“DISTRIBUTED SYSTEMS”
Principles and Paradigms
Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Module 1

Chapter 1
Introduction

1.1 Definition :

- A distributed system is:

- ***A collection of independent computers that appears to its users as a single coherent system.***
- In order to support heterogeneous computers and networks while offering a single-system view, distributed systems are often organized by means of a layer of software-that is, logically placed between a higher-level layer consisting of users and applications, and a layer underneath consisting of operating systems and basic communication facilities, as shown in Fig. 1-1.
- Accordingly, such a distributed system is sometimes called *middleware.*

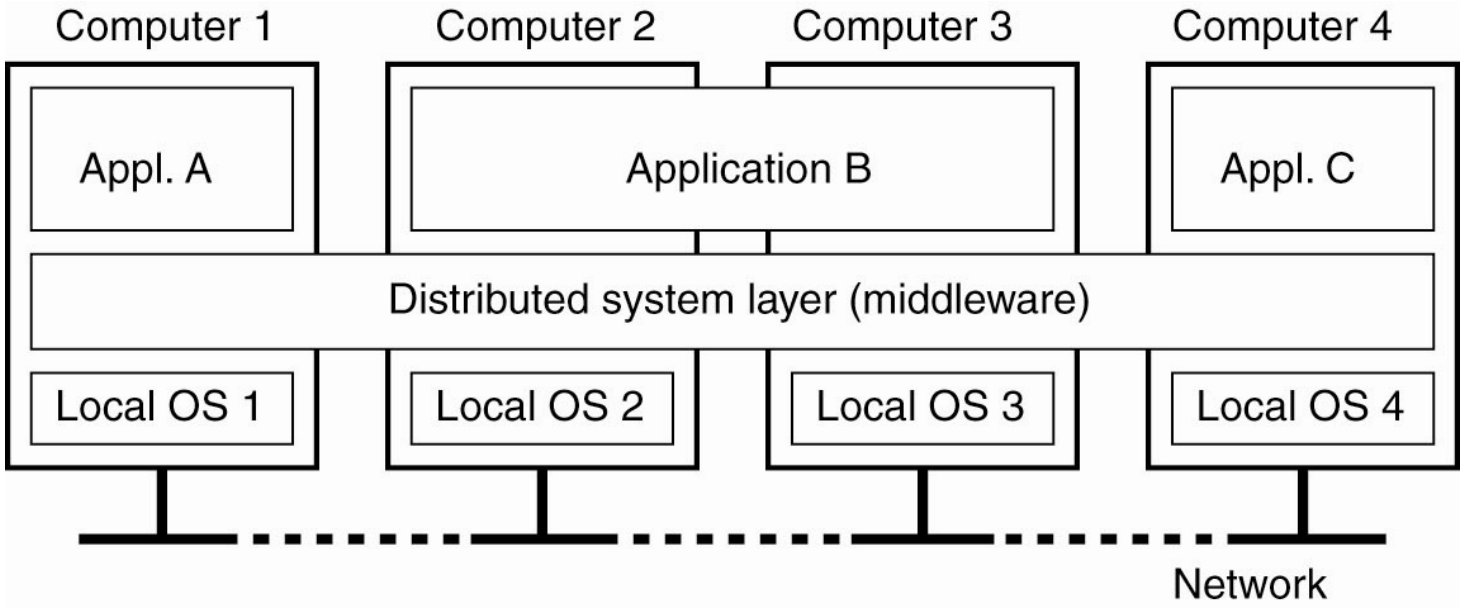


Figure 1-1. A distributed system organized as middleware. The middleware layer extends over multiple machines, and offers each application the same interface.

- Fig. 1-1, shows four networked computers and three applications, of which application B is distributed across computers 2 and 3.
- Each application is offered the same interface.
- The distributed system provides the means for components of a single distributed application to communicate with each other, but also to let different applications communicate.
- At the same time, it hides, as best and reasonable as possible, the differences in hardware and operating systems from each application.

1.2 Goals :

4 important goals that should be met to make building a distributed system worth the effort.

- A distributed system should make resources easily accessible
- It should reasonably hide the fact that resources are distributed across a network
- it should be open
- And it should be scalable(**Scalability** is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.)

1.2.1 Making Resources Accessible

- The main goal of a distributed system is to make it easy for the users (and applications) to access remote resources, and to share them in a controlled and efficient way.
- Resources can be just about anything, but typical examples include things like printers, computers, storage facilities, data, files, Web pages, and networks etc.
- There are many reasons for wanting to share resources. One obvious reason is that of economics.
- For example, it is cheaper to let a printer be shared by several users in a small office than having to buy and maintain a separate printer for each user.

1.2.2 Distribution Transparency

- An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers.
- A distributed system that is able to present itself to users and applications as if it were only a single computer system is said to be transparent.
- Kinds of transparency exist in distributed systems :-

The concept of transparency can be applied to several aspects of a distributed system, the most important ones shown in Fig. 1-2.

Transparency	Description
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource is replicated
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource

Figure 1-2. Different forms of transparency in a distributed system

- ***Access transparency*** deals with hiding differences in data representation and the way that resources can be accessed by users
- For example, a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions.
- Differences in naming conventions, as well as how files can be manipulated, should all be hidden from users and applications.
- ***Location transparency*** refers to the fact that users cannot tell where a resource is physically located in the system.
- Naming plays an important role in achieving location transparency. In particular, location transparency can be achieved by assigning only logical names to resources, that is, names in which the location of a resource is not secretly encoded.
- An example of a such a name is the URL" **http://www.prenhall.com/index.html"** , which gives no clue about the location of Prentice Hall's main Web server.
- The URL also gives no clue as to whether index.html has always been at its current location or was recently moved there.

- Distributed systems in which resources can be moved without affecting how those resources can be accessed are said to provide ***migration transparency*.**
- The situation in which resources can be relocated while they are being accessed without the user or application noticing anything. In such cases, the system is said to support ***relocation transparency***
- An example of relocation transparency is when mobile users can continue to use their wireless laptops while moving from place to place without ever being (temporarily) disconnected.

- ***Replication*** plays a very important role in distributed systems.
- For example, resources may be replicated to increase availability or to improve performance by placing a copy close to the place where it is accessed.
- Replication transparency deals with hiding the fact that several copies of a resource exist.
- To hide replication from users, it is necessary that all replicas have the same name.

- In many cases, sharing resources is done in a cooperative way, as in the case of communication.
- However. there are also many examples of competitive sharing of resources.
- For example, two independent users may each have stored their files on the same file server or may be accessing the same tables in a shared database.
- In such cases, it is important that each user does not notice that the other is making use of the same resource.
- This phenomenon is called ***concurrency transparency***.

- Making a distributed system ***failure*** transparent means that a user does not notice that a resource (he has possibly never heard of) fails to work properly, and that the system subsequently recovers from that failure.
- Masking failures is one of the hardest issues in distributed systems
- The main difficulty in masking failures lies in the inability to distinguish between a dead resource and a painfully slow resource.
- For example, when contacting a busy Web server, a browser will eventually time out and report that the Web page is unavailable.
- At that point, the user cannot conclude that the server is really down.

1.2.3 Openness

- Another important goal of distributed systems is openness.
- An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.
- For example, in computer networks, standard rules govern the format, contents, and meaning of messages sent and received.
- An open distributed system should also be extensible.
- For example, in an extensible system, it should be relatively easy to add parts that run on a different operating system. or even to replace an entire file system.
- As many of us know from daily practice, attaining such flexibility is easier said than done.

1.2.4 Scalability

- Scalability of a system can be measured along at least three different dimensions (Neuman, 1994).
- *First*, a system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system.
- *Second*, a geographically scalable system is the one in which the users and resources may lie far apart.
- *Third*, a system can be administratively scalable, means that it can still be easy to manage, even if it spans many independent administrative organizations.

1.2.5 Pitfalls

- Distributed systems differ from traditional software because components are dispersed across a network.
- Not taking this dispersion into account during design time is what makes so many systems needlessly complex and results in mistakes that need to be patched later on.
- Following false assumptions that everyone makes when developing a distributed application for the first time:

1. The network is reliable(consistently good in quality or performance).
2. The network is secure.
3. The network is homogeneous(of the same kind).
4. The topology(arrangement of a network, including its nodes and connecting lines) does not change.
5. Latency is zero(Network latency is an expression of how much time it takes for a packet of data to get from one designated point to another)
6. Bandwidth(the amount of data that can be transmitted in a fixed amount of time) is infinite.
7. Transport cost is zero.
8. There is one administrator

1.3 TYPES OF DISTRIBUTED SYSTEMS

1.3.1 Distributed Computing Systems

- An important class of distributed systems is the one used for high-performance computing tasks.
- One can make a distinction between two subgroups :

  a. ***cluster computing s/m s***

  b. grid computing s/m s

- In ***cluster computing*** the underlying hardware consists of a collection of similar workstations or PCs, closely connected by means of a high speed local-area network. In addition, each node runs the same operating system.

- The ***grid computing*** sub group consists of distributed systems that are often constructed as a federation of computer systems, where each system may fall under a different administrative domain, and may be very different when it comes to hardware, software, and deployed network technology.

a. Cluster computing systems :

- Cluster computing systems became popular when the price/performance ratio of personal computers and workstations improved.
- At a certain point, it became financially and technically attractive to build a supercomputer using off-the-shelf technology by simply hooking up a collection of relatively simple computers in a high-speed network.
- In virtually all cases, cluster computing is used for parallel programming in which a single (compute intensive) program is run in parallel on multiple machines

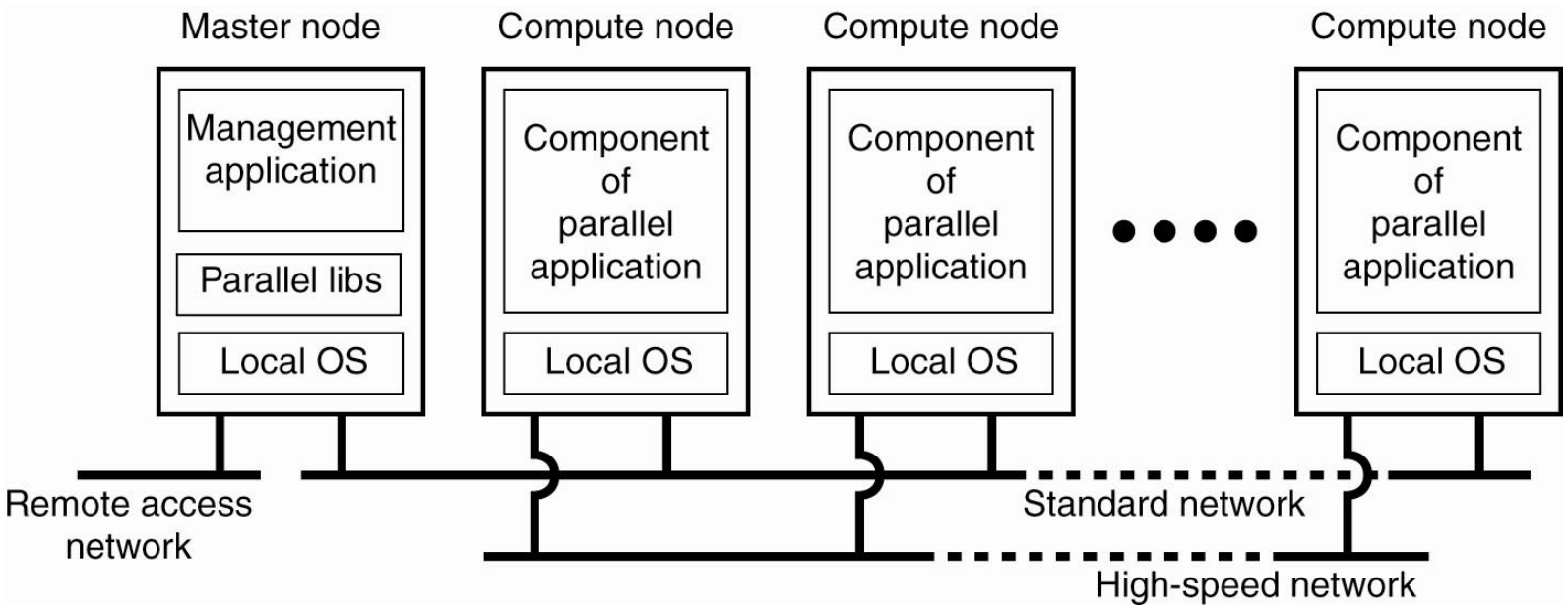


Figure 1-6. An example of a cluster computing system

- Each cluster consists of a collection of compute nodes that are controlled and accessed by means of a single master node.
- The master typically handles the allocation of nodes to a particular parallel program, maintains a batch queue of submitted jobs, and provides an interface for the users of the system
- The master actually runs the middleware needed for the execution of programs and management of the cluster, while the compute nodes often need nothing else but a standard operating system.
- An important part of this middleware is formed by the libraries for executing parallel programs

