log was found to be between 40 and 1400MB per day. This is a reasonable number and, in practice, allows logging for several days, possibly even months.

10.1.4 Role of Dynamic Binary Rewriting Technology in Security

In [Section 4.7](#) we described a type of process virtual machine for which both the host and guest ISAs are identical but where the principal aim is the optimization of code for a particular platform. In such a system, a program is executed under the control of runtime software, which interprets the instructions in the program, determines code regions (e.g., superblocks) that are frequently executed, translates such code regions, optimizes them, and saves the optimized binary in a code cache from which the code is executed when encountered next.

The ability to control the execution of a program through the runtime also makes it possible for the runtime to sandbox the execution of a program, both to prevent the program from being attacked by

potential intruders and to prevent the program from being the launching point of an attack on the system. This is the principle behind *program shepherding* ([Kiriansky, Bruening, and Amarasinghe 2002](#)), a concept that has been implemented on the RIO dynamic optimization infrastructure ([Bruening, Duesterwald, and Amarasinghe 2001](#)), which itself is an evolution of the Dynamo binary optimization system ([Bala, Duesterwald, and Banerjia 2000](#)) described in [Chapter 4](#).

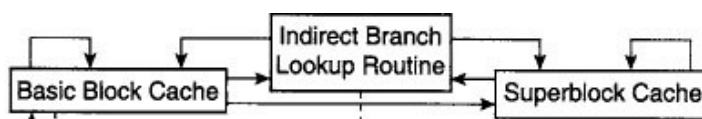
The fundamental idea behind program shepherding is that a program can be adequately protected by ensuring that every branch or jump is to a legal location and every branch into a code region originates from either another part of the same program or from some other trusted location on the system. This is not done easily at compile time because static analysis fails to resolve addresses of dynamically shared libraries as well as the targets of many indirect jumps. Thus safe execution of a program can be ensured only through runtime checks.

Restricting Control Transfers

Let us now examine the way in which control transfers can be restricted. One way to do this is to add instrumentation (i.e., instructions) at branch and jump sites to check the validity of the target. However, this could degrade performance of the program quite significantly. Moreover, as long as the program is vulnerable to attack, the attacker could change this instrumentation to circumvent the checks. A more effective way of achieving this is through techniques used in dynamic binary optimization. A section of code is interpreted the first time it is executed. The targets of the conditional and indirect branches are recorded in a safe location. When the same sequence of basic blocks has been executed a predetermined number of times, this sequence of blocks is gathered into a superblock, reoptimized, and placed in a code cache, as described in [Chapter 3](#). The runtime software of such a system performs checks for the validity

of branch targets both during the interpretation phase as well as during execution from the code cache. If the superblock represents the typical path of execution, these checks will be performed much less frequently when executing out of the code cache. In order to ensure that the code being executed is not maliciously modified, the code cache itself should be protected by making it writable only when the runtime is in control.

An overview of the RIO system infrastructure is shown in **Figure 10.6**. The specific characteristic of the system, namely, that the RIO code and the application code both work as part of the same process and share an address space, is an inheritance from the Dynamo system, which was described in **Section 4.7.2**. Even though they are part of the same process, there are two modes in which the machine operates, the *RIO mode*, in which the infrastructure code operates, and the *application mode*, in which the application work is performed. The system begins by scanning instructions and collecting basic blocks, which are inserted into a *basic block cache*, a region of memory protected from being overwritten. At a branch or jump, a map (hash) table lookup is performed to check whether the next basic block has already been cached. If it has been cached and if a direct address is given, the two basic blocks in the cache are linked through a new direct branch so that the table lookup is avoided the next time. Further performance improvement is obtained by stitching together frequently executed sequences of basic blocks to form a superblock, which is then inserted into the code cache (shown as the *superblock cache* in **Figure 10.6**). The collection of basic blocks and superblocks is done in RIO mode, but the execution of the code itself is done in application mode. A mode switch into the RIO mode occurs when a control transfer lookup fails to find an entry either in the basic block cache or in the code cache.



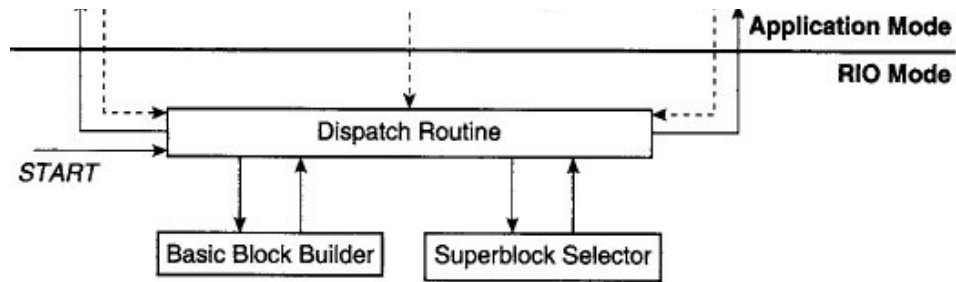


Figure 10.6 Flowchart of the RIO System (Kiriansky, Bruening, and Amarasinghe 2002). Dotted arrows depict performance-critical functions, with the operation switching between application mode and RIO mode.

Management of control flow transfers is achieved as follows. For each direct branch from one cached basic block to another, security checks are performed at the time the basic blocks are linked. A link is added only if the security policy allows a direct transfer; otherwise control returns to a runtime routine that checks for and handles all potential security violations. Once the link is inserted, it is part of the basic block cache (or superblock code cache), which itself is protected from being tampered with. As long as the program is restricted to such linked branches, there is no overhead in execution. In fact there could even be a performance benefit due to more efficient layout of the code.

Indirect-jump destinations have to be looked up in the map table to determine the mapping of destination addresses to corresponding locations in the basic block cache or the code cache. Only those cache addresses that are validated are placed in the map table; these account for most of the addresses to which jumps are made. Additional checks can be recorded in map table entries. For example, the target of a return from a subroutine can be checked to make sure the program returns to a point immediately after the original call. Such a check would thwart the type of attack that was described in [Section 10.1](#), where the program returns to an inappropriate address because the stack has been clobbered.

In general, the only branch or jump targets that lead to any significant program degradation are indirect jumps targeted outside the

code segment being protected. The RIO system checks for the validity of such targets by performing dynamic checks whenever they are encountered.

Restricting Code Execution

The origin of a piece of code must be authenticated before it is allowed to execute. This authentication must be performed not only at the time the code is loaded from the disk or from the network, but also while it is executing—the code could be changed in memory either by another program on the system or even by itself. Any changes to the code in memory must either conform to the security rules or be disallowed.

Most of the checks in the RIO system of this kind are performed at the time the system copies a sequence of instructions into the basic block cache. The main challenge here is to ensure that the code was not tampered with after being fetched into memory and before being copied into the basic block cache. Modern program binary file formats separate the code sections from the data sections and allow loaders to write-protect code pages as they are brought into memory. Any attempt to change the code results in an exception. However, the system also needs to handle the case of dynamically generated code, permitting the execution of such code when it is legal to do so and preventing it when it is not.

There are two cases of importance. The first is the case of new sections of code that are created and then loaded for execution. In this case, the region of code is usually converted from being writable to being write-protected. The RIO system authenticates the code at this point and allows the conversion only if the request is from trusted code. Similarly, if the program makes a direct jump into writable space, the system verifies the origin of the jump and write-protects the target area before allowing execution to proceed.

The second case occurs when code and data are shared on the same page—this case poses an interesting problem. Since data areas should not be write-protected, it is necessary to have two copies of such pages, one of which is write-protected and is the source of instructions, while the other serves as the target for data reads and writes and is not write-protected. This solution is adequate for most common situations. It does not handle the situation where the code itself is changed by the program, however—the case of self-modifying code. Such writes into the code area must be intercepted by the runtime monitor in order to ensure they are being performed legally. It is necessary for the system to monitor all writes into write-protected regions and to authenticate the source of such writes. If it is a legal self-modifying code situation, the runtime monitor makes the page writable, allows the write to proceed, and then write-protects the code page again, after ensuring that the basic block cache and the code cache are purged of all the affected regions. This is clearly a case that could significantly degrade the performance of the system. Fortunately, it is not a common occurrence in modern programs.

