Unit 3: Processes

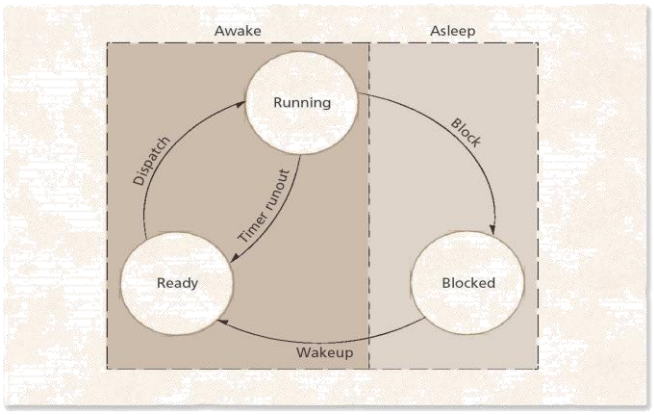
Contents:

Process

Process is the program in execution.

A process can be in any of the following states:

- Running state- The process which is currently being executed by CPU. Only one process can be in the running state.
- Ready state- The process which is waiting to be executed by CPU. It will be executed as soon as it gets the CPU time.
- Blocked state- The process which is waiting for some event to happen. It is suspended till the event like I/O operation is completed.



Process Control Block (PCB) is a data structure that contains information about the process. It is used to save the process as it has to be saved when the process is switched from one state to another so that it can be restarted later. A PCB contains information like:

1. Process state: running, ready, blocked.
2. Program counter: Address of next instruction for the process.
3. Registers: Stack pointer, accumulator etc.
4. Scheduling information: Process priority, pointer to scheduling queue
5. Memory-allocation: value of base and limit register, page table, segment table
6. Accounting information: time limit, process numbers etc.
7. Status information: list of I/O devices, list of open files etc.

An important issue related to process, especially in wide-area distributed systems, is moving processes between different machines. Process migration or more specifically, code migration, can help in achieving scalability, but can also help to dynamically configure clients and servers.

To understand the role of threads in distributed systems, it is important to understand what a process is, and how processes and threads relate. To execute a program, an operating system creates a number of **virtual processors**, each one for running a different program. To keep track of these virtual processors, the operating system has a **process table**, containing entries to store CPU register values, memory maps, open files, accounting information, privileges, etc. Jointly, these entries form a **process context**.

A **process** is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors. An important issue is that the operating system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behavior. In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent. Usually, the operating system requires hardware support to enforce this separation.

This concurrency transparency comes at a price. For example, each time a process is created, the operating system must create a complete independent address space. Allocation can mean initializing memory segments by, for example, zeroing a data segment, copying the associated program into a text segment, and setting up a stack for temporary data. Likewise, switching the CPU between two processes may require some effort as well. Apart from saving the data as currently stored in various registers (including the program counter and stack pointer), the operating system will also have to modify registers of the memory management unit (MMU) and invalidate address translation caches such as in the translation lookaside buffer (TLB). In addition, if the operating system supports more processes than it can simultaneously hold in main memory, it may have to swap processes between main memory and disk before the actual switch can take place.

3.1 Threads

Thread is a lightweight process. Threads are very useful in modern programming whenever a process has multiple tasks to perform independently of the others. This is particularly true when one of the tasks may block, and it is desired to allow the other tasks to proceed without blocking.

For example: In a word processor, a background thread may check spelling and grammar while a foreground thread processes user inputs, while yet a third thread loads images from the hard drive and a fourth does periodic automatic backups of the file being edited.

There are four major benefits of multi-threading:

1. Responsiveness – One thread may provide rapid response while other threads are blocked or slowed down doing intensive calculations.
2. Resources haring – By default threads share common code, data, and other resources, which allows multiple tasks to be performed simultaneously in a single address space.
3. Economy – Creating and managing threads (and context switches between them) is much faster than performing the same tasks for processes.
4. Scalability i.e. utilization of multiprocessor architecture – A single threaded process can only run on one CPU, no matter how many may be available, whereas the execution of multi-threaded application may be split amongst available processors.

Like a process, a thread executes its own piece of code, independently from other threads. There are two important implications of deploying threads:

- The performance of a multithreaded application need hardly ever be worse than that of its single-threaded counterpart. In fact, in many cases, multithreading even leads to a performance gain.
- Threads are not automatically protected against each other the way processes are, development of multithreaded applications requires additional intellectual effort. Proper design and keeping things simple, as usual, help a lot.

Thread usage in non-distributed systems/traditional system

There are several benefits to multithreaded processes that have increased the popularity of using thread systems.