- Many of these libraries effectively provide only advanced message-based communication facilities, but are not capable of handling faulty processes, security, etc.

b. Grid Computing Systems

- ***A characteristic feature of cluster computing is its homogeneity***.
- In most cases, the computers in a cluster are largely the same, they all have the same operating system, and are all connected through the same network.
- In contrast, ***grid computing systems have a high degree of heterogeneity*** : no assumptions are made concerning hardware, operating systems, networks, administrative domains, security policies, etc.

- A key issue in a grid computing system is that resources from different organizations are brought together to allow the collaboration of a group of people or institutions.
- Such a collaboration is realized in the form of a virtual organization.
- The people belonging to the same virtual organization have access rights to the resources that are provided to that organization.
- Typically, resources consist of compute servers (including supercomputers, possibly implemented as cluster computers), storage facilities, and databases.

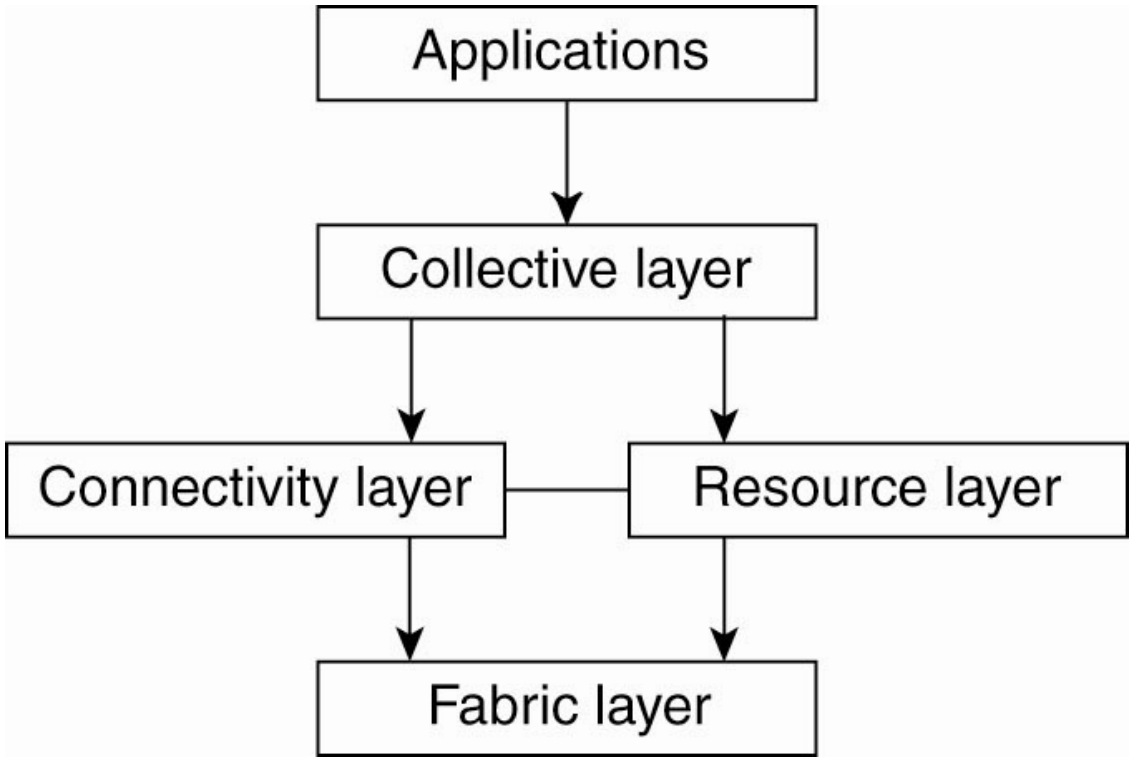


Figure 1-7. A layered architecture for grid computing systems

- The architecture consists of four layers :
- The lowest ***fabric layer*** provides interfaces to local resources at a specific site.
- Note that these interfaces are tailored to allow sharing of resources within a virtual organization
- The ***connectivity layer*** consists of communication protocols for supporting grid transactions that span the usage of multiple resources.
- For example, protocols are needed to transfer data between resources, or to simply access a resource from a remote location.
- In addition, the connectivity layer will contain security protocols to authenticate users and resources.

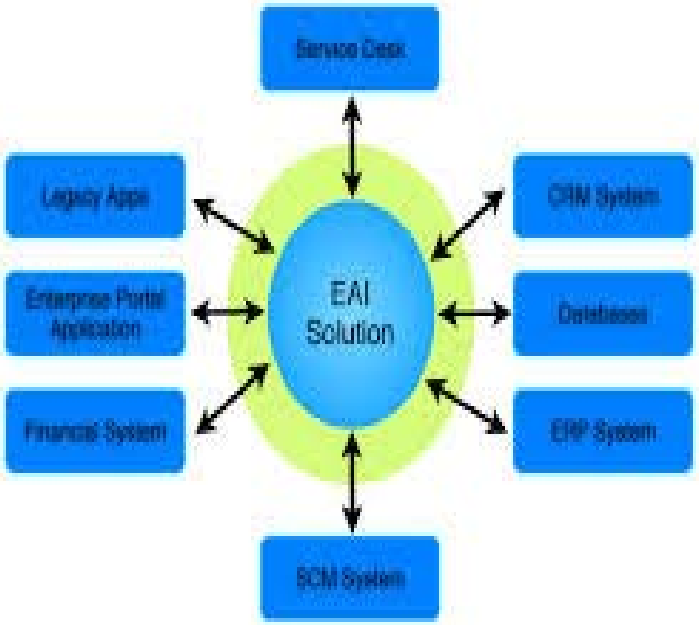
- The ***resource layer*** is responsible for managing a single resource.
- It uses the functions provided by the connectivity layer and calls directly the interfaces made available by the fabric layer.
- For example, this layer will offer functions for obtaining configuration information on a specific resource, or, in general, to perform specific operations such as creating a process or reading data
- ***Collective layer*** deals with handling access to multiple resources and typically consists of services for resource discovery, allocation and scheduling of tasks onto multiple resources, data replication, and so on.

- Unlike the connectivity and resource layer, which consist of a relatively small, standard collection of protocols, the *collective layer* may consist of many different protocols for many different purposes, reflecting the broad spectrum of services it may offer to a virtual organization.
- ***Application layer*** consists of the applications that operate within a virtual organization and which make use of the grid computing environment.
- The collective, connectivity, and resource layer form the heart of what could be called a grid middleware layer.
- These layers jointly provide access to and management of resources that are potentially dispersed across multiple sites.

1.3.2 Distributed Information Systems

- A set of information systems physically distributed over multiple sites, which are connected with some kind of communication network.
- *We can distinguish several levels at which integration took place.*
- In many cases, a networked application simply consisted of a server running that application (often including a database) and making it available to remote programs, called clients.
- Such clients could send a request to the server for executing a specific operation, after which a response would be sent back.

- ***Integration at the lowest level*** would allow clients to wrap a number of requests, possibly for different servers, into a single larger request and have it executed as a distributed transaction.
- The key idea was that all, or none of the requests would be executed.
- As applications became more sophisticated and were gradually separated into independent components (notably distinguishing database components from processing components), it became clear that ***integration should also take place by letting applications communicate directly with each other.***
- This has now led to a huge industry that concentrates on **enterprise application integration** (EAI).


Service Desk
Legacy Apps
CRM System
Enterprise Portal Application
EAI Solution
Database
Financial System
ERP System
SCM System

- Two forms of distributed systems :

a. Transaction Processing Systems

- Programming using transactions, requires special primitives(segment of code that can be used to build more sophisticated program) that must either be supplied by the underlying distributed system or by the language runtime system.
- Typical examples of transaction primitives are shown in **Fig. 1-8**.
- The exact list of primitives depends on what kinds of objects are being used in the transaction
- In a mail system, there might be to send, receive, and forward mail. In an accounting system primitives, they might be quite different.
- READ and WRITE are typical examples

Primitive	Description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore the old values
READ	Read data from a file, a table, or otherwise
WRITE	Write data to a file, a table, or otherwise

Figure 1-8. Example primitives for transactions

This all-or-nothing property of transactions is one of the four characteristic properties that transactions have.
More specifically, transactions are: (**ACID** properties)

1. **Atomic** : To the outside world, the transaction happens indivisibly.
2. **Consistent** : The transaction does not violate system invariants.
3. **Isolated** : Concurrent transactions do not interfere with each other.
4. **Durable** : Once a transaction commits, the changes are permanent.

- The ***first key property*** exhibited by all transactions is that they are **atomic**.
- This property ensures that each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action.

- The ***second property*** says that they are **consistent**.
- What this means is that if the system has certain invariants(functions / properties) that must always hold, if they held before the transaction, they will hold afterward too.

- The ***third property*** says that transactions are **isolated** or serializable.
- What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions are sequentially in some(system dependent) order.

- The ***fourth property*** says that transactions are **durable**.
- It refers to the fact that once a transaction commits, no matter what happens, the transaction goes forward and the results become permanent.
- No failure after the commit can undo the results or cause them to be lost.

- So far, transactions have been defined on a single database.
- A ***nested(distributed) transaction*** is constructed from a number of sub transactions, as shown in **Fig. 1-9**.