TABLE 10.1 Page Access Privileges in the RIO System

Page Type	RIO Mode	Application Mode
Application Code	R	R
Application Data	RW	RW
RIO Code Cache	RW	R (E)
RIO Code	R (E)	R
RIO Data	RW	R

R — read only, RW — read/write, R (E) — executable

Protecting the Runtime Monitor

The RIO system designers protected the RIO system code itself from attack by using different page protections during the RIO mode of operation and the application mode. The RIO mode has slightly higher page access privileges than the application mode, even

though both modes operate within the same address space. An emulated “context switch” occurs when the machine switches modes. **Table 10.1** indicates the access privileges to the various types of pages in the RIO mode and in the application mode. For instance, the code cache itself has read and write privileges in the RIO mode, whereas it has only read and execute privileges in the application mode. Tables used by the RIO system, for example, the map tables, have read-only privilege in the application mode whereas they have read and write privileges in the RIO mode.

10.1.5 Secure Systems of the Future

Security is an aspect of computing that has not been systematically addressed in the past, and is beginning to gain tremendous importance. Users and organizations are already beginning to think of ways in which they can partition their platforms so that they are able to run open, general-purpose, widely available systems and applications that are insecure alongside other applications and data that have to be protected and kept highly secure. The Terra system (**Garfinkel et al. 2003**) attempts to achieve this by creating a *trusted virtual machine monitor* that supports virtual machines incorporating the semantics of dedicated, tamper-resistant hardware on the same hardware platform as virtual machines that run normal, open, general-purpose applications.

We have seen in this section the application of various techniques described in earlier chapters of this book to handle the security problem. The process of thinking through the various issues and of coming up with solutions to handle them has motivated and will continue to motivate the search for fundamental ways of avoiding thorny security issues. Thus the world is moving toward new paradigms where security, perhaps even at the expense of some performance, is a fundamental aspect in the design of an environment and its applications. The HLL VMs, as described in **Chapters 5** and **6**, ex-

emply this approach. For some time to come, however, the vast legacy of existing environments will continue to call for security solutions of the type discussed in this section.

10.2 Migration of Computing Environments

It is not uncommon for individuals to maintain programs and to work on multiple computers at different locations, for example, one at home and one at work. Usually, there is an overlap in the activity conducted in the user's different work locations, and it is clear that the user's data is likely to be needed at multiple locations. To facilitate this multiple work-site environment, it is likely that the same operating system and the same set of applications are installed on different computers. Yet there are often subtle differences between the environments of the various installations that prove frustrating to the user. For example, one location may have a slightly different version of an application program from another, perhaps because an upgrade was applied to the machine at one location and not the other. One might naturally ask whether there is some way in which the user can see an identical environment everywhere he or she works.

Many users get around this problem by working exclusively on portable devices such as laptop computers. While this ensures that the user always sees and works with exactly the same environment, it does have two significant disadvantages. First, the user is required to carry a physical device to transport the environment, and second, this means that the user is responsible for the physical security of the portable device at all times.

In the 1960s, mainframe servers solved this problem (at least within the environs of a large office building) by providing users with simple, "dumb" terminals as the only interface to the system. The oper-

pic, dumb terminals as the only interface to the system. The operating system, applications, and data of the user resided in a central location and were accessible from the terminal through a network of serial links. The system virtual machine environment described in **Chapter 8** extended this concept by allowing each user to perceive the computer being accessed as his or hers exclusively. The virtual machine concept provides flexibility, by allowing each user to customize his or her environment, and security, by isolating a particular user's environment from those of other users of the machine.

The last few decades have seen a dramatic decrease both in the cost and in the size of computing devices. The dumb terminal of the mainframes has now become a sophisticated computer in its own right. Thus, while the concept of the mainframe server still survives, most users today access both mainframes and the Internet through a device capable of performing significantly sophisticated computation. Such processing may include applications such as graphics rendering and game simulation, Web browsing, and the editing of text documents. It is the environment associated with this interface that needs to be replicated when a user moves from place to place.

Replicating such an environment from one computer to another is not trivial, because this implies that a tremendous amount of state information must be transferred. The state of the machine includes not only the state of the resources used by the operating system and the applications running on the machine, but also the code and data belonging to the operating system and applications. Thus, a *capsule* is formed to capture the state of a running machine and hence information about the processes currently active on the system. The notion of a capsule allows a user environment targeted for migration to be treated as an object that can be manipulated.

Even assuming that the actual hardware on multiple sites is identical, replicating the environment requires all this state information

to be encapsulated, transmitted across a network, and installed on the other computer before the user can take over operation at a second computer. These operations have to be done efficiently in order for the switch between environments to appear seamless to the user. The following subsections will be devoted to ways to make such environment mobility practical and efficient.

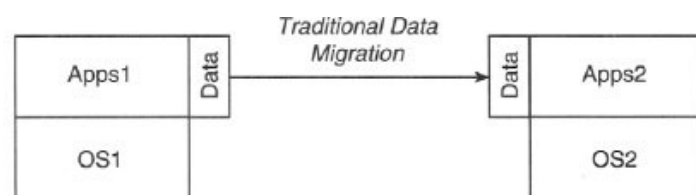
10.2.1 Virtual Computers

As we mentioned in the introduction, virtual machine technology provides the ability to capture the entire state of a computer system and thus facilitates migration of a full computing environment.

Figure 10.7 shows what is involved in such a migration.

Traditionally, in order to migrate from one computer to another, one would close all programs on one computer, shut it down, and either transport the hard drive or copy all information contained on the hard drive of the first computer on to the hard drive of the second.

Often even this seemingly simple scenario runs into complications because of subtle differences in the configurations of the two machines. Virtual machine technology inherently is better at smoothing out such subtle differences. The process of environment encapsulation is simplified if the environment is running on a virtual machine rather than directly on real hardware. The virtual machine monitor must implement data structures that separate the resources belonging to the different virtual machines running on the system. Thus, the virtual machine itself can serve as a capsule that can be migrated to another system that implements the same virtual machine system, as shown in **Figure 10.7b**.



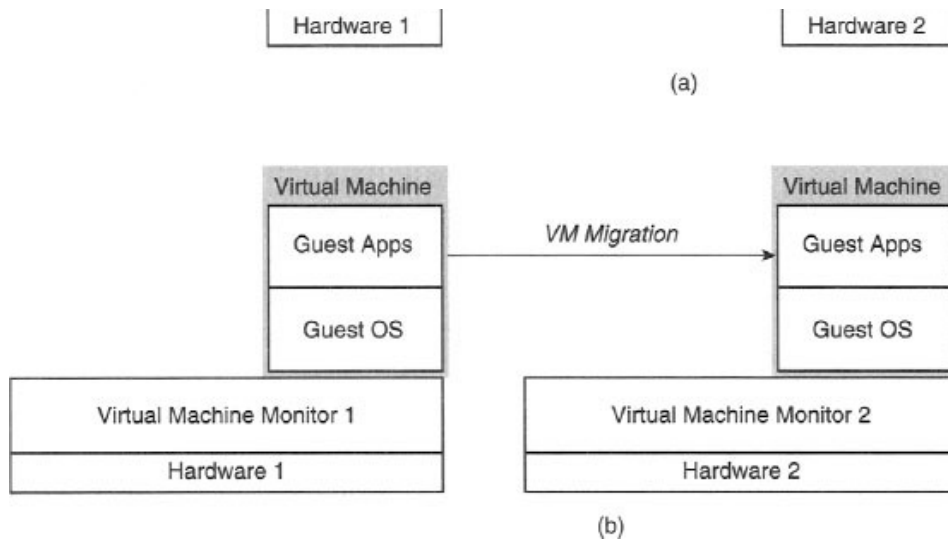


Figure 10.7 Environment Migration Using Virtual Machines. (a) Traditionally, migration involves the movement of data between machines, each with its own hardware, operating system, and complement of applications. (b) A faithful reproduction of the user environment is possible from one machine to another through the use of virtual machine technology. The state of the virtual machine, including the state of the application, and the state of the operating system are packaged and migrated. The data for the application may also be migrated if it is not located on a file system accessible at the destination.