- **Avoid needless blocking**: The most important benefit comes from the fact that in a single-threaded process, whenever a blocking system call is executed, the process as a whole is blocked. If there is only a single thread of control, computation cannot proceed while the program is waiting for input. Likewise, it is not easy to provide input while dependencies are being calculated. The easy solution is to have at least two threads of control: one for handling interaction with the user and one for updating the spreadsheet. In the meantime, a third thread could be used for backing up the spreadsheet to disk while the other two are doing their work.
- **Exploit parallelism**: Another advantage of multithreading is that it becomes possible to exploit parallelism when executing the program on a multiprocessor or multicore system. In that case, each thread is assigned to a different CPU or core while shared data are stored in shared main memory. When properly designed, such parallelism can be transparent: the process will run equally well on a uniprocessor system, albeit slower. Such computer systems are typically used for running servers in client-server applications, but are by now also extensively used in devices such as smartphones.
- **Avoid process switching**: Multithreading is also useful in the context of large applications. Such applications are often developed as a collection of cooperating programs, each to be executed by a separate process. This approach is typical for a UNIX environment. Cooperation between programs is implemented by means of inter-process communication (IPC) mechanisms. The major drawback of all IPC mechanisms is that communication often requires relatively extensive context switching. Instead of using processes, an application can also be constructed such that different parts are executed by separate threads. Communication between those parts is entirely dealt with by using shared data. The effect can be a dramatic improvement in performance.
- Finally, there is also a pure software engineering reason to use threads: many applications are simply easier to structure as a collection of cooperating threads. Think of applications that need to perform several (more or less independent) tasks, like word processor, spreadsheet.

Thread implementation

Threads are often provided in the form of a thread package. Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as mutexes and condition variables.

There are basically two approaches to implement a thread package:

1. To construct a thread library that is executed entirely in user space.

2. To have the kernel be aware of threads and schedule them.

A user-level thread library has a number of advantages:

- It is cheap to create and destroy threads. Because all thread administration is kept in the user's address space, the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack. Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer used. Both operations are cheap.
- Switching thread context can often be done in just a few instructions. Basically, only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched. There is no need to change memory maps, flush the TLB, do CPU accounting, and so on. Switching thread context is done when two threads need to synchronize.

A major drawback of user-level threads comes from deploying the **many-to-one threading model**: multiple threads are mapped to a single schedulable entity. As a consequence, the invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process. In that case, blocking on I/O should not prevent other parts to be executed in the meantime. For such applications, user-level threads are of no help.

These problems can be mostly avoided by implementing threads in the operating system's kernel, leading to what is known as the **one-to-one threading model** in which every thread is a schedulable entity. The price to pay is that every thread operation (creation, deletion, synchronization, etc.), will have to be carried out by the kernel, requiring a system call. Switching thread contexts may now become as expensive as switching process contexts. However, in light of the fact that the performance of context switching is generally dictated by ineffective use of memory caches, and not by the distinction between the many-to-one or one-to-one threading model, many operating systems now offer the latter model, if only for its simplicity.

Threads in distributed systems

An important property of threads is that they can provide a convenient means of allowing blocking system calls without blocking the entire process in which the thread is running. This property makes threads particularly attractive to use in distributed systems as it makes it much easier to express communication in the form of maintaining multiple logical connections at the same time.

Multithreaded clients

To establish a high degree of distribution transparency, distributed systems that operate in wide-area networks may need to conceal long inter-process message propagation times. The round-trip delay in a wide-area network can easily be in the order of hundreds of milliseconds, or sometimes even seconds. The usual way to hide communication latencies is to initiate communication and immediately proceed with something else.

A typical example where this happens is in Web browsers. In many cases, a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc. To fetch each element of a Web document, the browser has to set up a TCP/IP connection, read the incoming data, and pass it to a display component. Setting up a connection as well as reading incoming data are inherently blocking

operations. When dealing with long-haul communication, we also have the disadvantage that the time for each operation to complete may be relatively long.

A Web browser often starts with fetching the HTML page and subsequently displays it. To hide communication latencies as much as possible, some browsers start displaying data while it is still coming in. While the text is made available to the user, including the facilities for scrolling and such, the browser continues with fetching other files that make up the page, such as the images. The latter are displayed as they are brought in. The user need thus not wait until all the components of the entire page are fetched before the page is made available.

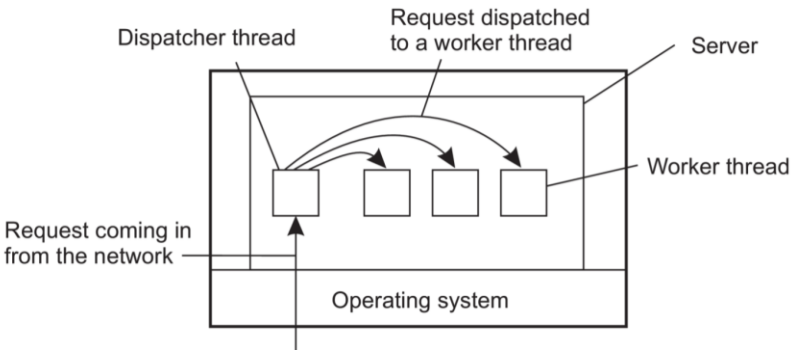
In effect, it is seen that the Web browser is doing a number of tasks simultaneously. As it turns out, developing the browser as a multithreaded client simplifies matters considerably. As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts. Each thread sets up a separate connection to the server and pulls in the data. Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process. Meanwhile, the user notices only delays in the display of images and such, but can otherwise browse through the document.