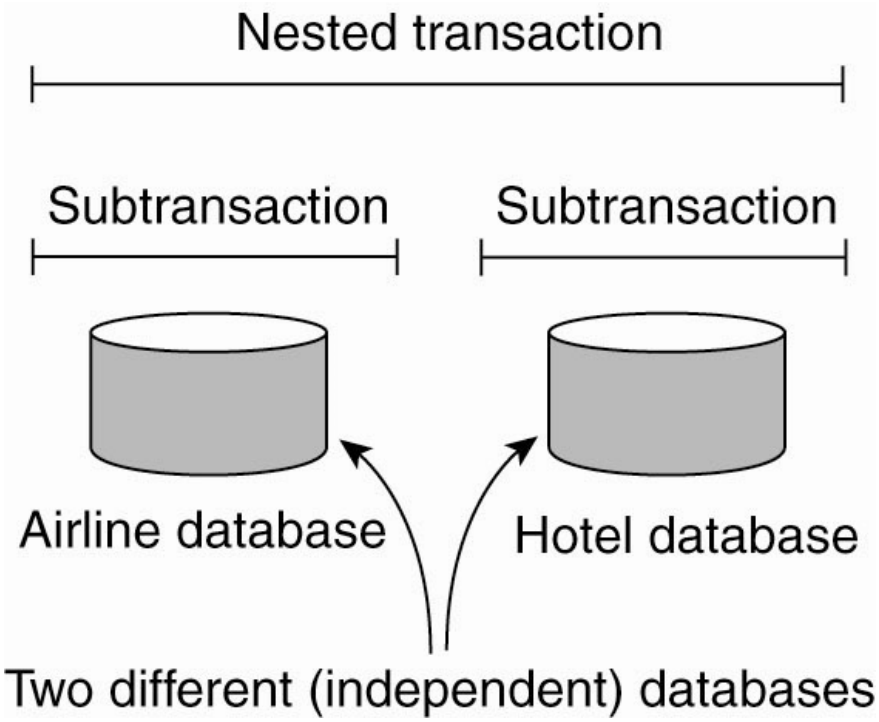


Figure 1-9. A nested transaction

- In the early days of enterprise middleware systems, *the component that handled distributed (or nested) transactions* formed the core for integrating applications at the server or database level.
- This component was called a ***transaction processing monitor or TP monitor***
- ***Its main task was to allow an application*** to access multiple server/databases by offering it a transactional programming model, as shown in **Fig. 1-10**.

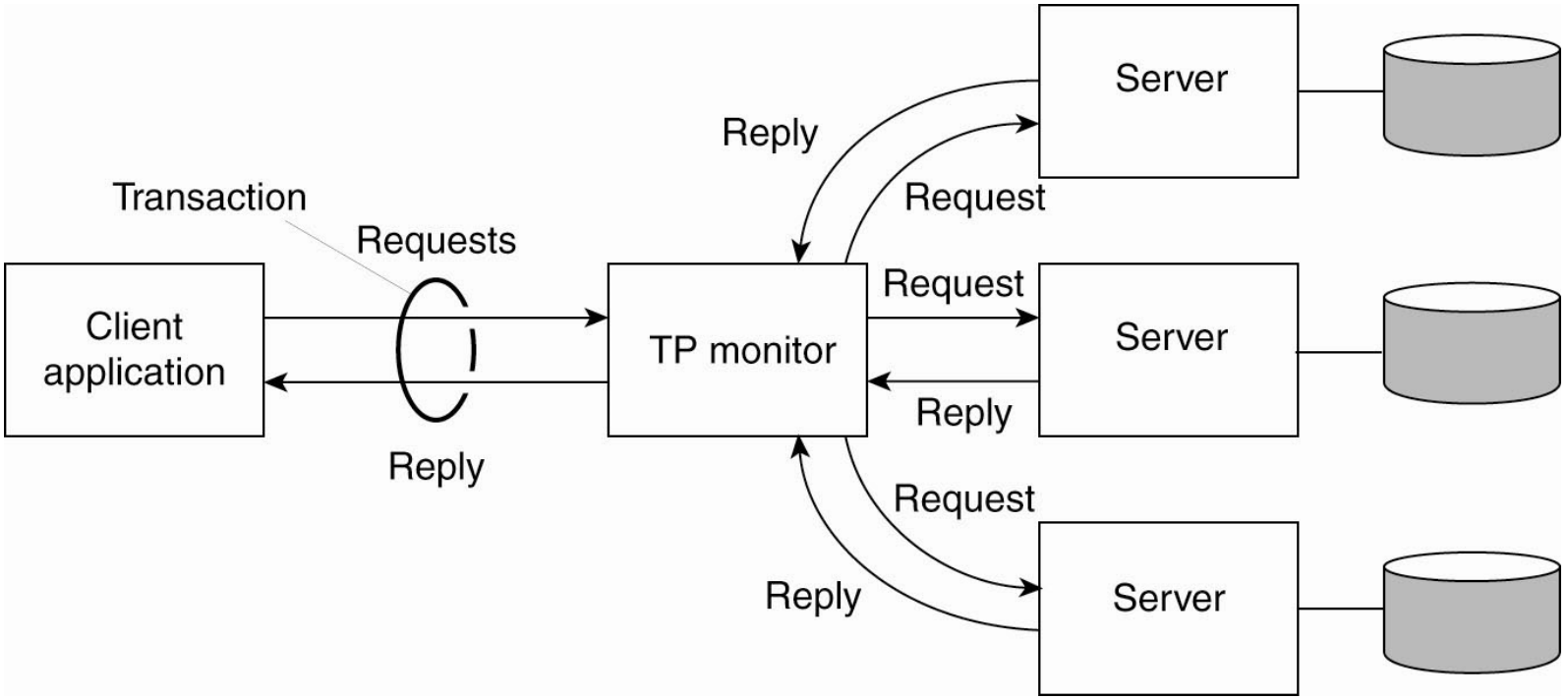


Figure 1-10. The role of a TP monitor in distributed systems.

b. Enterprise Application Integration

- ***Application components should be able to communicate directly with each other*** *and* ***not merely by means of the request/reply behaviour*** that was supported by transaction processing systems.
- The main idea was that existing applications could directly exchange information, as shown in **Fig. 1-11**.

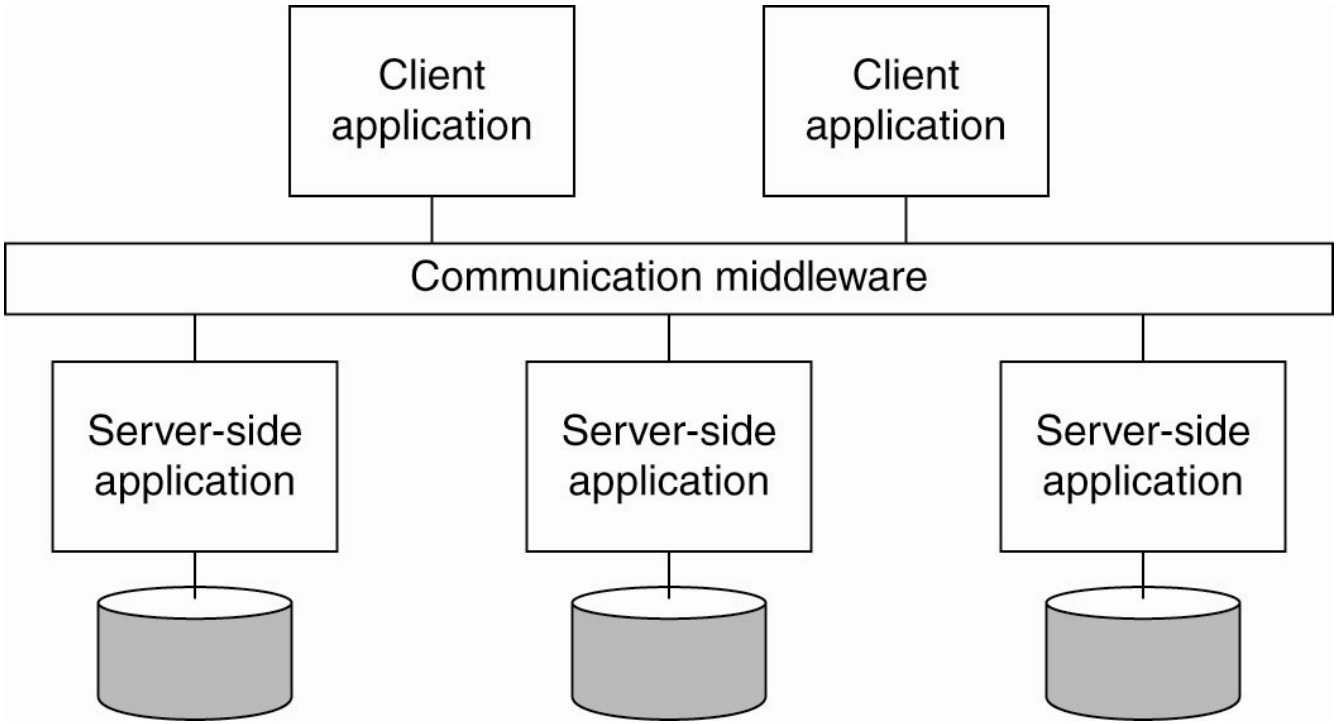


Figure 1-11. Middleware as a communication facilitator in enterprise application integration.

- Several ***types of communication middleware*** exist :
- With ***remote procedure calls (RPC),*** an application component can effectively send a request to another application component by doing a local procedure call, which results in the request being packaged as a message and sent to the callee.
- Likewise, the result will be sent back and returned to the application as the result of the procedure call.

- As the popularity of object technology increased, techniques were developed to allow calls to remote objects, leading to what is known as ***remote method invocations (RMI-*** is a way in which objects on different computers can interact in a distributed network.***).***
- An RMI is essentially the same as an RPC, except that it operates on objects instead of applications.
- **RPC and RMI have the disadvantage that the caller and callee both need to be up and running at the time of communication.**
- In addition, they need to know exactly how to refer to each other.
- This tight coupling is often experienced as a serious drawback, and has led to what is known as **message-oriented middleware**, or simply ***MOM***.

- In this case, applications simply send messages to logical contact points, often described by means of a subject.
- Likewise, applications can indicate their interest for a specific type of message, after which the communication middleware will take care that those messages are delivered to those applications.
- These so-called publish/subscribe systems form an important and expanding class of distributed systems.

1.3.3 Distributed Pervasive(present everywhere) Systems

- The distributed systems we have been discussing so far are largely characterized by their stability : nodes are fixed and have a more or less permanent and high-quality connection to a network.
- To a certain extent, this stability has been realized through the various techniques which aim at achieving distribution transparency.
- However, matters have become very different with the introduction of mobile and embedded computing devices.

- We are now confronted with distributed systems in which instability is the default behaviour.
- The devices in these, what we refer to as distributed pervasive systems, are often characterized by being small, battery-powered, mobile, and having only a wireless connection, although not all these characteristics apply to all devices.
- As its name suggests, a distributed pervasive system is part of our surroundings (and as such, is generally inherently distributed).

- An important feature is the general lack of human administrative control.
- At best, devices can be configured by their owners, but otherwise they need to automatically discover their environment and "nestle in" as best as possible.
- This nestling in has been made more precise by formulating the following three requirements for pervasive applications:

    1. Embrace contextual changes.
    2. Encourage adhoc composition.
    3. Recognize sharing as the default.

- ***Embracing contextual changes means that*** a device must be continuously be aware of the fact that its environment may change all the time.
- One of the simplest changes is discovering that a network is no longer available, for example, because a user is moving between base stations.
- In such a case, the application should react, possibly by automatically connecting to another network, or taking other appropriate actions.

- ***Encouraging adhoc(***Ad hoc generally signifies a solution designed for a specific problem or task, **non**-generalizable, and not intended to be able to be adapted to other purposes.) ***composition refers to the*** fact that many devices in pervasive systems will be used in very different ways by different users.
- As a result, it should be easy to configure the suite of applications running on a device, either by the user or through automated (but controlled) interposition.
- One very important aspect of pervasive systems is that devices generally join the system in order to access (and possibly provide) information.
- This calls for means to easily read, store, manage, and share information.