The information about a virtual machine that is typically kept for use by the virtual machine monitor is restricted to the configuration of the virtual machine and pointers to tables that link virtual resources such as memory to real resources on the system. A capsule, on the other hand, does not contain any information about the mapping of the virtual resources. A capsule must also capture the state contained in all resources, including the disks and memory. It is conceivable that the information contained in a capsule can be many gigabytes in size and could take several hours to be transmitted on networks commonly used for connecting home computers to the outside world, for example.

Encapsulation of a running machine is similar to checkpointing. It allows the machine to be suspended for an indefinitely long time and resumed exactly from the point where the execution was originally suspended. The capsule is a host-independent checkpoint that can be transported from one machine to another; execution can be suspended on one computer and resumed at exactly the same point on another computer. If the virtual machine architecture is identical

to some real machine architecture, the execution can be resumed on a real implementation rather than on another virtual machine system.

It must be remembered that the system virtual machines running on a centralized mainframe such as the IBM z/VM also allow a user to see the same environment when traveling from one location to another. However, in that case, both the data and the processor reside in a remote location and stay there, even during execution. The amount of state information that needs to be transferred to a remote terminal is negligible, and the overhead of updating the environment to make it cognizant of the new terminal is small. The harder problem is the one we have outlined here—that of migrating the environment to work on different data and compute facilities.

There are several issues involved in developing a practical solution for this problem, and we list some of them here.

- The entire state of a computer (which could include tens of gigabytes or more of disk space) can be so large that the time taken to migrate the state may be prohibitive. Note that it is generally not necessary to send the entire state from one computer to another initially. It is usually sufficient to transmit a small part of the state at first, transmitting additional parts of the state as needed. It may also be possible, as we shall see later, to take advantage of information that may already be available on the second computer, e.g., portions of the operating system itself.
- Having decided what portion of the machine state needs to be migrated, there is still the problem of packaging and securely transmitting the information. Compression and encryption techniques have to be employed to reduce the amount of data transmitted and to transmit it securely.
- When the hardware on the two machines is identical and when the virtual machine monitors running on them are identical, the process of transmitting the environment from one machine to another should be seamless. It may be the case that while the processors are identical in the two machines, the memory or I/O configuration on the two machines is different. For example, one machine may be connected to the external network using an Ethernet adapter, the other with a wireless connection. We have already seen in **Section 8.4** how such situations are handled, both in native system virtual machines and in hosted virtual

machines.

- Finally, the ISA of the user's virtual machine and the ISA of each of the host computers determine the extent to which the performance of the system appears identical on the different sites. When the ISA of the guest is different from the ISA of the host, the virtual machine monitor is required to translate from one ISA to the other. We have already discussed various binary translation and optimization techniques that help in doing this.

In the next two subsections we examine the basic functioning of two systems that have implemented the environment-migration concept.

10.2.2 Using a Distributed File System: The Internet Suspend/Resume Scheme

Kozuch and Satyanarayanan (2002) proposed the use of virtual machines in migrating the state of a user environment from one computer to another via a technique they called *Internet Suspend/Resume* (ISR). The name attempts to capture the notion that the system essentially allows a user to suspend operation on one machine, travel to another machine, and resume execution on the other machine by migrating the state of the first machine over the Internet (**Figure 10.8**). The authors use the VMware GSX Server described in **Chapter 8** to carry out their experiments. The ISR proposal encapsulates the state of the virtual machine, including the contents of the virtual memory and the disk; the virtual machine monitor keeps this state in a file on a distributed file system accessible also from the destination computer.

One of the longstanding problems with the migration of state from one location to another is the need to change the IP address to access the network and all references to this address in the saved environment. MobileIP technology (**Perkins 1998**) solves this problem for mobile users of the Internet. The authors of the ISR scheme suggest that migration can be made completely transparent as long as the guest operating system is configured to use only such mobile technology.

The simplest approach for resuming the environment on a new machine is for this machine to access the state saved in the distributed file system and to load it as the state of a new virtual machine. If the system configuration and the virtual machine monitor on the destination machine are the same as those of the originating machine, the loading of the file may simply involve the loading of tables and data structures in the virtual machine monitor of the destination. In order to accommodate the variety of distributed file system definitions, Kozuch and Satyanarayanan suggest that the state file be written to an abstract file system interface, with machine-dependent aspects encoded in a small block that resides with the file.

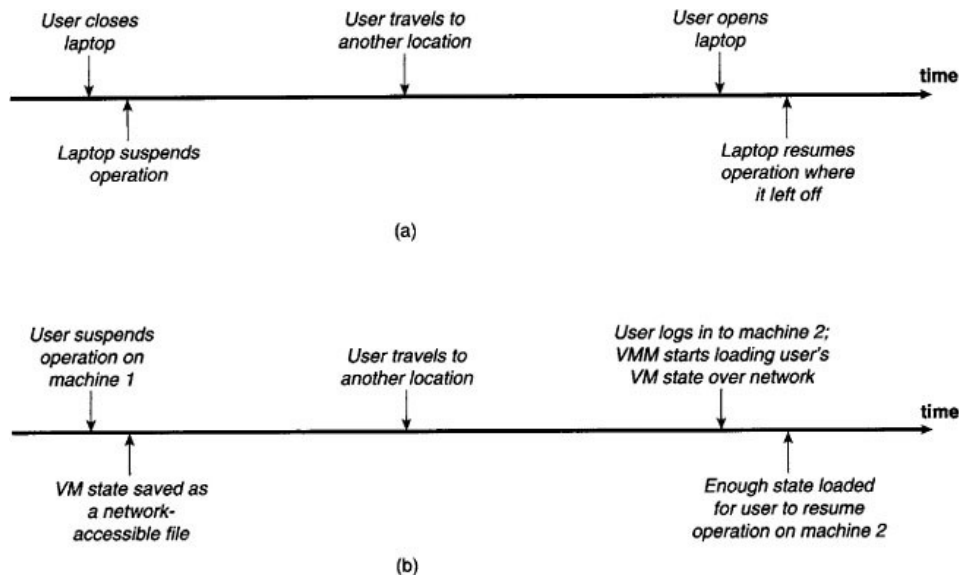


Figure 10.8 Internet Suspend/Resume. (a) What happens on a laptop when the user travels from one place to another. (b) How a similar effect is obtained using VM migration. The state of the user's environment is saved on a distributed file system accessible to the machine in the user's destination location.

Loading the entire state on the destination machine could take a considerably long time. In order to make the migration experience more responsive to the user, the authors propose the organization of the state information in modules that are loaded incrementally, with the environment resuming on the new virtual machine as soon as the needed elements have been loaded on the destination system, rather than waiting until the entire state has been loaded.

The ISR scheme uses a “pull” model, with the destination computer reading in the environment state file on demand, for example, when the user logs in to the destination machine. This could result in a long latency between the time the user logs in and the time when a sufficient amount of state is available on the destination to allow it to resume the environment. The authors suggest that there may be usage patterns that could be learned by a system in order to hide the latency of the resumption by proactively copying in the state to the local machine even before the user requests it. For example, the system could learn the pattern of travel between work and home for a particular user and bring in a part of the state of the system at the destination before the expected time at which the user arrives at the destination.