There is another important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously. In the previous example, several connections were set up to the same server. If that server is heavily loaded, or just plain slow, no real performance improvements will be noticed compared to pulling in the files that make up the page strictly one after the other. However, in many cases, Web servers have been replicated across multiple machines, where each server provides exactly the same set of Web documents. The replicated servers are located at the same site, and are known under the same name. When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load balancing technique. When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a non-replicated server. This approach is possible only if the client can handle truly parallel streams of incoming data. Threads are ideal for this purpose.

Multithreaded Servers

Although there are important benefits to multithreaded clients, the main use of multithreading in distributed systems is found at the server side. Practice shows that multithreading not only simplifies server code considerably, but also makes it much easier to develop servers that exploit parallelism to attain high performance, even on uniprocessor systems. However, with modern multicore processors, multithreading for parallelism is an obvious path to follow.

To understand the benefits of threads for writing server code, consider the organization of a file server that occasionally has to block waiting for the disk. The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply. One possible, and particularly popular organization is shown in Figure below. Here one thread, the dispatcher, reads incoming requests for a file operation. The requests are sent by clients to a well-known end point for this server. After examining the request, the server chooses an idle (i.e., blocked) worker thread and hands it the request.



The worker proceeds by performing a blocking read on the local file system, which may cause the thread to be suspended until the data are fetched from disk. If the thread is suspended, another thread is selected to be executed. For example, the dispatcher may be selected to acquire more work. Alternatively, another worker thread can be selected that is now ready to run.

Now consider how the file server might have been written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. Consequently, requests from other clients cannot be handled. In addition, if the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests per time unit can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

Note that instead of using threads, we can also use multiple processes to organize a server (leading to the situation that we actually have a multi-process server). The advantage is that the operating system can offer more protection against accidental access to shared data. However, if processes need to communicate a lot, we may see a noticeable adverse effect on performance in comparison to using threads.

3.2 Virtualization

Threads and processes can be seen as a way to do more things at the same time. In effect, they allow us to build (pieces of) programs that appear to be executed simultaneously. On a single-processor computer, this simultaneous execution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time. By rapidly switching between threads and processes, the illusion of parallelism is created.

This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as **resource virtualization**. This virtualization has been applied for many decades, but has received renewed interest as (distributed) computer systems have become more commonplace and complex, leading to the situation that application software is mostly always outliving its underlying systems software and hardware.

Principle of virtualization

In practice, every (distributed) computer system offers a programming interface to higher-level software. There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware

systems. In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system.

Virtualizations creates an environment wherein it emulates and imitates various hardware components like CPU, OS, software, I/O devices and storage devices to numerous virtual machines (VM). Each node in the distributed system is a virtual machine running independently.

Virtualization creates a virtual version of a device or resource, such as a server, storage device, network or even an operating system where the framework divides the resource into one or more execution environments.

Virtualization and distributed systems

One of the most important reasons for introducing virtualization back in the 1970s, was to allow legacy software to run on expensive mainframe hardware. The software not only included various applications, but in fact also the operating systems they were developed for. This approach toward supporting legacy software has been successfully applied on the IBM 370 mainframes (and their successors) that offered a virtual machine to which different operating systems had been ported.

As hardware became cheaper, computers became more powerful, and the number of different operating system flavors was reducing, virtualization became less of an issue. However, matters have changed again since the late 1990s. First, while hardware and low-level systems software change reasonably fast, software at higher levels of abstraction (e.g., middleware and applications), are often much more stable. In other words, we are facing the situation that legacy software cannot be maintained in the same pace as the platforms it relies on. Virtualization can help here by porting the legacy interfaces to the new platforms and thus immediately opening up the latter for large classes of existing programs.

Equally important is the fact that networking has become completely pervasive. It is hard to imagine that a modern computer is not connected to a network. In practice, this connectivity requires that system administrators maintain a large and heterogeneous collection of server computers, each one running very different applications, which can be accessed by clients. At the same time the various resources should be easily accessible to these applications. Virtualization can help a lot: the diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possibly including the related libraries and operating system, which, in turn, run on a common platform.

Types of virtualization

1. Desktop virtualization: It is when the host server can run virtual machines using a hypervisor (a software program). A hypervisor can directly be installed on the host machines or over the operating system (like Windows, Mac, and Linux). Virtualized desktop don't use the host system's hardware drive; instead they run on a remote central server. This type of virtualization is useful for development and testing teams who need to develop or test applications on different operating system.
2. Application virtualization: The process of installing an application on a central server (single computer system) that can virtually be operated on multiple systems is known as application virtualization. For end users, the virtualized application works exactly like a native application installed on a physical machine. With application virtualization, it's easier for organization to update, maintain, and fix applications centrally. Another benefit of application virtualization

is portability. It allows users to access virtualized applications even on non-Windows devices such as iOS or Android.