- In light of the intermittent and changing connectivity of devices, the space where accessible information resides will most likely change all the time.
- in the presence of mobility, devices should support easy and application-dependent adaptation to their local environment.
- They should be able to efficiently discover services and react accordingly.
- It should be clear from these requirements that distribution transparency is not really in place in pervasive systems.
- In fact, distribution of data, processes, and control is inherent to these systems, for which reason it may be better just to simply expose it rather than trying to hide it

- Concrete examples of pervasive systems :

a. Home Systems

- An increasingly popular type of pervasive system, but which may perhaps be the least constrained, are systems built around home networks.
- These systems generally consist of one or more personal computers, but more importantly integrate typical consumer electronics such as TVs, audio and video equipment. gaming devices, (smart) phones, PDAs, and other personal wearables into a single system.
- In addition, we can expect that all kinds of devices such as kitchen appliances, surveillance cameras, clocks, controllers for lighting, and so on, will all be hooked up into a single distributed system.

- From a system's perspective ***there are several challenges that need to be addressed before pervasive home systems become reality.***
- *An important one is that such a system should be completely self-configuring and self-managing.*
- It cannot be expected that end users are willing and able to keep a distributed home system up and running if its components are prone to errors .
- Much has already been accomplished through the Universal Plug and Play (UPnP) standards by which devices automatically obtain IP addresses, can discover each other, etc.

- However, more is needed.
- For example, it is unclear how software and firmware in devices can be easily updated without manual intervention, or when updates do take place, that compatibility with other devices is not violated.
- *Another pressing issue is managing what is known as a "personal space."*
- Recognizing that a home system consists of many shared as well as personal devices, and that the data in a home system is also subject to sharing restrictions, much attention is paid to realizing such personal spaces.

- For example, part of Alice's personal space may consist of her agenda, family photo's, a diary, music and videos that she bought, etc.
- These personal assets should be stored in such a way that Alice has access to them whenever appropriate.
- Moreover, parts of this personal space should be (temporarily) accessible to others, for example. when she needs to make a business appointment.

- However, problems may be alleviated(to make problems less severe) due to the rapid increase in the capacity of hard disks, along with a decrease in their size.
- However, having enough storage does not solve the problem of managing personal spaces.
- Being able to store huge amounts of data shifts the problem to storing relevant data and being able to find it later.

b. Electronic Health Care Systems

c. Sensor Networks

PROCESSES

https://www.youtube.com/watch?v=exbKr6fnoUw

https://www.youtube.com/watch?v=fxMOi7BlTyM

https://www.youtube.com/watch?v=nKdQgn_udtA

Process vs Threads

- A **process** *is an executing instance of an application.*
- What does that mean?
- Well, for example, when you double-click the Microsoft Word icon, you start a process that runs Word.
- A **thread** *is a path of execution within a process.*
- Also, a process can contain multiple threads.
- When you start Word, the operating system creates a process and begins executing the primary thread of that process.

Multi Threading

- Threads, of course, allow for multi-threading.
- *A common example of the advantage of multithreading* is the fact that you can have a word processor that prints a document using a background thread, but at the same time another thread is running that accepts user input, so that you can type up a new document.
- *If we were dealing with an application that uses only one thread*, then the application would only be able to do one thing at a time – so printing and responding to user input at the same time would not be possible in a *single threaded application.*

3.1 THREADS

3.1.1 Introduction to Threads

- 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.
- 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 behaviour.
- In other words, the fact that multiple processes may be concurrently sharing the same CPU and other hardware resources is made transparent.
- Apart from saving the ***CPU*** *context* (which consists of ***register values***, program counter, stack pointer, etc.), 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).

- Like a process, a **thread** *executes its own piece of code, independently from other threads*.
- However, *in contrast to processes*, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation.
- Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads.
- In particular, **a thread context often consists of nothing more than the CPU context, along with some other information for thread management**.

- **For example**, a thread system may keep track of the fact that a thread is currently blocked on a mutex (*A **mutex** is a binary variable whose purpose is to provide locking mechanism. It is used to provide mutual exclusion to a section of code, means only one process can work on a particular code section at a time*), so as not to select it for execution.
- Information that is not strictly necessary to manage multiple threads is generally ignored

3.1.2 Threads in Distributed Systems

- **Multithreaded Clients**
- 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 thus, need 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 setup 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.

- 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(of server), 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, as we have seen, *the main use of multithreading in distributed systems is found at the server side.*
- *Multithreading makes it much easier to develop servers that exploit parallelism to attain high performance*
- *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 **Fig. 3-3**.
- *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.

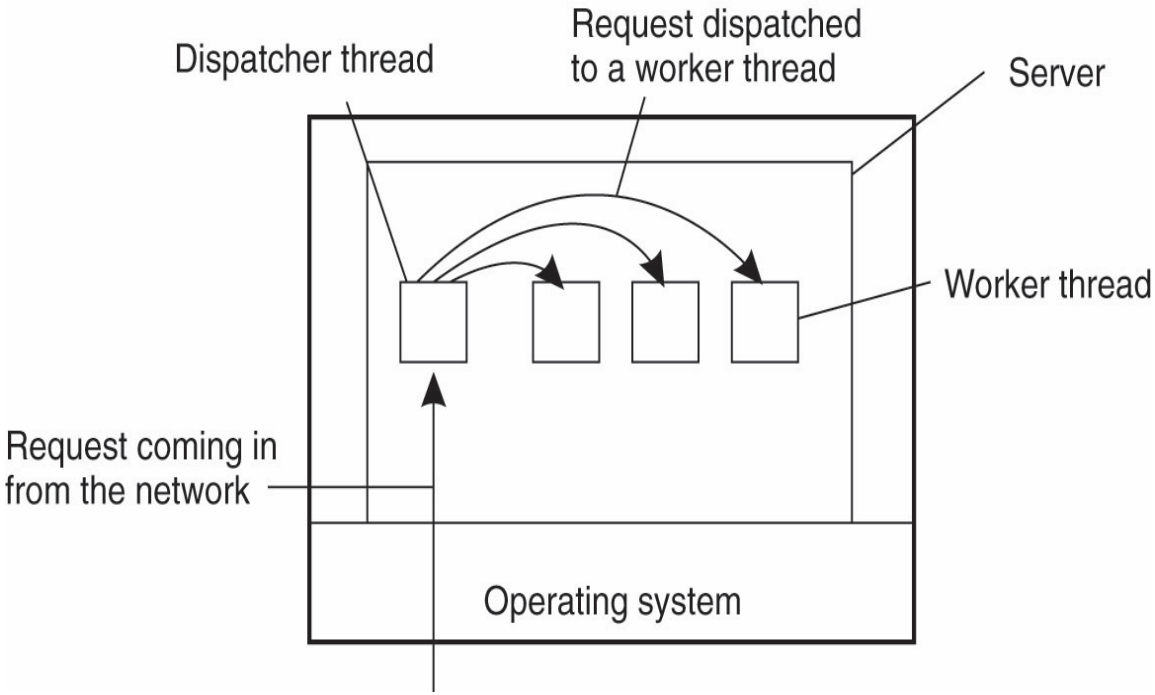


Figure 3-3 : A multithreaded server organized in a dispatcher/worker model.

- *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*.
- ***So far we have seen two possible designs*** : (a.)a multithreaded file server and a (b.)single-threaded file server.

- **A third possibility is** *to run the server as a (c.)*big finite-state machine.
- *When a request comes in,* ***the one and only thread*** *examines it.*
- *If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.*
- *However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message.*
- *The next message may either be a request for new work or a reply from the disk about a previous operation.*

- If it is new work, that work is started.
- If it is a reply from the disk, the relevant information is fetched from the table and the reply is processed and subsequently sent to the client.
- In this scheme, the server will have to make use of non blocking calls to send and receive.
- The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

- (a.)**Threads** make it possible to retain the idea of sequential processes that make blocking system calls (e.g., an RPC to talk to the disk) and still achieve parallelism.
- Blocking system calls make programming easier and parallelism improves performance.
- (b.)The **single-threaded server** retains the ease and simplicity of blocking system calls, but gives up some amount of performance.
- (c.)The **finite-state machine** approach achieves high performance through parallelism, but uses non blocking calls, thus is hard to program.
- These models are summarized in **Fig. 3-4.**

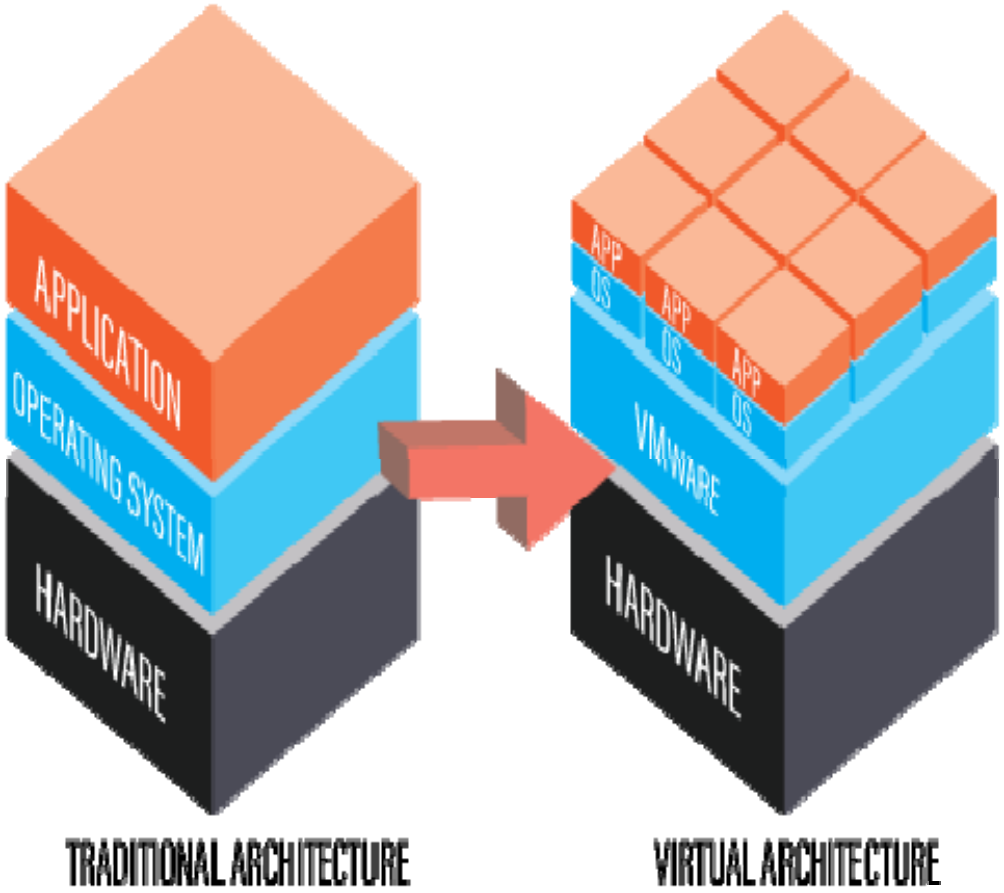
Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Figure 3-4 : Three ways to construct a server.