The amount of information that is part of the state of a virtual machine is considerably large. Copying the state, compressing it, transmitting it, uncompressing it, and then loading it back on another machine will be time consuming as well as bandwidth consuming. Techniques are needed to limit the amount of information that has to be transmitted in a burst, without compromising the ability to transmit the full state information from one host to another. For example, when a user frequently travels between two sites, the suspension of the environment at a site need not completely eliminate the environment at the site. It may be possible to keep information in some structured way so that when the environment is resumed at the site at a later time, only the changes to the environment need be loaded. Another possibility is to reuse portions of the environment from the environment of other users already on the system. Candidates for such reuse include files associated with the operating system, only a few of which are user specific.

The eventual adoption of such a scheme will depend on the security of information guaranteed by the migration system. Users would not want any information in their virtual machine to be accessible

by other users or even by the VMM. Kozuch and Satyanarayanan propose that the state saved in the distributed file system be maintained in an encrypted form. They also suggest that users be validated before being allowed use of a host machine.

10.2.3 State Encapsulation in the Stanford Collective

The Stanford Collective project ([Sapuntzakis et al. 2002](#)) takes a conceptually similar approach to environment migration. The principal difference from the Internet Suspend/Resume approach is that the capsule that holds the complete state of the user's virtual machine is sent to the destination computer to which the user is headed rather than to a distributed file system accessible by the destination. One advantage of this "push" model is that by appropriately planning the migration activity, the user can have instantaneous access to his or her environment at the destination site.

The virtual machine of the Collective system is an Intel IA-32 platform running on the VMware GSX server (VMware), the basic operation of which was described in [Chapter 8](#). As noted in that chapter, the GSX server is a hosted virtual machine system, with the host operating system being either Linux or Microsoft Windows. On such a virtual machine system, the VMM can take full advantage of the device drivers written for the wide range and types of I/O adapters and devices that already exist for the host operating system.

The Stanford Collective developers noted that there still remain challenges in supporting certain I/O devices, e.g., network adapters, because of the portability requirement. If a virtual machine is using a virtual network card to communicate over the Internet, the encapsulation will include an IP address that may not work on a physical computer different from the physical computer producing the capsule. The Collective system enhances the virtual machine monitor to

enable tunneling of network packets travelling to and from the capsule's old network over a virtual private network (VPN).

The Collective system developers performed extensive tuning of their system to be able to suspend a virtual machine and transmit it to a destination machine in about 20 minutes (a period they considered a typical commute time) using standard 384 kbps DSL lines. The system attempts to be oblivious of the details of the guest operating system. Hence many of the techniques they employ are applicable more generally than the ISR scheme described in the last subsection. Some of these are listed next.

Reducing Memory State Before Migration

Most physical machines today contain hundreds of megabytes of memory, and it is reasonable to expect virtual machines to have main memories that are similarly large. At any given instant, however, the part of main memory that is critical for good performance is relatively small. There is typically a large part of main memory that could be paged out to disk without noticeable difference in the system's response time. Unfortunately it is not possible to determine which pages do or do not belong to the current working set at suspend time without intruding into the guest operating system. The Collective system works around this by starting a *balloon* program on the virtual machine that requests a large number of pages. This program causes pages to be released from the other active processes on the system by the guest operating system—the expectation is that these pages are released from the currently inactive processes that do not need to respond instantaneously on a resume at the destination. The size of the memory state is now reduced by zeroing out the pages requested by the balloon program, thus enabling better compaction of the capsule.

There are several subtle policy issues associated with ballooning. While transmission of the state of the memory is more effective

when more of the pages are recovered through the balloon program, the response of the system during the resume operation becomes more sluggish unless the right number and type of pages belonging to the active processes remain mapped in memory. Pages holding cached data or those that are part of dirty buffers are ideally left untouched. On the other hand, active pages that are easily compressible need not be released to the balloon program; there is less benefit in their being zeroed out for compression and possibly greater overhead in their being restored during the resume process.

Note that the balloon program is a process running within the environment of the virtual machine, though it is triggered by the VMM when it receives a suspend request. The fact that this program runs on the guest operating system allows the program to be specialized for the characteristics of the operating system. Operating systems such as Microsoft Windows limit the size of the maximum working set for each process—the balloon program cannot recover more pages than the specified maximum number of pages that may be used by the balloon program itself. On the other hand, Linux does not set a limit to the size of the working set for a process—the Collective system monitors the swap space, while the balloon program keeps allocating and zeroing out new pages, until the free swap space reduces by a preset maximum.

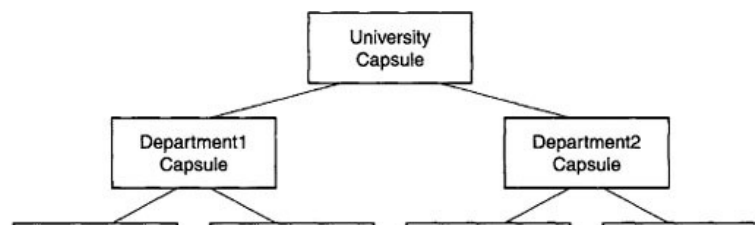
Reducing the Size of the Transmitted Packet

The balloon-program approach just mentioned goes a long way toward reducing the bandwidth required to transmit the information associated with a capsule. However, with very large disks becoming common, the state represented by the disk is rather formidable. Moreover, disks already hold many types of data, such as pictures and video, as compressed files, and additional compression is seldom effective. On the other hand, typical user migration involves just a few platforms, and the required disk image is usually not a big change from the image that was left on a platform when last visited.

The Collective exploits this observation by maintaining differences between capsules rather than the full disk image for each capsule, thus saving a considerable amount of time and resources.

Figure 10.9, taken from **Sapuntzakis et al. (2002)**, shows a sample capsule hierarchy, where each node represents a capsule state. The state of the disk in a node is saved in its entirety only in the capsule corresponding to the root node. Each node in this path inherits a base disk image corresponding to its parent node. Only the incremental difference between the states of the parent and the child is stored in each child node. The state of the disk at any node can be derived from the disk information contained in the nodes along the path from the root down to that node.

Each incremental disk uses the *copy-on-write* concept. The idea here is that whenever a copy of a disk is needed, the entire disk contents are not copied. Rather, the image of the disk is represented by a set of pointers to unique copies of various segments of the disk. Associated with each segment is a reference counter that counts the number of pointers to that segment. When a change needs to be made and a segment must be written into, the reference counter is checked. If the count is 1, then the original copy of the segment can be written into. If not, the segment is copied, the pointer is made to point to the new copy, and its reference counter is decreased by 1. This new copy can now be modified without affecting the earlier versions. Copy-on-write allows many different versions to share a single copy of the actual data; this is particularly effective when these versions are largely the same and differ in only some small aspects.



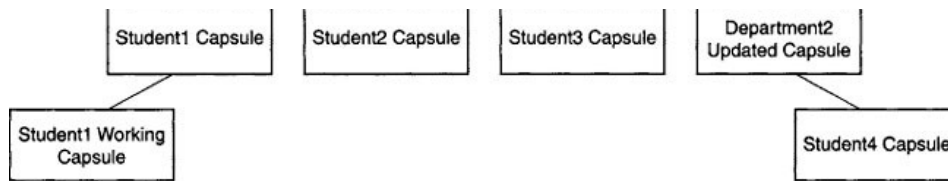


Figure 10.9 Example Capsule Hierarchy for a University. The root capsule contains all the software that will be needed by all students. Each department customizes the capsule to suit its requirements. The department administrator does this by deriving a child capsule from the parent capsule. As mentioned in the text, only changes to the university capsule are saved in each department capsule. Students derive capsules from the department capsule, perhaps having one capsule per course in a department. Alternatively, like Student1 depicted earlier, students may derive their own private capsule. The leaf node, e.g. Student1 Working Capsule, is what migrates when the student moves from place to place.