3. Server virtualization: It is the process of partitioning the resources of a single server into multiple virtual servers. These virtual servers can run as separate machines. It allows businesses to run multiple independent Oss all with different configurations using a single (host) server. The process also saves the hardware cost involved in keeping a host of physical servers.
4. Network virtualization: It helps to manage and monitor the entire computer network as a single administrative entity. Admin can keep track of various elements of network infrastructure such as routers and switches from a single software-based administrator's console. It helps for network optimization for data transfer rates, flexibility, reliability, security and scalability. It improves the overall network's productivity and efficiency.
5. Storage virtualization: It is the process of pooling physical storage of multiple network storage devices so it looks like a single storage device. Storage virtualization facilitates archiving, easy backup, and recovery tasks. It helps administrators allocate, move, change and set up resources efficiently across the organizational infrastructure.

Advantages of Virtualization:

1. It is cheaper as it doesn't require actual hardware components.
2. It keeps cost predictable.
3. It reduces the workload.
4. It offers a better uptime.
5. It supports for faster deployment of resources.
6. It promotes digital entrepreneurship.
7. It provides energy saving.

Disadvantages of Virtualization:

1. It can have a high cost of implementation.
2. It creates a security risk.
3. It creates an availability issue.
4. It still has limitations.

Application of virtual machines to distributed systems

From the perspective of distributed systems, the most important application of virtualization lies in cloud computing. Cloud providers offer roughly three different types of services:

- Infrastructure-as-a-Service (IaaS) covering the basic infrastructure
- Platform-as-a-Service (PaaS) covering system-level services
- Software-as-a-Service (SaaS) containing actual applications

Virtualization plays a key role in IaaS. Instead of renting out a physical machine, a cloud provider will rent out a virtual machine (monitor) that may, or may not, be sharing a physical machine with other customers. The beauty of virtualization is that it allows for almost complete isolation between customers, who will indeed have the illusion that they have just rented a dedicated physical machine. Isolation is, however, never complete, if only for the fact that the actual physical resources are shared, in turn leading to observable lower performance.

3.3 Clients

A major task of client machines is to provide the means for users to interact with remote servers. There are roughly two ways in which this interaction can be supported.

First, for each remote service the client machine will have a separate counterpart that can contact the service over the network. A typical example is a calendar running on a user's smartphone that needs to synchronize with a remote, possibly shared calendar. In this case, an application-level protocol will handle the synchronization, as shown in Figure below.

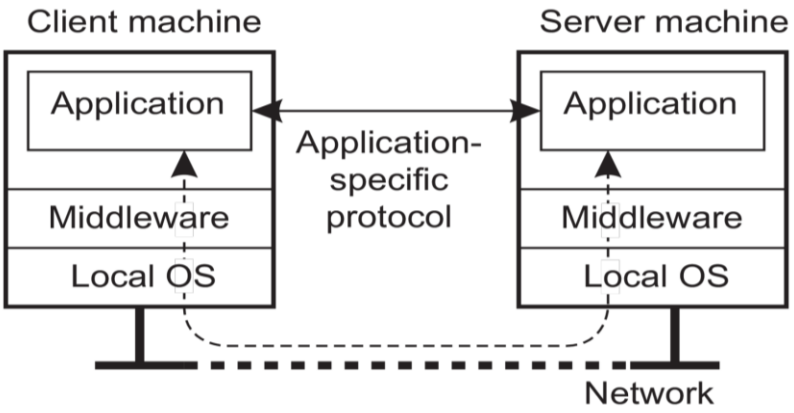


Figure: A networked application with its own protocol

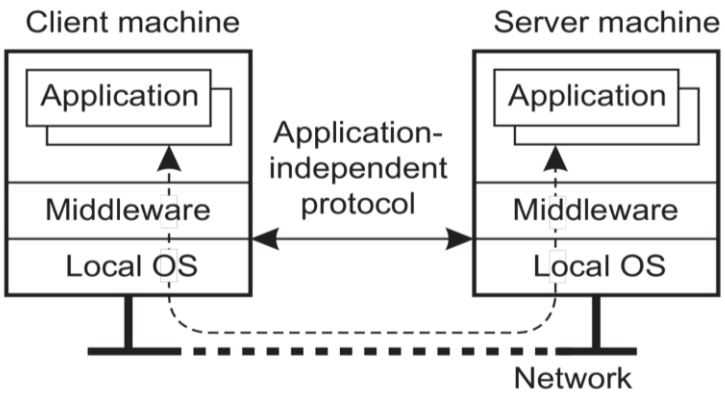


Figure: A general solution to allow access to remote applications

A second solution is to provide direct access to remote services by offering only a convenient user interface. Effectively, this means that the client machine is used only as a terminal with no need for local storage, leading to an application-neutral solution as shown in Figure above. In the case of networked user interfaces, everything is processed and stored at the server. This thin-client approach has received much attention with the increase of Internet connectivity and the use of mobile devices. Thin-client solutions are also popular as they ease the task of system management.