//extra

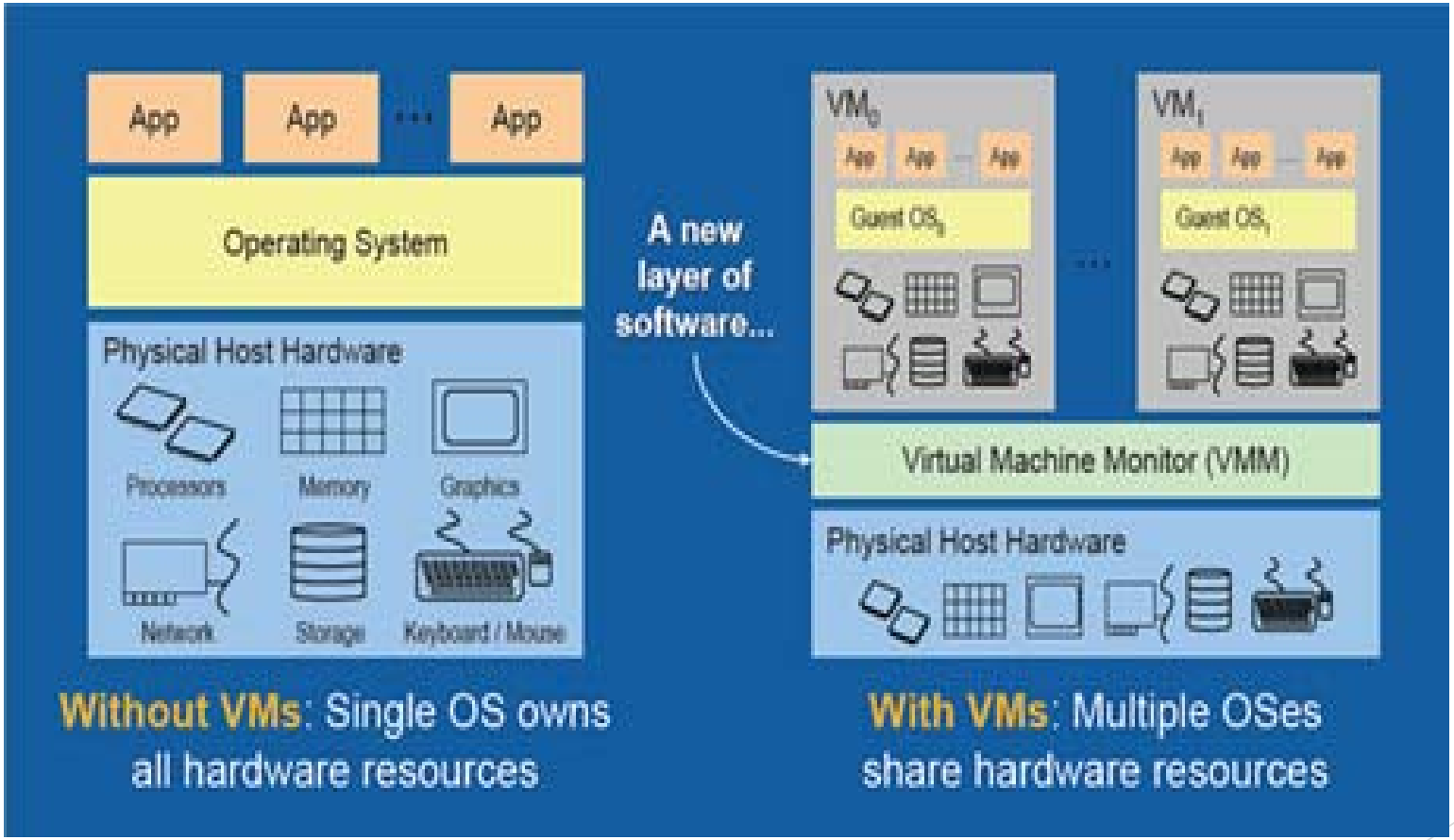
Virtualization- https://opensource.com/resources/virtualization

- Virtualization is the process of running a virtual instance of a computer system in a layer abstracted from the actual hardware.
- Most commonly, it refers to running multiple operating systems on a computer system simultaneously.
- To the applications running on top of the virtualized machine, it can appear as if they are on their own dedicated machine, where the operating system, libraries, and other programs are unique to the guest virtualized system and unconnected to the host operating system which sits below it.


APPLICATION
OPERATING SYSTEM
HARDWARE
APP
OS
APP
OS
APP
OS
VMWARE
HARDWARE
TRADITIONAL ARCHITECTURE
VIRTUAL ARCHITECTURE

//extra

//extra


App
App
App
Operating System
Physical Host Hardware
Processors
Memory
Graphics
Network
Storage
Keyboard / Mouse
A new layer of software...
Virtual Machine Monitor (VMM)
Physical Host Hardware
Without VMs: Single OS owns all hardware resources
With VMs: Multiple OSes share hardware resources

//extra

- A ***hypervisor*** is a function which abstracts -- isolates -- operating systems and applications from the underlying computer hardware.
- This abstraction allows the underlying host machine hardware to independently operate one or more virtual machines as guests, allowing multiple guest VMs to effectively share the system's physical compute resources, such as processor cycles, memory space, network bandwidth and so on.
- *A hypervisor is sometimes also called a* ***virtual machine monitor.***

https://searchservervirtualization.techtarget.com/definition/hypervisor

//extra

system call and library function

- Examples of systems calls
  - Read() / write ()
  - Fork()/ exec ()
- Example of library functions
  - Date/time
  - Strcpy/atoi/printf

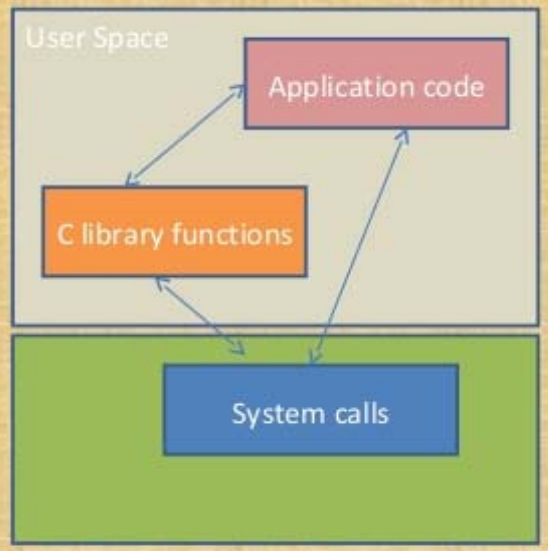


Figure not to study,only for reference

API : //extra

Imagine you're sitting at a table in a restaurant with a menu of choices to order from. The kitchen is the part of the "system" that will prepare your order. What is missing is the critical link to communicate your order to the kitchen and deliver your food back to your table. That's where the waiter or API comes in. The **waiter is the messenger – or API** – that takes your request or order and tells the kitchen – the system – what to do. Then the waiter delivers the response back to you; in this case, it is the food.

https://www.mulesoft.com/resources/api/what-is-an-api

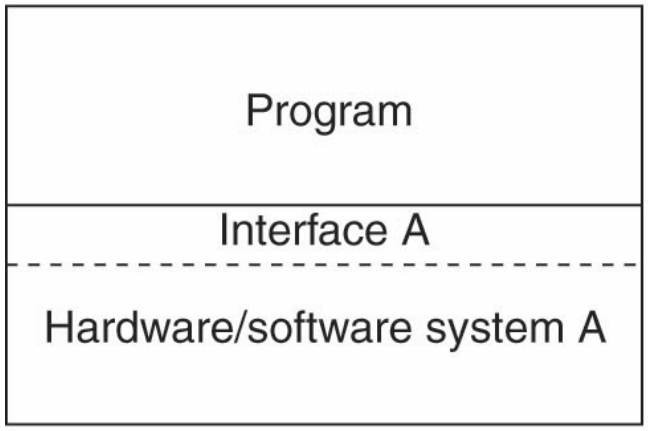
https://www.vmware.com/in/solutions/virtualization.html

3.2 VIRTUALIZATION

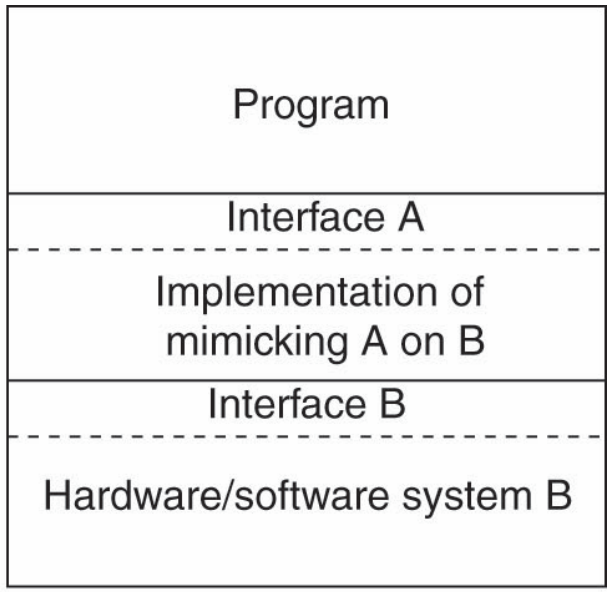
3.2.1 The Role of Virtualization in Distributed Systems

- Every (distributed) computer system offers a programming interface to higher level software, as shown in **Fig. 3-5(a).**
- 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

- Virtualization deals with extending or replacing an existing interface so as to mimic the behaviour of another system, as shown in **Fig.3-5(b).**



(a)



(b)

- Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

3.2.2 Architectures of Virtual Machines

- There are many different ways in which virtualization can be realized in practice.
- To understand the differences in virtualization, it is important to realize that *computer systems generally offer four different types of interfaces, at four different levels* :

1. An interface between the hardware and software, consisting of *machine instructions* that can be invoked by any program.

2. An interface between the hardware and software, consisting of machine instructions that can be invoked only by *privileged programs*, such as an operating system.

3. An interface consisting of *system calls* as offered by an operating system.

4. An interface consisting of library calls, generally forming what is known as an *application programming interface* (API). In many cases, the aforementioned(previously mentioned) system calls are hidden by an API.

- These different types are shown in **Fig. 3-6.**
- The essence of virtualization is to mimic the behaviour of these interfaces.

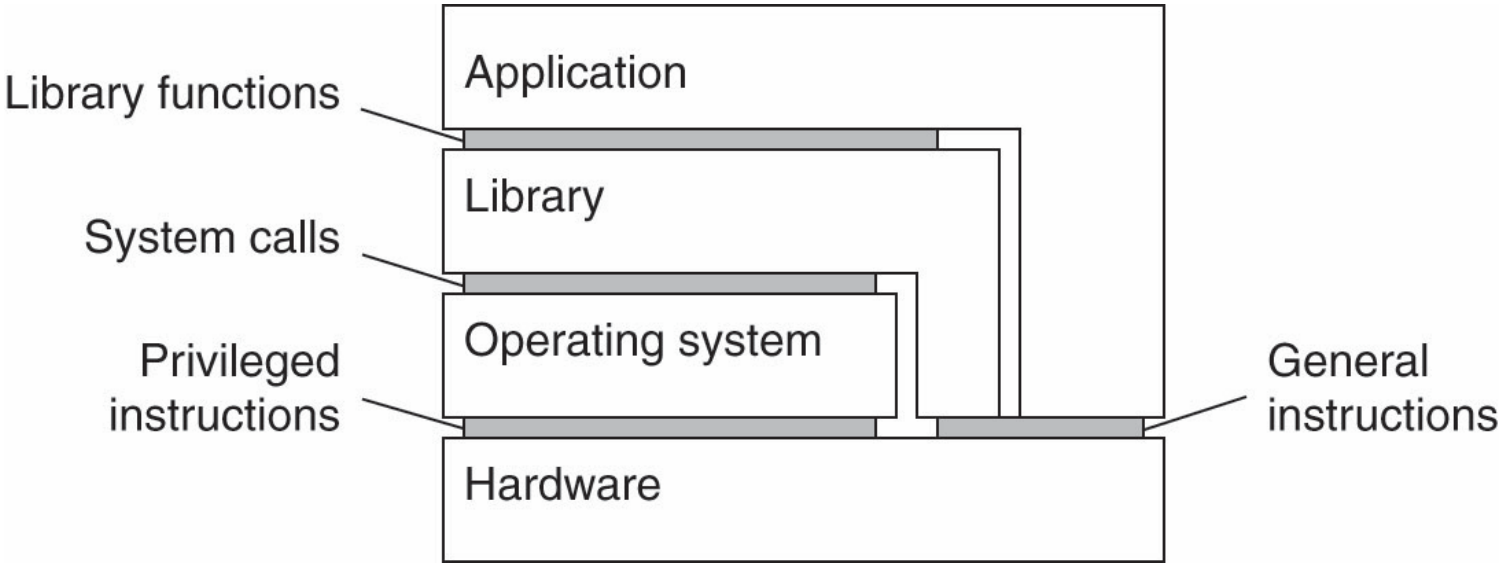


Figure 3-6. Various interfaces offered by computer systems.