Before changes are made to the disk image, all nodes in the path from the root to the leaf corresponding to the image have to be transferred. Modifications can be made only to leaf nodes in a hierarchy. For example, in **Figure 10.9**, if Student4 wishes to make a change to his or her disk on a location that does not already have any of the information corresponding to nodes in the path from the root up to the leaf node, this information is brought over. A new child node is created with Student4 as the parent, and changes are made strictly to the incremental disk at this node. When the student moves to another computer, the currently open incremental disk image is transferred to the new computer. The images corresponding to the other nodes are essentially read-only images and need not be deleted from the computer from which execution was suspended. As students move from computer to computer, various nodes in the hierarchy remain behind, increasing the probability of finding a copy of a desired node and decreasing the incremental traffic that will be incurred when the user moves from place to place.

Reducing Start-Up Time on a Resume

On a resume operation, a lot of time can be saved if the information in the capsule disks is fetched on demand rather than being prefetched completely, because the working set typically needed by a user in a session is small. On the Collective system this is implemented as follows. All accesses to disks are intercepted by the VMM

and forwarded to a disk server. The disk server translates the request to an access in one of the nodes in the hierarchy. If it is a write, the access must be to the leaf node, as mentioned earlier. An access to information residing in one of the intermediate nodes may need a remote access to the node that owns the information. On such an access, local shadow copies of the desired blocks are made from the remote incremental disk image. Thus the desired blocks of the remote disks are incrementally built up on demand on the local system. Note that disk blocks in a node can be shared by all capsules derived from this node. In **Figure 10.9**, for example, if some blocks of the departmental copy are brought over on demand, they need not be brought over again when another student attempts to access the same blocks.

Reducing Transmission Time and Bandwidth by Exploiting Redundancy in Disk Blocks

It is not uncommon to find disk blocks that are identical, because the nature of activity on most systems tends to promote similarity among data blocks. Here are some sample situations when a needed block is already available on the system.

- A user moving back and forth between two systems leaves behind a trail of disk blocks that can potentially be reused the next time the site is visited.
- Blocks of program files are often already resident in memory. Most programs do not modify their own code, and hence it is generally possible to copy these disk blocks directly from memory.
- The memory of a machine often contains disk blocks other than program files, for example, blocks that are part of a disk cache.
- Often different users (and hence different capsules) utilize the same programs or data files. If a copy of a block already present in the capsule of a different user could be employed, it eliminates the need to transmit the copy from a remote site.

The Collective makes use of the *hashed copy* scheme. Each block on disk is associated with a hash value that aims to identify uniquely the contents of the block. The hash scheme employed in the

Collective is a strong cryptographic hash, SHA-1, which has a very low probability of a collision (**NIST 2002**). Instead of transferring the actual data blocks between computers, the Collective transmits the computed hash values of the data blocks. When a disk block is needed by a computer, it checks to see whether a block with the same hash value is available locally. It then uses this block directly or generates a copy of the block if the block is to be written into (copy-on-write). This avoids the need to transmit the block from a remote location. If the block is not available locally, the computer broadcasts a request for a copy. Other providers on the system receive this request and check against a table of hash values of already-available blocks and satisfy the request. The authors claim that the scheme is reliable because the probability of a collision, i.e., of having the same hash value for different data, is less than the probability that a requestor of a disk block receives a bad block because of an error in a TCP connection or an error in memory. The scheme is also claimed to be secure because SHA-1 makes it practically impossible to manufacture data that produce a given hash value.

Results of experiments (see **Figure 10.10**) using the Collective system (**Sapuntzakis et al. 2002**) indicate that these optimizations significantly reduce the traffic and hence the time for migrating a user environment from one machine to another on a typical DSL network. The authors simulated the migration of a user between home and work machines using snapshots based on the Business Winstone 2001 benchmark. The various techniques described earlier appeared to reduce the amount of data transferred to typically around 50MB, which can then be reduced further to less than 22MB through compression using the gzip program. Hashing reduces the transferred amounts even more. Needless to say, there were situations, for example, at the beginning of program execution sessions, where the raw amount of data needed to be transferred was as high as 500MB. However, even in these cases, compression brought down the size by a factor of 2 or more, with hashing providing further re-

ductions. The authors conclude that the experiments indicate that in typical situations of commuting between home and work, the entire environment could be transmitted during the commute. This makes it possible for a user to suspend execution at one site and resume execution at the other without noticing any change in the computing environment.

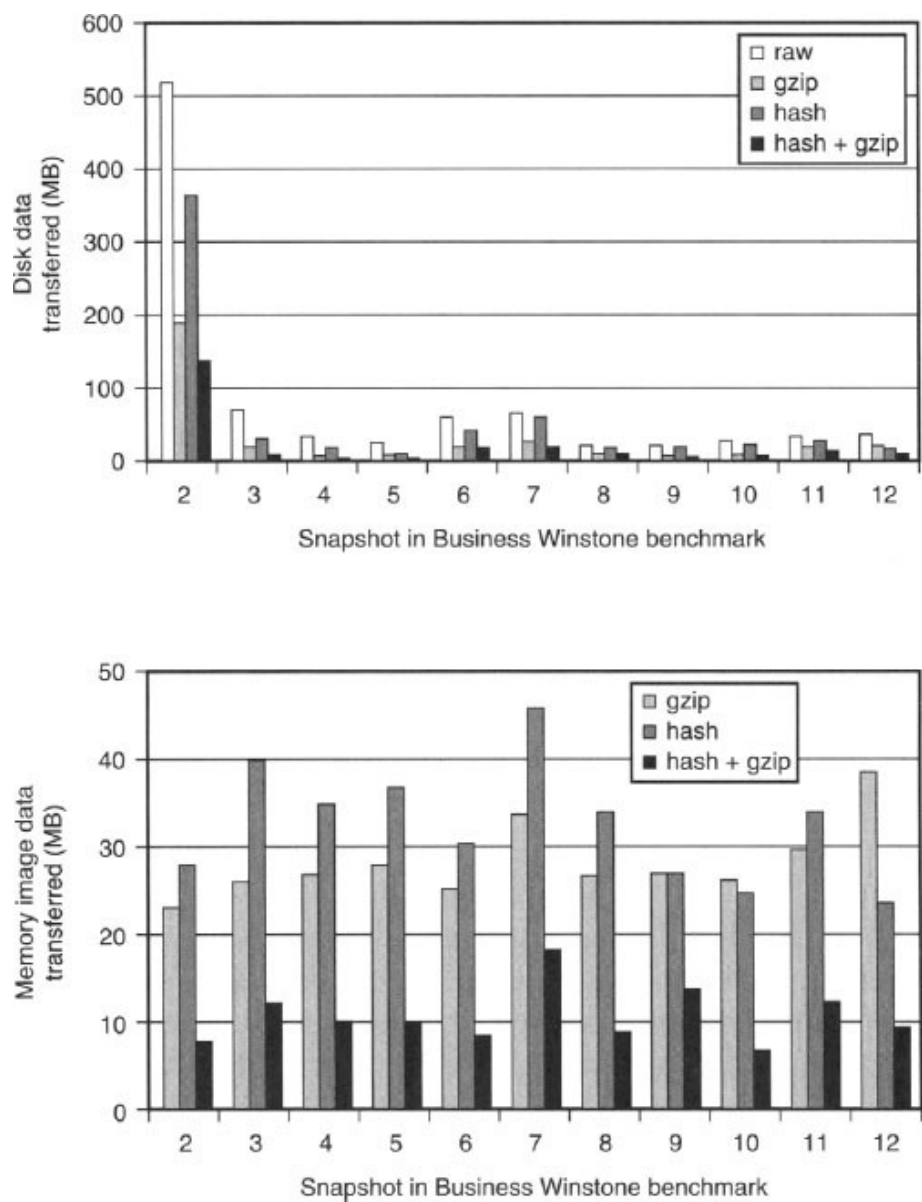


Figure 10.10 Performance of Optimization Techniques Used in the Collective (Sapuntzakis et al. 2002). The raw memory size in the experiments is 256MB. Snapshots were taken while running the Winstone benchmark every minute after the first three minutes. The migration of each snapshot was simulated assuming the destination had all information up to the previous snapshot. The graphs show the effectiveness of the hashing scheme combined with compression using gzip in reducing the traffic from the originating host to the destination host.