Example: The X window system

X Window System is one of the oldest and still widely used networked user interfaces. The X Window System, generally referred to simply as X, is used to control bit-mapped terminals, which include a monitor, keyboard, and a pointing device such as a mouse. Next to supporting traditional terminals as can

be found with desktop computers and workstations, X also supports modern devices such a touchscreens on tablets and smartphones. In a sense, X can be viewed as that part of an operating system that controls the terminal. The heart of the system is formed by what we shall call the X kernel. It contains all the terminal-specific device drivers, and as such, is generally highly hardware dependent.

The X kernel offers a relatively low-level interface for controlling the screen, but also for capturing events from the keyboard and mouse. This interface is made available to applications as a library called Xlib. This general organization is shown in Figure below. Note that Xlib is hardly ever used directly by applications, which instead deploy easier to use toolkits implemented on top of Xlib.

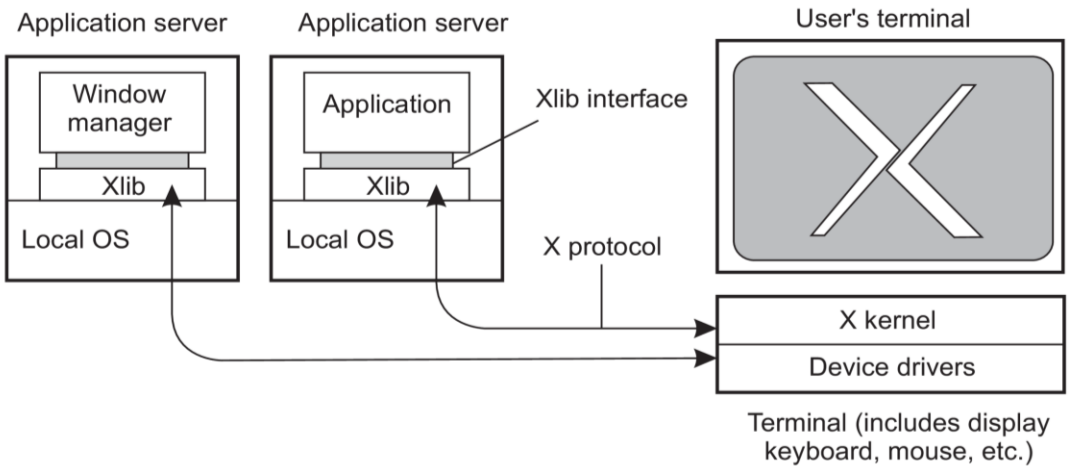


Figure: The basic organization of the X Window system

The interesting aspect of X is that the X kernel and the X applications need not necessarily reside on the same machine. In particular, X provides the X protocol, which is an application-level communication protocol by which an instance of Xlib can exchange data and events with an X kernel. For example, Xlib can send requests to the X kernel for creating or killing a window, setting colors, and defining the type of cursor to display, among many other requests. In turn, the X kernel will react to local events such as keyboard and mouse input by sending event packets back to Xlib.

Thin-client network computing

Obviously, applications manipulate a display using the specific display commands as offered by X. These commands are generally sent over the network where they are subsequently executed by the X kernel. By its nature, applications written for X should preferably separate application logic from user-interface commands. Unfortunately, this is often not the case. As reported by Lai and Nieh [2002] it turns out that much of the application logic and user interaction are tightly coupled, meaning that an application will send many requests to the X kernel for which it will expect a response before being able to make a next step. This synchronous behavior may adversely affect performance when operating over a wide-area network with long latencies.

There are several solutions to this problem.

- One is to re-engineer the implementation of the X protocol. An important part of this work concentrates on bandwidth reduction by reducing the size of X messages. To this end, messages are considered to consist of a fixed part, which is treated as an identifier, and a variable part. In many cases, multiple messages will have the same identifier in which case they will often contain

similar data. This property can be used to send only the differences between messages having the same identifier. By having the sender and receiver maintain identifiers, decoding at the receiver can be readily applied. Bandwidth reductions up to a factor 1000 have been reported, which allows X to also run through low-bandwidth links of only 9600 kbps.

- As an alternative to using X, researchers and practitioners have also sought to let an application completely control the remote display, that is, up the pixel level. Changes in the bitmap are then sent over the network to the display, where they are immediately transferred to the local frame buffer. A well-known example of this approach is Virtual Network Computing (VNC). Obviously, letting the application control the display requires sophisticated encoding techniques in order to prevent bandwidth availability to become a problem. For example, consider displaying a video stream at a rate of 30 frames per second on a simple 320 × 240 screen. If each pixel is encoded by 24 bits, then without an efficient encoding scheme, we would need a bandwidth of approximately 53 Mbps. In practice, various encoding techniques are used, yet choosing the best one is generally application dependent.

The drawback of sending raw pixel data in comparison to higher-level protocols such as X is that it is impossible to make any use of application semantics, as these are effectively lost at that level. Baratto et al. [2005] propose a different technique. In their solution, referred to as THINC, they provide a few high-level display commands that operate at the level of the video device drivers. These commands are thus device dependent, more powerful than raw pixel operations, but less powerful compared to what a protocol such as X offers. The result is that display servers can be much simpler, which is good for CPU usage, while at the same time application-dependent optimizations can be used to reduce bandwidth and synchronization.