- *Virtualization can take place in two different ways*.
- *First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications*.
- Instructions can be interpreted (as is the case for the Java runtime environment), but could also be emulated as is done for running Windows applications on UNIX platforms.
- This type of virtualization leads to a *process virtual machine*, stressing that virtualization is done essentially *only for a single process*

- *An alternative approach toward virtualization is* to provide a system that is essentially implemented as a layer completely shielding the original hardware , but offering the complete instruction set of that same (or other hardware) as an interface.
- As a result, it is now possible to have multiple, and different operating systems run independently and concurrently on the same platform.
- The layer is generally referred to as a ***virtual machine monitor (VMM).***
- *Typical examples of this approach are VMware and Xen*
- These two different approaches are shown in **Fig. 3-7**.

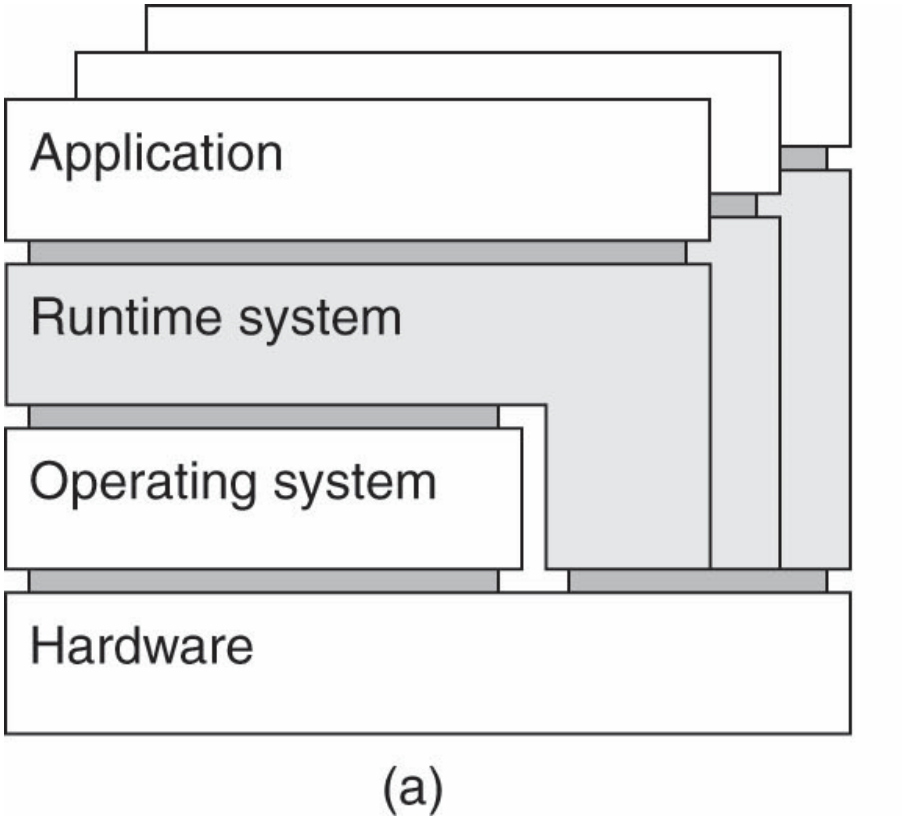


Figure 3-7. (a) A process virtual machine, with multiple instances of(application, runtime) combinations.

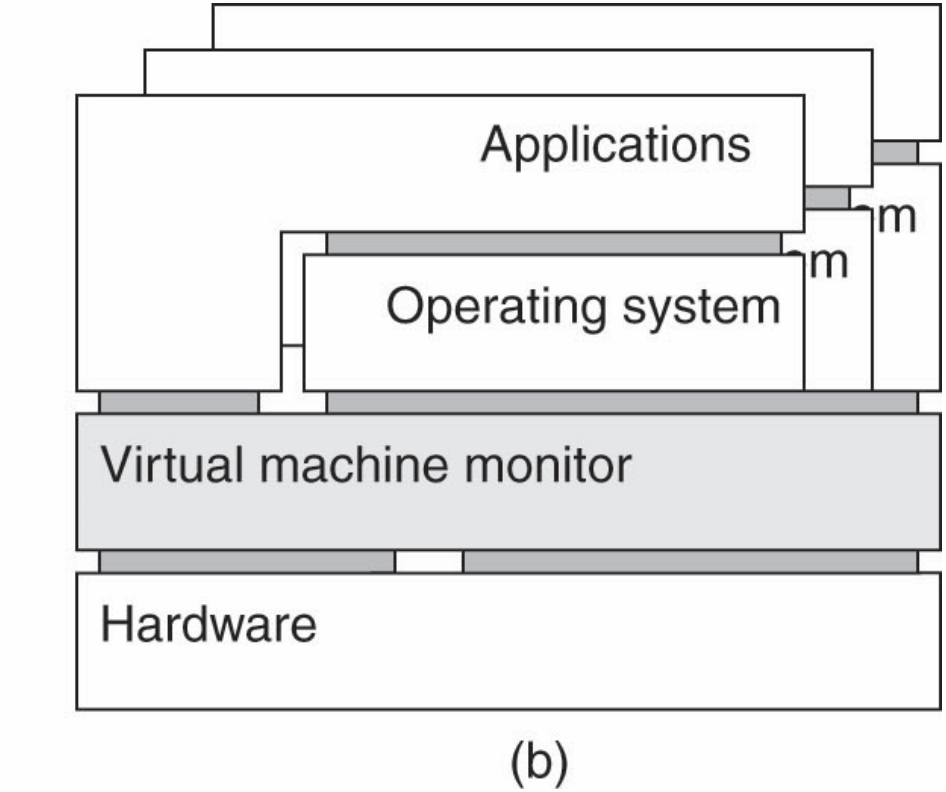


Figure 3-7. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.

- ***VMMs will become increasingly important*** in the context of reliability and security for (distributed) systems.
- As they allow for the isolation of a complete application and its environment, a failure caused by an error or security attack need no longer affect a complete machine.
- In addition, as we also mentioned before, portability is greatly improved as VMMs provide a further decoupling between hardware and software, allowing a complete environment to be moved from one machine to another.

3.3 CLIENTS

3.3.1 Networked User Interfaces

- 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 an agenda running on a user's PDA that needs to synchronize with a remote, possibly shared agenda.
- In this case, an application-level protocol will handle the synchronization, as shown in **Fig. 3-8(a).**

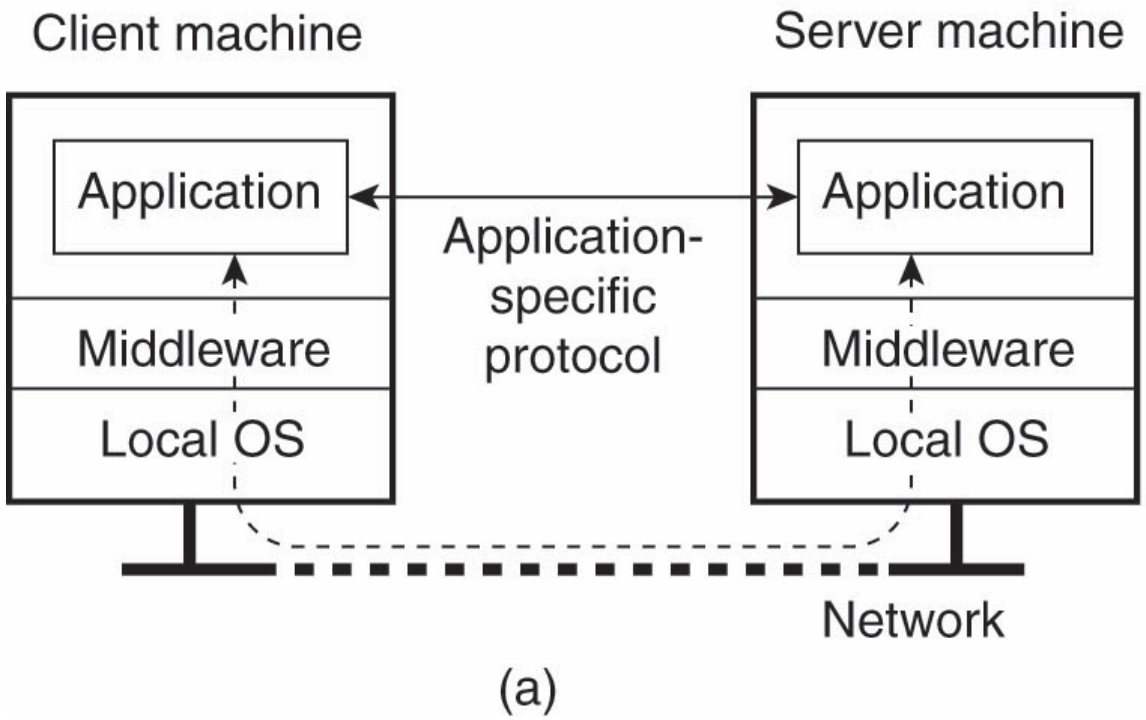


Figure 3-8. (a) A networked application with its own protocol.