10.2.4 Migration of Virtual Machines in VMotion

The Internet Suspend/Resume scheme and the Collective scheme are research projects, and it may be a while before all the issues involving the migration of virtual machines over the Internet are sorted out. Meanwhile, there are aspects of such migration that already exist in VMotion, developed by VMware.

VMotion is part of the VirtualCenter (VirtualCenter) infrastructure management software that manages a cluster of Intel IA-32 virtual machine systems connected in a local-area network, as shown in **Figure 10.11**. The management functions of VirtualCenter include the deployment and monitoring of virtual machines running on a native virtual machine system called the ESX server. Under certain circumstances VirtualCenter may need to migrate a running virtual machine from one host in the cluster to another. Examples of such circumstances include:

- **Load balancing**, to improve the response time of the system through better utilization of resources
- **Security**, to quarantine a virtual machine that has been attacked
- **Collocation**, to bring communicating virtual machines closer together
- **Fault tolerance**, to move a failing host to another processor
- **Power management**, to move the load away from an overheated processor
- **Maintenance**, to move the load away from some processor while it is upgraded

The migration of a virtual machine from one node to another in VirtualCenter is done using VMotion. This migration involves copying the state of the virtual machine, including the processor state, the memory state, and the state of other virtual hardware resources, such as BIOS, devices, MAC address for Ethernet cards, and chip set states. The issues here are quite similar to the issues we discussed in connection with the ISR and Collective schemes. However, VMotion, being a product, has to grow cautiously—it is currently restricted in

its capabilities in several respects.

- The source and destination computers must be in the same server cluster managed by the same VirtualCenter manager.
- The file systems on the source and destination computers must be identical and located on shared disks in a storage-area network (SAN). This avoids the complexities of migrating disks encountered by the Collective.
- The processors running on the two computers must have the same architecture and be provided by the same vendor. This avoids potential incompatibilities that could prevent the state saved on one machine from being directly loaded into the other machine.

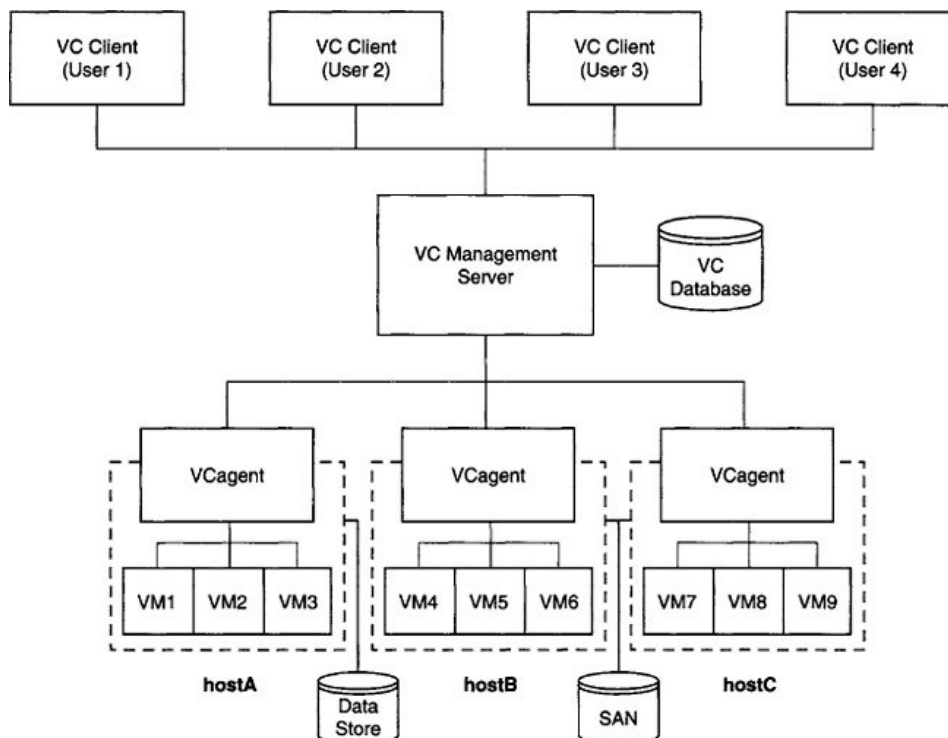


Figure 10.11 Overview of the VMware VirtualCenter. VC stands for VirtualCenter. A cluster of three nodes is shown, with the nodes named hostB and hostC sharing a file system on a storage-area network (SAN).

- The virtual machine supports a Gigabit Ethernet adapter.
- The virtual machines must not be running multiprocessor cluster applications; they should be running only stand-alone applications.

When a request is received to migrate a virtual machine using VMotion, software in the VirtualCenter performs the following actions.

1. It first ensures that the virtual machine is in a stable state on the current host.
2. It then copies the contents of the memory belonging to the virtual machine to the destination host. Data associated with the guest operating system as well as the data of applications running on the virtual machine are copied. This is referred to as the *baseline copy*. It is not the final copy because the virtual machine on the original host continues to run during this process.
3. The virtual machine on the original host is suspended. VirtualCenter then copies the last changes to memory along with the rest of the state of the virtual machine to the destination host. The information sent to the destination host at this stage is a capsule containing changes to the information sent earlier, similar to the incremental capsule as described in the Collective.
4. It then activates the virtual machine on the new host.

It is almost certain that more migration technology that has been developed in various research projects will continue to find its way into systems such as the VirtualCenter. What we observe is that there are many forms in which the problem of migration of virtual environments presents itself. We have reported here the early work in this area. As research progresses and as vendors get more experience dealing with products that allow migration, many new techniques will be developed. The eventual popularity and success of this paradigm may, in fact, mark the coming of age of virtual machine technology, particularly of system virtual machines.

10.3 Grids: Virtual Organizations

In the past couple of decades, the world has seen a rapid increase in the total amount of available computation power. Continual advances in lithographic techniques have allowed silicon feature sizes to shrink to incredibly small dimensions and hence have enabled the incorporation of billions of transistors on a fingernail-size die. This has been accompanied by the development of microprocessors, systems, and applications available at low cost to the common person and by the availability of easy-to-use programs that exploit the Internet. The economics of the computer industry are such that it is more profitable for companies to provide only a few types of micro-

processors and a few types of systems rather than to spend the enormous amount of money needed to develop a wide variety of microprocessors, to manufacture them at different locations, and to develop different operating systems for each type of system. The result is the wide availability of fairly high-powered general-purpose computing engines that sit on the desks of millions of people around the world or are being carried by millions of people virtually everywhere they go.

Yet the nature of tasks that most users perform on their computers is such that only a tiny fraction of the available compute power is actually used. Whether it is browsing the Internet to shop for an appliance or editing a term paper for a history class, the typical user does not exploit anywhere close to the full capabilities of the computer he or she is working on. The immediate impact of this is an environmental one—there is some energy consumed by a system as long as it is turned on—even when it is not performing any useful computation. This impact is being addressed in many ways, especially through the use of automatic power-management techniques. However, as computation and storage keep getting cheaper and as these computers get faster, there is more energy consumed by each of them. Thus, despite the incorporation of sophisticated power-management techniques, there is a sharp increase worldwide in the energy used by computers.

Ironically, there are occasions when users would like even more computation power than what they have available on their own systems. Scientists working with real-time data often do not have the processing capability to manipulate the large volumes of data they gather. They may need such processing capability either simply for visualization or for actually making decisions based on the data in real time. Their ability to perform such computation may be restricted due to the lack of compute capability, but it could also be due to the sophisticated software needed for the computation. Such users may not be willing to spend the large sums of money needed

to buy systems that give them the desired response time or to buy the relatively expensive software, because of the infrequent use they make of such systems or software. This scenario has an analog in other areas, such as transportation, where a user generally owns a vehicle that is adequate for most but not all of his or her needs; for the occasional extraordinary need, the user rents an appropriate vehicle, borrows it from an acquaintance, or takes advantage of a service provider such as a public transportation company.