Client-side software for distribution transparency

Client software comprises more than just user interfaces. In many cases, parts of the processing and data level in a client-server application are executed on the client side as well. A special class is formed by embedded client software, such as for automatic teller machines (ATMs), cash registers, barcode readers, TV set-top boxes, etc. In these cases, the user interface is a relatively small part of the client software, in contrast to the local processing and communication facilities.

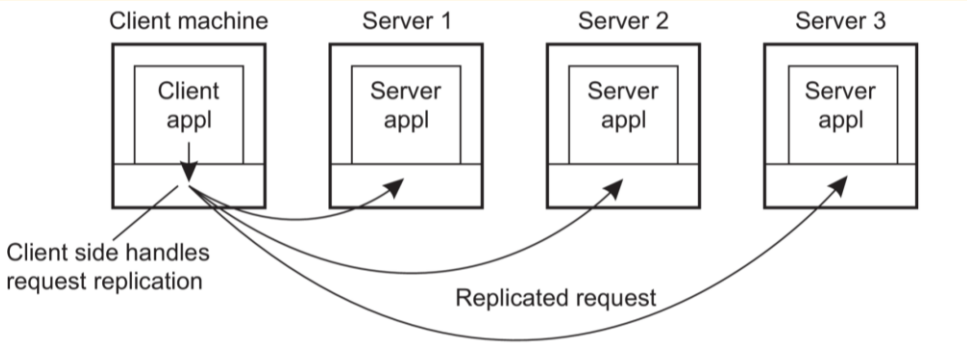
Besides the user interface and other application-related software, client software comprises components for achieving **distribution transparency**. Ideally, a client should not be aware that it is communicating with remote processes. In contrast, distribution is often less transparent to servers for reasons of performance and correctness.

Access transparency is generally handled through the generation of a client stub from an interface definition of what the server has to offer. The stub provides the same interface as the one available at the server, but hides the possible differences in machine architectures, as well as the actual communication. The client stub transforms local calls to messages that are sent to the server, and vice versa transforms messages from the server to return values as one would expect when calling an ordinary procedure.

There are different ways to handle **location**, **migration**, and **relocation transparency**. Using a convenient naming system is crucial. In many cases, cooperation with client-side software is also important. For example, when a client is already bound to a server, the client can be directly informed when the server changes location. In this case, the client's middleware can hide the server's current network location from

the user, and also transparently rebind to the server if necessary. At worst, the client's application may notice a temporary loss of performance.

In a similar way, many distributed systems implement **replication transparency** by means of client-side solutions. For example, imagine a distributed system with replicated servers, such replication can be achieved by forwarding a request to each replica, as shown in Figure. Client-side software can transparently collect all responses and pass a single response to the client application.



Regarding **failure transparency**, masking communication failures with a server is typically done through client middleware. For example, client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts. There are even situations in which the client middleware returns data it had cached during a previous session, as is sometimes done by Web browsers that fail to connect to a server.

Finally, **concurrency transparency** can be handled through special intermediate servers, notably transaction monitors, and requires less support from client software.

3.4 Servers

General design issues

A server is a process implementing a specific service on behalf of a collection of clients. In essence, each server is organized in the same way: it waits for an incoming request from a client and subsequently ensures that the request is taken care of, after which it waits for the next incoming request.

Concurrent versus iterative servers

There are several ways to organize servers. In the case of an **iterative server**, the server itself handles the request and, if necessary, returns a response to the requesting client.

A **concurrent server** does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request. A multithreaded server is an example of a concurrent server. An alternative implementation of a concurrent server is to fork() a new process for each new incoming request. This approach is followed in many UNIX systems. The thread or process that handles the request is responsible for returning a response to the requesting client.

Contacting a server: end points

Another issue is where clients contact a server. In all cases, clients send requests to an end point, also called a port, at the machine where the server is running. Each server listens to a specific end point. How do clients know the end point of a service? One approach is to globally assign end points for well-known

services. For example, servers that handle Internet FTP requests always listen to TCP port 21. Likewise, an HTTP server for the World Wide Web will always listen to TCP port 80. These end points have been assigned by the Internet Assigned Numbers Authority (IANA). With assigned end points, the client needs to find only the network address of the machine where the server is running. Name services can be used for that purpose.

There are many services that do not require a preassigned end point. For example, a time-of-day server may use an end point that is dynamically assigned to it by its local operating system. In that case, a client will first have to look up the end point. One solution is to have a special daemon running on each machine that runs servers. The daemon keeps track of the current end point of each service implemented by a co-located server. The daemon itself listens to a well-known end point. A client will first contact the daemon, request the end point, and then contact the specific server, as shown in Figure.