- ***A second solution*** *is to provide direct access to remote services by only offering 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 **Fig. 3-8(b).**
- **In the case of networked user interfaces, everything is processed and stored at the server**.
- This *thin-client approach* is receiving more attention as Internet connectivity increases, and hand-held devices are becoming more sophisticated

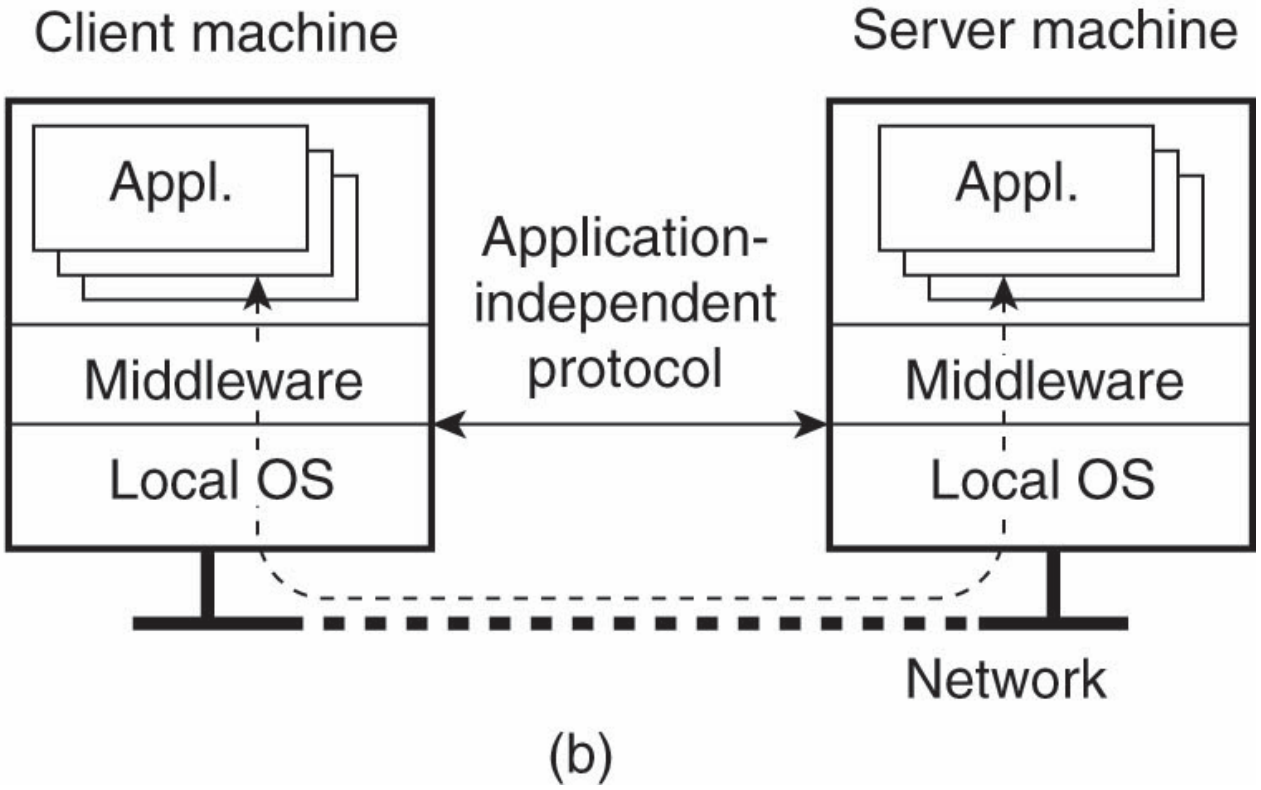


Figure 3-8. (b) A general solution to allow access to remote applications.

- *Example: The X Window System*
- *One of the oldest and still widely-used* ***networked user interfaces*** *is the* ***X Window system*****.**
- 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.
- 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 **Fig. 3-9**.

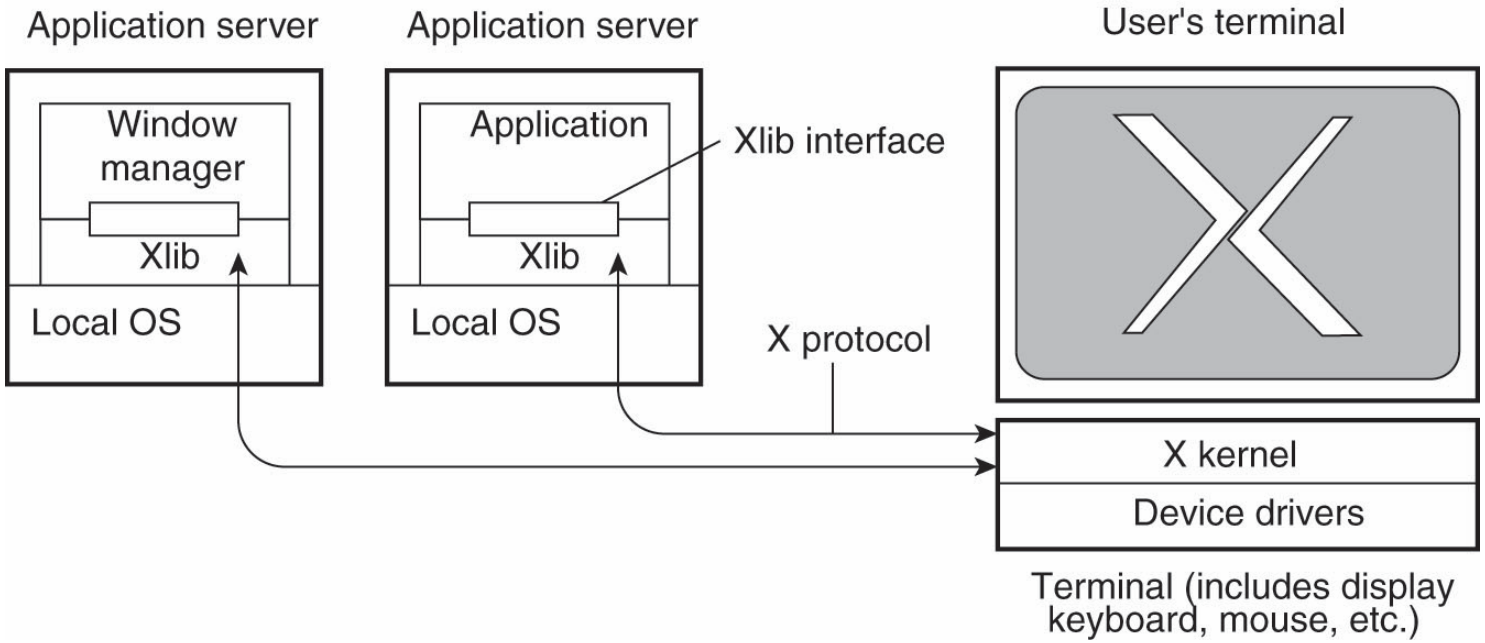


Figure 3-9. The basic organization of the XWindow System.

- 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 the X kernel.*
- For example, Xlib can send requests to the X kernel for creating or killing a window, setting colours, 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.

- Several applications can communicate at the same time with the X kernel.
- There is one specific application that is given special rights, known as the ***window manager***.
- This application can dictate the "look and feel" of the display as it appears to the user

- Thin-Client Network Computing

https://www.slideshare.net/VinodKumarVH/thin-client-17126209

https://techterms.com/definition/thinclient

https://searchnetworking.techtarget.com/definition/thin-client

https://www.techopedia.com/definition/462/thin-client

https://www.youtube.com/watch?v=3ssKCCYXueY

- 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.

- In THINC, display requests from the application are intercepted and translated into the lower level commands.
- By intercepting application requests, THINC can make use of application semantics to decide what combination of lower level commands can be used best.
- Translated commands are not immediately sent out to the display, but are instead queued.
- By batching several commands it is possible to aggregate display commands into a single one, leading to fewer messages.

- For example, when a new command for drawing in a particular region of the screen effectively overwrites what a previous (and still queued) command would have established, the latter need not be sent out to the display.
- Finally, instead of letting the display ask for refreshments, THINC always pushes updates as they come available.
- This push approach saves latency as there is no need for an update request to be sent out by the display.
- As it turns out, *the approach followed by THINC provides better overall performance*.

Compound Documents

- Many user interfaces allow applications to share a single graphical window, and to use that window to exchange data through user actions.
- Additional actions that can be performed by the user include what are generally called drag-and-drop operations, and in-place editing, respectively.

- A typical example of *drag-and-drop functionality* is moving an icon representing a file A to an icon representing a trash can, resulting in the file being deleted.
- In this case, the user interface will need to do more than just arrange icons on the display : it will have to pass the name of the file A to the application associated with the trash can as soon as A's icon has been moved above that of the trash can application

- *In-place editing* can best be illustrated by means of a document containing text and graphics. *Imagine that the document is being displayed within a standard word processor.*
- As soon as the user places the mouse above an image, the user interface passes that information to a drawing program to allow the user to modify the image.
- For example, the user may have rotated the image, which may effect the placement of the image in the document.
- The user interface therefore finds out what the new height and width of the image are, and passes this information to the word processor.
- The latter, in tum, can then automatically update the page layout of the document.

- A ***compound document***, which can be defined as a collection of documents, possibly of very different kinds (like text, images, spreadsheets, etc.), which are seamlessly integrated at the user-interface level.
- *A user interface that can handle compound documents hides the fact that different applications operate on different parts of the document*.
- To the user, all parts are integrated in a seamless way.
- When changing one part affects other parts, the user interface can take appropriate measures, for example, by notifying the relevant applications.

3.3.2 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 available at the server, but hides the possible differences in machine architectures, as well as the actual communication.

- There are different ways to handle *location, migration, and relocation transparency.*

  Using a convenient naming system is crucial.
- 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 **Fig. 3-10**.
- Client-side software can transparently collect all responses and pass a single response to the client application

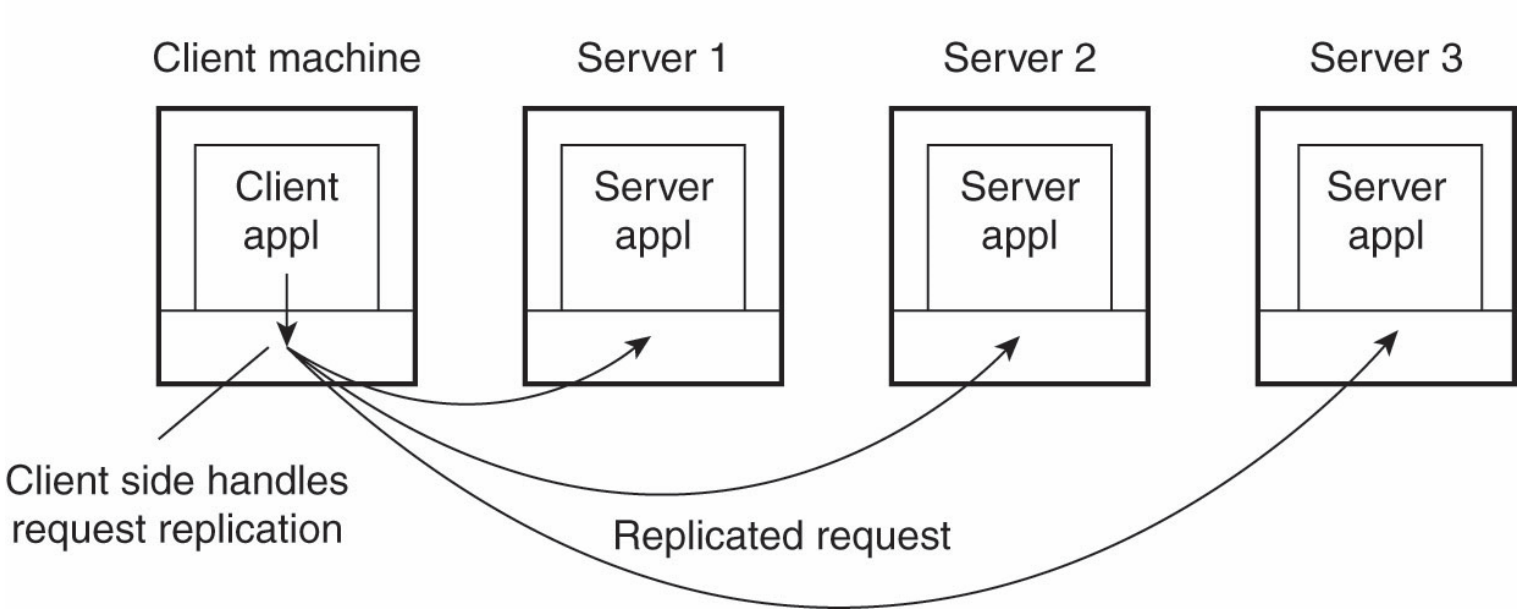


Figure 3-10. Transparent replication of a server using a client-side solution.

- Finally, consider *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.