Applying the vehicle rental model to the computer world seems a plausible thing to do. In fact, in the early days of personal computers, computer rentals were not uncommon. However, as the complexity of configuring software increased and as the cost of owning hardware decreased, short-term rental of computers fell out of favor.

With the emergence of ubiquitous connectivity and the Internet, rental is no longer the right model for making more efficient use of compute power. Instead, the public utility is beginning to look like a more appropriate model. Rather than own or lease the physical resources needed to carry out an occasional task, the emerging model for the future is for the user to seek these resources over a communication network and to have the task performed remotely. The computation world is, as suggested in **Foster and Kesselman (1998)**, in a state analogous to the electricity world in 1910, when local generators were the norm. Over time the electrical system evolved toward power grids, where suppliers of electricity pooled their resources for use by a much larger number of consumers. The design of the power utility system allows each user as much or as little power as he or she may need at any given time, with charges based on the amount consumed. The application of this model to the computing world would allow a user as much computation power as he or she may need to get an acceptable response time for a task, without having to own all the required resources.

The utility model for computation is already being driven by the needs of the high-end scientific computation community. The problems in understanding fundamental forces in nature, in the development of new drugs for common diseases, and in the prediction of weather all have one aspect in common—they rely on the ability to conduct large simulations involving massive amounts of computation. The physics community has long exploited large-scale collaboration across the globe (“big science”) as an alternative to dividing up the available funding among smaller, less effective projects. This community has been quietly spearheading the adoption of this collaborative model for computing, by championing the utility model under the name of *grid computing*. Just as “big science” converted individual projects and laboratories into one large laboratory aimed at solving one fundamental problem after another, grid computing is bringing together many small and big computers in a heterogeneous “grid,” both to make available to others unused computational resources on one person’s computer, as well as to allow a bigger group of users to share a variety of specialized, often expensive, computing resources, such as supercomputers, in the manner of a utility.

About a decade ago, the World Wide Web was developed by the scientific computing community as a means for more effective collaboration among scientists to share both data and experimental results. Later, the same model proved to be useful for the commercial world and for the layperson. It is possible that models of grid computing originating with the scientific community will similarly prove to be functional to a much wider set of users in harnessing the enormous amounts of computation power already available and that will become available in the future.

Grid computing is a form of virtualization. It behaves like a system virtual machine by automatically directing a user application to a machine that has available resources and that matches the require-

ments of the application. Taking the system virtual machine concept to a higher level, grid computing creates a *virtual organization* by allowing a systematic sharing of resources between disparate systems that may be geographically distributed (**Figure 10.12**). Grid computing thus extends the virtual machine concept from instruction-level dynamic optimizers through process-level and system virtual machines to what we will call *organization-level virtual machines*.

Unlike the other types of virtual machines discussed in this book, the real machine that is virtualized does not have a well-defined, formal architecture. Rather, its architecture is embodied in the procedures and protocols that govern the relationships between the collaborators. Many of these procedures and protocols are often nothing more than a verbal understanding between human beings. The creation of the virtual organization using a federation of computer systems will hopefully provide the impetus to understand and document the interfaces that define these relationships in a formal manner.

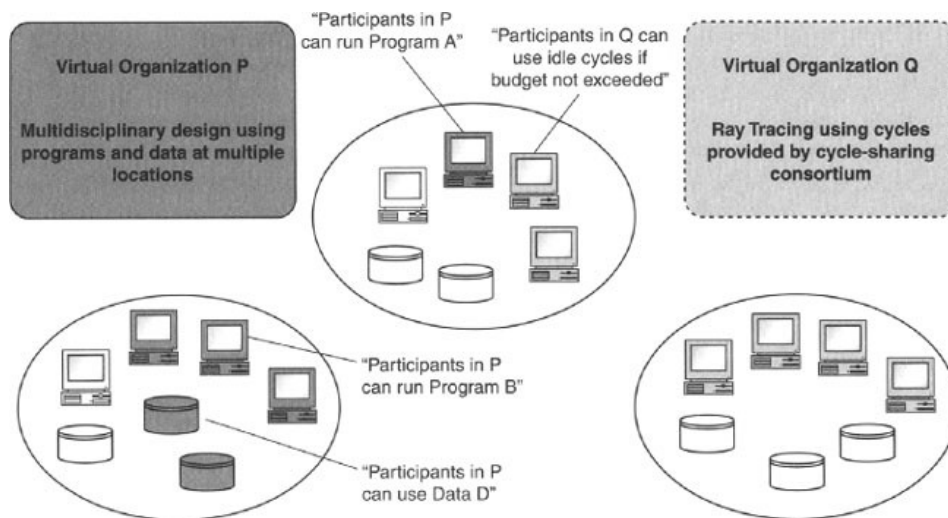


Figure 10.12 A Grid as a Set of Organizations, Both Real and Virtual (**Foster, Kesselman, and Tuecke 2001**). Pictured are three real organizations (ovals) and two virtual organizations. Virtual Organization P is a collaborative project with distributed resources, as shown by the dark grey shading. Virtual Organization Q is a project that exploits idle cycles in resources on the grid (shown in light grey shading) to perform ray tracing. Sample policies that govern access to resources are shown in quotes.

The development of such interfaces will facilitate the deployment of new services and allow the grid concept to be exploited by a

broader community. Thus, rather than the virtual organization being an attempt to faithfully reproduce the functioning of a real organization, the virtual organization enables the creation of function that did not exist before. The enablement of new function is a property of all virtual machines, as we have seen throughout this book.

10.3.1 Characteristics of an Ideal Grid

In this section we attempt to describe the salient characteristics of a grid. These characteristics are a combination of what exists on grids today and a wish list of features necessary for grid computing eventually to become a utility. The discussion here is an adaptation of the discussion of the computational grid described in [Foster and Kesselman \(1998\)](#) and in [Foster, Kesselman, and Tuecke \(2001\)](#). In the former, the authors describe a computational grid as a software and hardware infrastructure that provides dependable, consistent, inexpensive, and pervasive access to high-end computational capabilities. In their subsequent paper, Foster and his colleagues extended this definition, suggesting that a grid is an infrastructure that enables flexible, secure, and coordinated resource sharing among dynamic collections of individuals, institutions, and resources. These latter characteristics led them to refer to the grid as a *virtual organization*. We will now examine each of the characteristics of a grid in more detail.

Infrastructure

A computing grid is fundamentally a hardware and software infrastructure that enables sharing of resources. Hardware infrastructure components include computational resources, storage resources, sensors, and instruments. Software infrastructure components include programs to monitor use of resources, programs to schedule resources to requestors, and programs to turn resources on and off.

A computational resource on a grid may be a workstation, a main-frame, a supercomputer, or virtually any other type of computer. A shared storage resource may be a part of a disk on a personal workstation, a portion of a file system, or an entire network-attached storage. One important characteristic of a grid is that the systems comprising a grid need not be homogeneous—there can be a variety of processors and a variety of operating systems running on the grid.

Dependability of Service

A desirable characteristic of a grid that follows from its utility nature is that it be dependable. A true grid utility must be designed in such a way that all users of the grid either have equal access to its resources or can be guaranteed a quality of service specified in a contract with the service provider. This is sharply in contrast with most existing grids, which usually comprise a loose collection of resources provided by willing individuals or collaborating groups with a loose understanding of what is provided and what is expected. A grid participant who has volunteered the idle cycles on a workstation may not have any obligation today to provide the cycles that are pledged. As the concept of a grid evolves to that of a service, users will start to depend on the services of the grid for their day-to-day activities, and such informal understanding between participants will inevitably have to be replaced by more formal specifications and contracts. There are already efforts of this nature under way, especially with the establishment of the World Wide Grid Forum and the work being carried out in the Open Grid Services Architecture (OGSA).