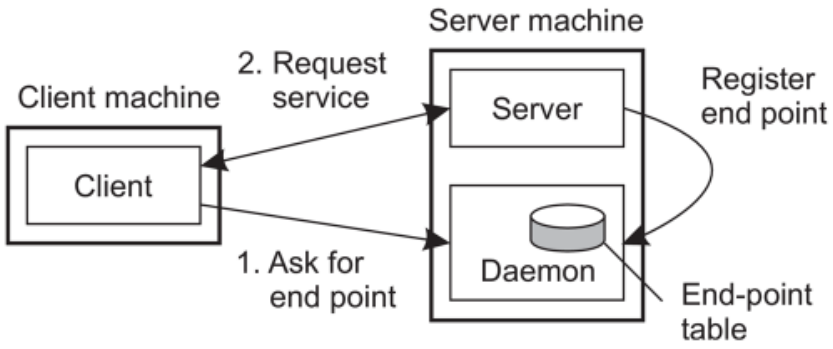


Figure: Client-to-Server binding using a daemon

It is common to associate an end point with a specific service. However, actually implementing each service by means of a separate server may be a waste of resources. For example, in a typical UNIX system, it is common to have lots of servers running simultaneously, with most of them passively waiting until a client request comes in. Instead of having to keep track of so many passive processes, it is often more efficient to have a single super-server listening to each end point associated with a specific service, as shown in Figure. For example, the inetd daemon in UNIX listens to a number of well-known ports for Internet services. When a request comes in, the daemon forks a process to handle it. That process will exit when finished.

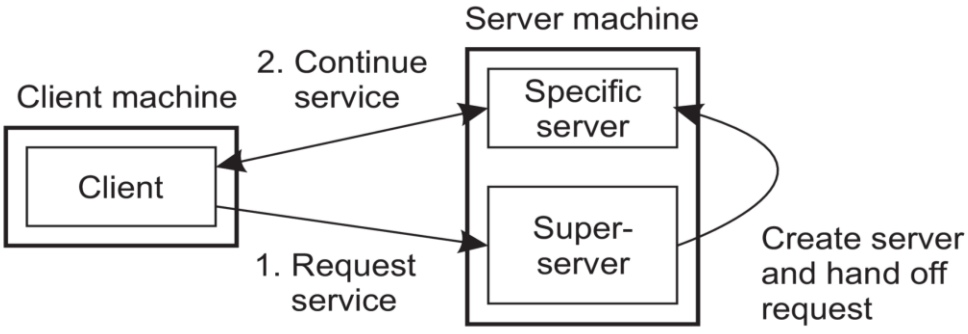


Figure: Client-to-Server binding using a super-server

Interrupting a server

Another issue that needs to be taken into account when designing a server is whether and how a server can be interrupted. For example, consider a user who has just decided to upload a huge file to an FTP server. Then, suddenly realizing that it is the wrong file, he wants to interrupt the server to cancel further data transmission. There are several ways to do this. One approach that works only too well in the current Internet (and is sometimes the only alternative) is for the user to abruptly exit the client application (which will automatically break the connection to the server), immediately restart it, and pretend nothing happened. The server will eventually tear down the old connection, thinking the client has probably crashed.

A much better approach for handling communication interrupts is to develop the client and server such that it is possible to send **out-of-band** data, which is data that is to be processed by the server before any other data from that client. One solution is to let the server listen to a separate control end point to which the client sends out-of-band data, while at the same time listening (with a lower priority) to the end point through which the normal data passes. Another solution is to send out-of-band data across the same connection through which the client is sending the original request. In TCP, for example, it is possible to transmit urgent data. When urgent data are received at the server, the latter is interrupted (e.g., through a signal in UNIX systems), after which it can inspect the data and handle them accordingly.

Stateless versus stateful servers

A final, important design issue, is whether or not the server is stateless.

A **stateless server** does not keep information on the state of its clients, and can change its own state without having to inform any client. A Web server, for example, is stateless. It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or (most often) for fetching a file. When the request has been processed, the Web server forgets the client completely. Likewise, the collection of files that a Web server manages (possibly in cooperation with a file server), can be changed without clients having to be informed.

Note that in many stateless designs, the server actually does maintain information on its clients, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server. For example, a Web server generally logs all client requests. This information is useful, for example, to decide whether certain documents should be replicated, and where they should be replicated to. Clearly, there is no penalty other than perhaps in the form of suboptimal performance if the log is lost.

A particular form of a stateless design is where the server maintains what is known as **soft state**. In this case, the server promises to maintain state on behalf of the client, but only for a limited time. After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client. An example of this type of state is a server promising to keep a client informed about updates, but only for a limited time. After that, the client is required to poll the server for updates. Soft-state approaches originate from protocol design in computer networks, but can be equally applied to server design.

In contrast, a **stateful server** generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is a file server that allows a client to keep a local copy of a file, even for performing update operations. Such a server would maintain

a table containing (client, file) entries. Such a table allows the server to keep track of which client currently has the update permissions on which file, and thus possibly also the most recent version of that file.

This approach can improve the performance of read and write operations as perceived by the client. Performance improvement over stateless servers is often an important benefit of stateful designs. However, the example also illustrates the major drawback of stateful servers. If the server crashes, it has to recover its table of (client, file) entries, or otherwise it cannot guarantee that it has processed the most recent updates on a file. In general, a stateful server needs to recover its entire state as it was just before the crash. Enabling recovery can introduce considerable complexity. In a stateless design, no special measures need to be taken at all for a crashed server to recover. It simply starts running again, and waits for client requests to come in.