Dependability of the grid system also implies fault tolerance—when a computational node in a system fails, the system must be able to recover from the failure and reallocate the resources in a manner transparent to the end user. Dependability also implies guarantee of security—users of the system must be guaranteed that their assets

security – users of the system must be guaranteed that their assets are not accessible to others and that others cannot maliciously prevent their access to resources.

Note that dependability and guarantee of quality of service are important in other forms of virtualization we have already seen. In the system virtual machines of [Chapter 8](#), many VMs access a single complex of hardware, and it is important for the virtual machine monitor to allocate resources in a fair way among the various virtual machines. Similarly, the different forms of partitioning we encountered in [Chapter 9](#) must ensure that resource allocation among the partitions is done both in a fair way and in a way that maximizes the performance of the system.

Consistency of Service

The behavior of the service provided by a grid must be predictable and consistent. A grid may contain several types of machines and systems, each of which may be able to satisfy the requirements of a particular task. While it would be impossible to ensure that the response time of a task is exactly the same each time it is executed, users will expect consistent results each time a task is executed. They may also expect some reasonable bound on the response time.

Pervasive Access

An important aspect of any utility is that it be accessible from virtually anywhere. Most current grids, as well as those that are being proposed, use the Internet as their communication medium. Thus, as long as the Internet is accessible from some part of the world, a user in that part of the world should have access to all resources made available by the grid.

Inexpensive Access

A utility service becomes popular only when the cost of a task to the

end user is attractive as compared to the alternative of performing the task on resources owned by the user. Like a utility, a grid should satisfy the needs of both the large organization wishing to perform data-mining tasks on a massive database as well as the individual who would like to have a video rendered.

Coordinated Resource Sharing

All the attributes of a grid mentioned so far view the grid as a utility. A grid is envisioned to be more than just a utility, however. Users of a utility may not have much in common with each other—they form a community only in the sense that they happen to be using the same service. In its role as a virtual organization, however, the grid is expected to allow diverse groups to pool their resources and work collaboratively toward a common goal. This view of the grid requires the development of conventions for access privileges to various resources, including not only hardware but also content and programs. It also requires the development of protocols for communication between various clusters or even various grids.

Coordinated resource sharing requires the precise identification of which resources are shared, who is allowed to share these resources, and in what manner they may be shared. Moreover, the underlying infrastructure must provide the ability to discover appropriate resources that a task may need anywhere on the grid.

Development of protocols that address these sharing and communication issues, called *intergrid protocols*, are key to the widespread adoption of the grid model, just as Internet protocols were key to the widespread adoption of the Internet.

Dynamic Communities

As the grid model evolves, it is likely that a single grid will support not just one community but, rather, several different communities. An example of a community is a group of researchers collaborating to design a new drug. The resources needed by this community may

be similar to those needed by another community that is collaborating on a high-energy physics problem. Both these communities should be able to coexist on the same grid. Once again the infrastructure of the grid must ensure that adequate means are provided to identify, monitor, and control the needs of each community and to isolate one community from another.

Communities need not be static. Communities may spring up for the purpose of solving a problem and dissolve soon after this goal is achieved. A user may enter a community at some point during the lifetime of the community and exit at some later point. A user may even be a member of two different communities at the same time. All these scenarios require that adequate provision be made in the architecture and the interfaces while designing a grid.

Whether or not the grid concept evolves to embrace one unified standard protocol, such as the Internet, remains to be seen. Today there are several different grids, each with its own protocols and each having its own suite of tools and management software. It is quite possible that the model that eventually emerges will be that of multiple grids, one for each community, working on a larger physical grid that pools an appropriate fraction of its resources for each of the community grids. Each community grid, with its own set of protocols, may actually be viewed as a *virtual grid*, much the way virtual machines with different operating systems coexist on a virtual machine system.

10.3.2 Evolution of the Grid Computing Model: The Globus Toolkit

A computing grid can take on several different forms, ranging from a simple network of workstations taking advantage of idle cycles on user machines to a complex commercial server that satisfies the requirements of users who may wish to contract out a single application or an entire virtual machine. It is likely that no single paradigm

tion or an entire virtual machine. It is likely that no single paradigm will emerge as the winner. Rather, the grid is likely to evolve ultimately as a service that customizes the solution to the needs of the user.

The number of grid systems being deployed is growing by leaps and bounds. Most of the current efforts are aimed at bringing together researchers in specific communities. **Table 10.2** contains different classes of users of the grid ranging from system administrators to the end user. Each of these classes has a purpose distinct from the others and it is important that any toolkit developed to facilitate the use of the grid satisfies the needs of all of these classes of users. The grid developers provide the basic services required to construct the grid. Tool developers construct the programming models and associated tools which are used by the application developers, who construct grid-enabled applications for the end user.

Today there is an active effort to provide basic infrastructure tools to enable users to deploy and manage grids of their own. The Globus project (Globus), for example, is an open-source effort that aims to provide core services, interfaces, and protocols that enable users to create a new class of applications accessing remote resources seamlessly while allowing resources to remain under local control. The Globus project distributes a toolkit that includes software services and libraries to perform discovery, monitoring, and management of resources, to enhance security of transactions on the grid, and to perform more routine functions, such as file management. The expectation is that it will provide a substrate not only for collaboration among the science and engineering communities, but also for businesses and corporations. The effort was originally funded by DARPA but today is sponsored by several federal agencies as well as by corporations.

TABLE 10.2 Classes of Grid Users (**Foster and Kesselman 1998**)

Class	Primary	Major Users	Comments
-------	---------	-------------	----------

Class	Purpose	Makes use of	Concerns
End users	Solve problems	Applications	Transparency, performance
Application developers	Develop applications	Programming models, tools	Ease of use, performance
Tool developers	Develop tools, programming models	Grid services	Adaptivity, exposure of performance, security
Grid developers	Provide basic grid services	Local system services	Local simplicity, connectivity, security
System administrators	Manage grid resources	Management tools	Balancing local and global concerns

10.3.3 Comparison with Conventional Virtual Machines

In this section we try to highlight the similarities and differences between the concept of a grid and that of other virtual machines we have been describing in this book.

Efficient Utilization of Resources

The eventual goal for the first grid systems was the desire to put to use the idle cycles available on the hundreds of millions of computers in the world today. There are many fundamental problems that require a lot of computation power but whose solutions do not need the large, expensive supercomputers commonly employed for such problems. Harnessing the unused resources toward solving such large problems would be an efficient use of the world's computational resources. This is similar to the motivation that led to the development of system virtual machines and to multiprocessor virtual machines, including logical partitioning.

Sharing of Resources

System virtual machines for both uniprocessors and multiprocessors support sharing of resources. However, unlike grid computing, the sharing in a conventional system VM is usually limited to the physical hardware resources—the VMM in such systems is typically not concerned with the sharing of content, for example.

A central concept in grid computing is the sharing of resources of disparate systems by a community of users working toward a common goal. The discovery of resources needed to solve a problem and the negotiation for use of such resources with their owners play a big role on a grid.

Distributed Versus Centralized Control

Compared to traditional virtual machines, the grid has a global scope, with resources spread over large areas, perhaps across the world. Control of shared resources therefore cannot be done in any centralized manner. As with the Internet, users of a grid may collaborate with each other to determine appropriate sharing and use of resources. This is quite unlike a conventional virtual machine monitor, which must have a centralized view of resources and must maintain strict control over the use of resources in order not to compromise the privacy and security of users.

Heterogeneous Nodes

Like operating systems, virtual machine monitors and hypervisors generally do not span nodes of a cluster. A VMM works best when it has the flexibility to reallocate virtual machines among a set of similar real processors. The components in a node may be somewhat dissimilar in terms of their sizes and perhaps in the nature of devices attached to them, but they are similar in terms of the ISA supported by the processors.

On a grid, the expected degree of heterogeneity in the types of components available is much higher. Each cluster could have a processor, memory, and I/O configuration quite different from other clusters on the grid. Moreover, the processors in a cluster could be running a different type of operating system or even a different ISA from those in other clusters. The set of programs that can be run on one cluster may be considerably different from the set that can be