Object servers

An object server is a server tailored to support distributed objects. The important difference between a general object server and other (more traditional) servers is that an object server by itself does not provide a specific service. Specific services are implemented by the objects that reside in the server. Essentially, the server provides only the means to invoke local objects, based on requests from remote clients. As a consequence, it is relatively easy to change services by simply adding and removing objects.

An object server thus acts as a place where objects live. An object consists of two parts: data representing its state and the code for executing its methods. Whether or not these parts are separated, or whether method implementations are shared by multiple objects, depends on the object server. Also, there are differences in the way an object server invokes its objects. For example, in a multithreaded server, each object may be assigned a separate thread, or a separate thread may be used for each invocation request.

For an object to be invoked, the object server needs to know which code to execute, on which data it should operate, whether it should start a separate thread to take care of the invocation, and so on. A simple approach is to assume that all objects look alike and that there is only one way to invoke an object. Unfortunately, such an approach is generally inflexible and often unnecessarily constrains developers of distributed objects.

A much better approach is for a server to support different policies. Consider, for example, a transient object: an object that exists only as long as its server exists, but possibly for a shorter period of time. An in-memory, read-only copy of a file could typically be implemented as a transient object. Likewise, a calculator could also be implemented as a transient object. A reasonable policy is to create a transient object at the first invocation request and to destroy it as soon as no clients are bound to it anymore.

The advantage of this approach is that a transient object will need a server's resources only as long as the object is really needed. The drawback is that an invocation may take some time to complete, because the object needs to be created first. Therefore, an alternative policy is sometimes to create all transient objects at the time the server is initialized, at the cost of consuming resources even when no client is making use of the object.

In a similar fashion, a server could follow the policy that each of its objects is placed in a memory segment of its own. In other words, objects share neither code nor data. Such a policy may be necessary when an object implementation does not separate code and data, or when objects need to be separated for security reasons.

The alternative approach is to let objects at least share their code. For example, a database containing objects that belong to the same class can be efficiently implemented by loading the class implementation only once into the server. When a request for an object invocation comes in, the server need only fetch that object's state and execute the requested method.

3.5 Code Migration

Traditionally, code migration in distributed systems take place in the form of process migration in which an entire process is moved from one machine to another. The basic idea is that the overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.

Code migration in the broadest sense deals with moving programs between machines, with the intention to have those programs be executed at the target.

Reasons for migrating code

Moving a running process to a different machine is a costly and complex task, and there had better be a good reason for doing so.

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

Process migration was no longer a viable option for improving distributed systems. However, instead of offloading machines, we can now witness that code is moved to make sure that a machine is sufficiently loaded. In particular, migrating complete virtual machines with their suite of applications to lightly loaded machines in order to minimize the total number of nodes being used is common practice in optimizing energy usage in data centers. Interestingly enough, although migrating virtual machines may require more resources, the task itself is far less complex than migrating a process.

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

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

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

executed at the handheld instead of the server may be the only viable solution to obtain acceptable performance, both for the client and the server.

Support for code migration can also help improve performance by exploiting parallelism, but without the usual intricacies related to parallel programming. A typical example is searching for information in the Web. It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent that moves from site to site. By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speed-up compared to using just a single program instance. However, mobile agents have never become successful because they did not really offer an obvious advantage over other technologies. Moreover, and crucial, it turned out to be virtually impossible to let this type of mobile code operate in a secure way.

Besides improving performance, there are other reasons for supporting code migration as well. The most important one is that of flexibility. The traditional approach to building distributed applications is to partition the application into different parts, and decide in advance where each part should be executed. This approach, for example, has led to different multitier client-server applications.

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.

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 ease many of the problems of porting Pascal to different machines was to generate machine independent intermediate code for an abstract virtual machine. That machine, of course, would need to be implemented on many platforms, but it would then allow Pascal programs to be run anywhere. Although this simple idea was widely used for some years, it never really caught on as the general solution to portability problems for other languages, notably C.

About 25 years later, code migration in heterogeneous systems is being tackled by scripting languages and highly portable languages such as Java. In essence, these solutions adopt the same approach as was done for porting Pascal. All such solutions have in common that they rely on a (process) virtual machine that either directly interprets source code (as in the case of scripting languages), or otherwise interprets intermediate code generated by a compiler (as in Java). Being in the right place at the right time is also important for language developers.

Practice Questions

1. What is a thread? What are the benefits of using multiple threads in both non-distributed system (traditional system) and distributed system? Explain.
2. What are the differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?
3. Explain the concept of multithreaded client and multithreaded server.
4. What is virtualization? Explain with its advantages, disadvantages and types.
5. Explain virtualization in distributed system.
6. What is a client? How is distribution transparency achieved in client side system?

7. Write short note on X-Window system.
8. What is a server? Explain the concepts of concurrent vs iterative server and stateful vs stateless server.
9. Write short note object server.
10. What is code migration? Explain the reasons for code migration.