3.4 SERVERS

3.4.1 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.
- **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.
- **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.
- These end points have been assigned by the Internet Assigned Numbers Authority (lANA)

- *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 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 endpoint 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 **Fig. 3-11(a).**

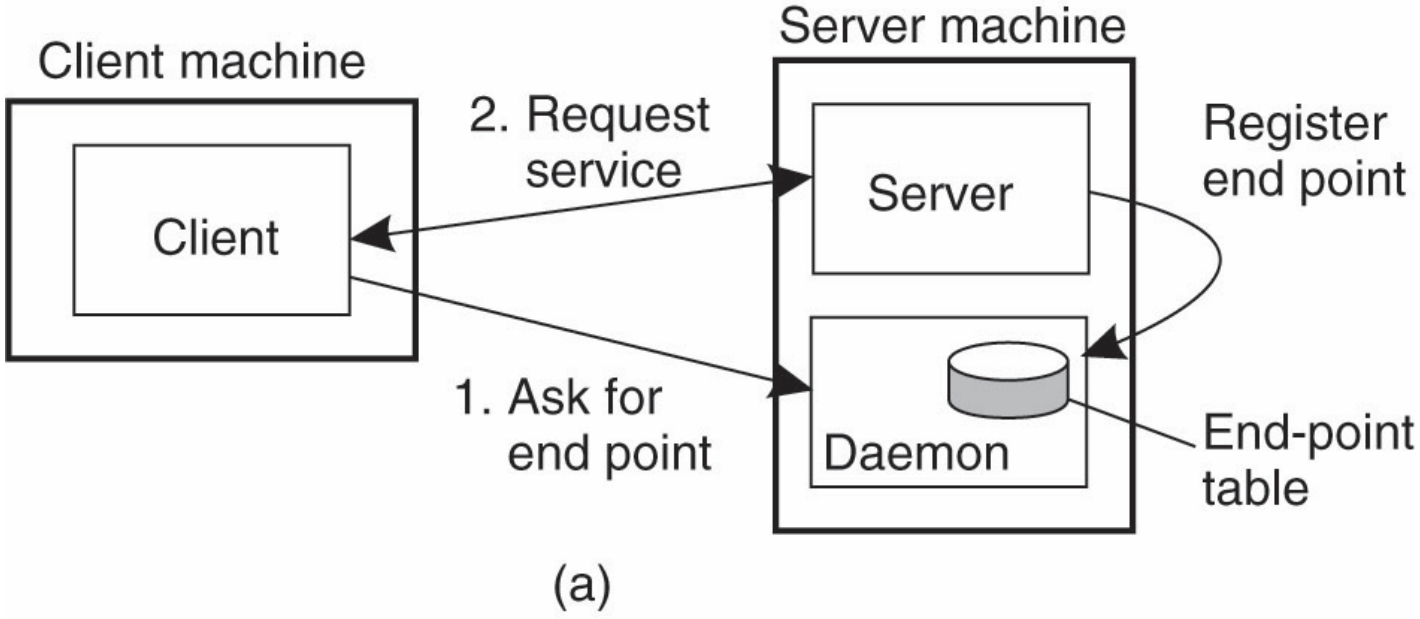


Figure 3-11 . (a) Client-to-server binding using a daemon.

- 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 **Fig.3-1l(b)**.

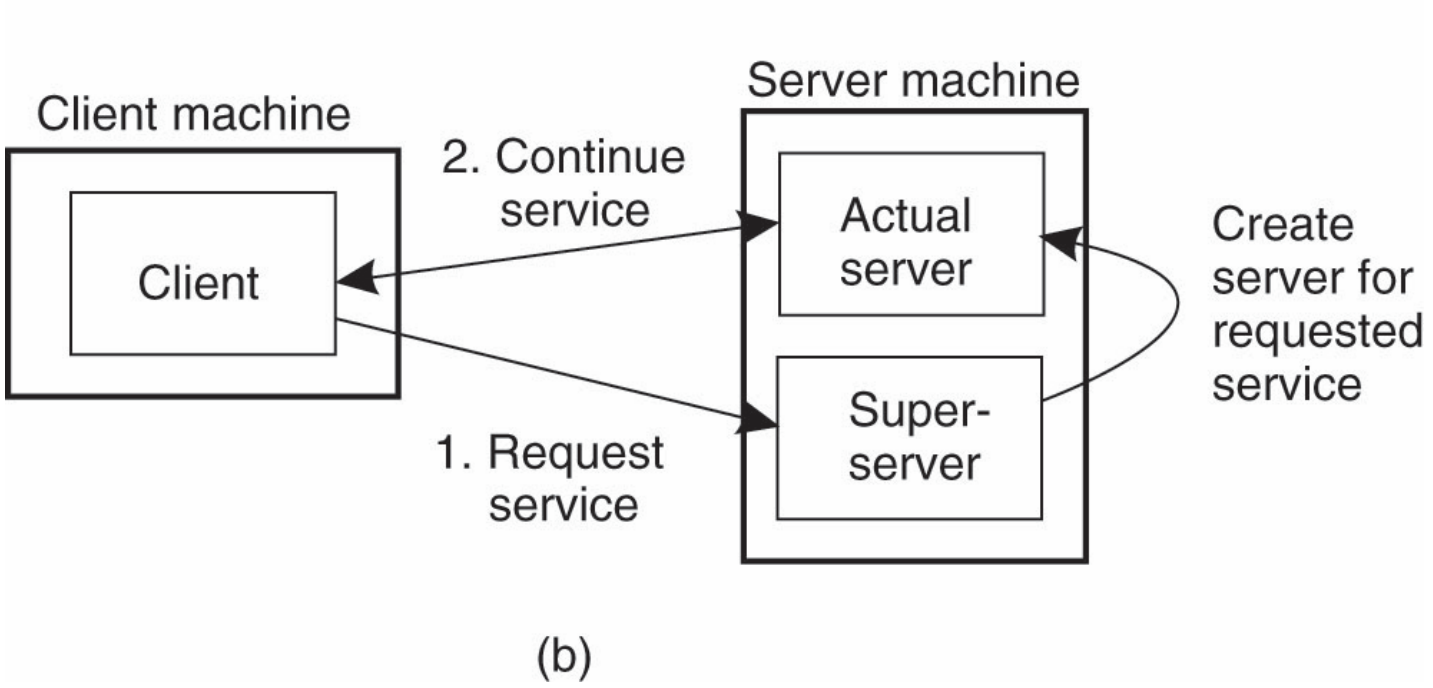


Figure 3-11. (b) Client-to-server binding using a super server.

- This is the approach taken, for example, with the inetd daemon in UNIX.
- Inetd listens to a number of well-known ports for Internet services.
- When a request comes in, the daemon forks a process to take further care of the request.
- That process will exit after it is finished.
- **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.

- *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.

- 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.
- When designing a server, the choice for a stateless or stateful design should not affect the services provided by the server.

3.4.2 Server Clusters

General Organization

- ***A server cluster is*** nothing else but **a collection of machines connected through a network, where each machine runs one or more servers**.
- The server clusters that we consider here, are the ones in which the machines are connected through a local-area network, often offering high bandwidth and low latency.
- In most cases, a server cluster is logically organized into three tiers, as shown in **Fig. 3-12**.
- ***The first tier*** consists of a (logical) switch through which client requests are routed.

- Such a switch can vary widely.
- For example, transport-layer switches accept incoming TCP connection requests and pass requests on to one of servers in the cluster.

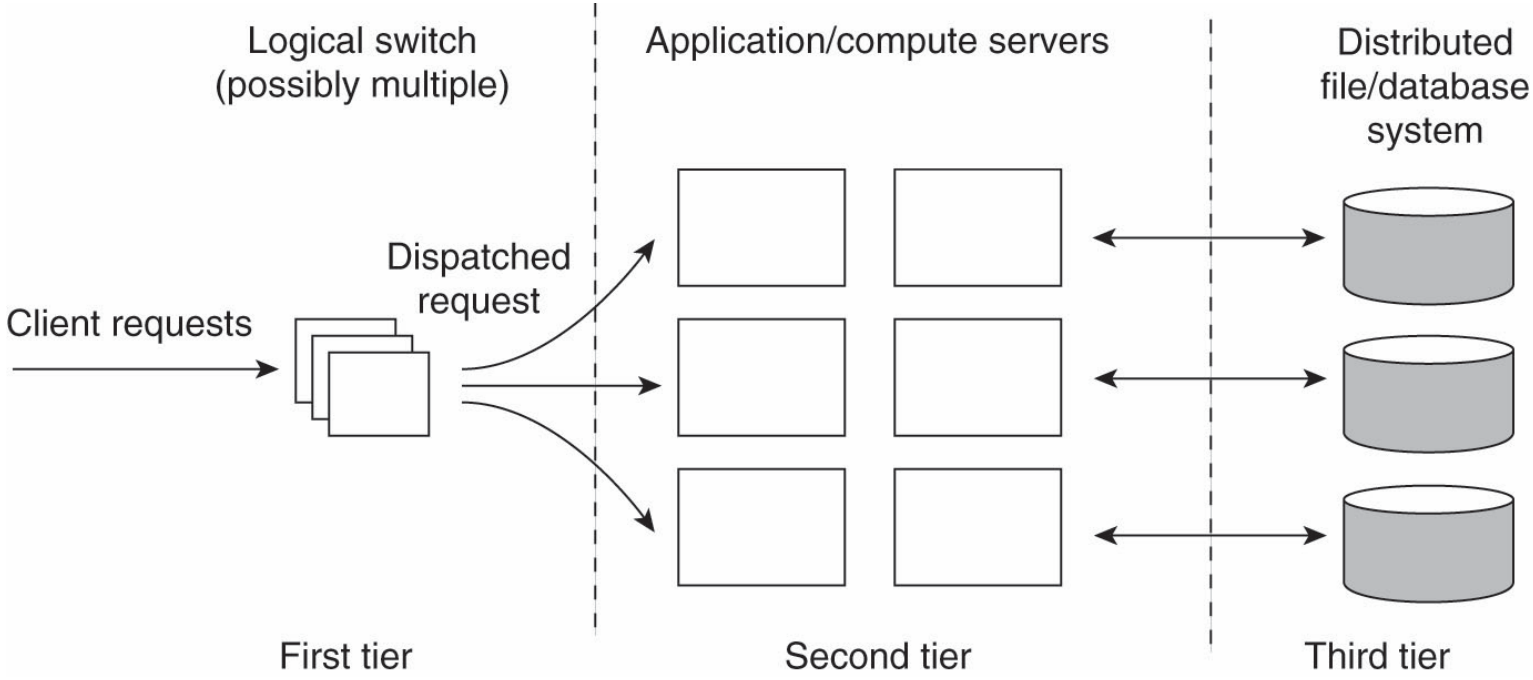


Figure 3-12. The general organization of a three-tiered server cluster.

- ***The third tier***, which consists of data-processing servers, notably file and database servers.
- Again, depending on the usage of the server cluster, these servers may be running an specialized machines, configured for high-speed disk access and having large server-side data caches
- *Not all server clusters will follow this strict separation.*
- It is frequently the case that each machine is equipped with its own local storage, *often integrating application and data processing in a single server leading to a two tiered architecture*.
- For example, when dealing with streaming media by means of a server cluster, it is common to deploy a two-tiered system architecture

- ***A standard way of accessing a server cluster is to set up a TCP connection*** over which application-level requests are then sent as part of a session.
- A session ends by tearing down the connection.
- In the case of transport-layer switches, the switch accepts incoming TCP connection requests, and hands off such connections to one of the servers
- ***The principle working of what is commonly known*** as **TCP handoff** is shown in **Fig. 3-13**.
- *When the switch receives a TCP connection request, it* subsequently *identifies the best server* for handling that request, and *forwards the request* packet *to that server*.
- *The server*, in turn, *will send an acknowledgment back to* the requesting *client*.

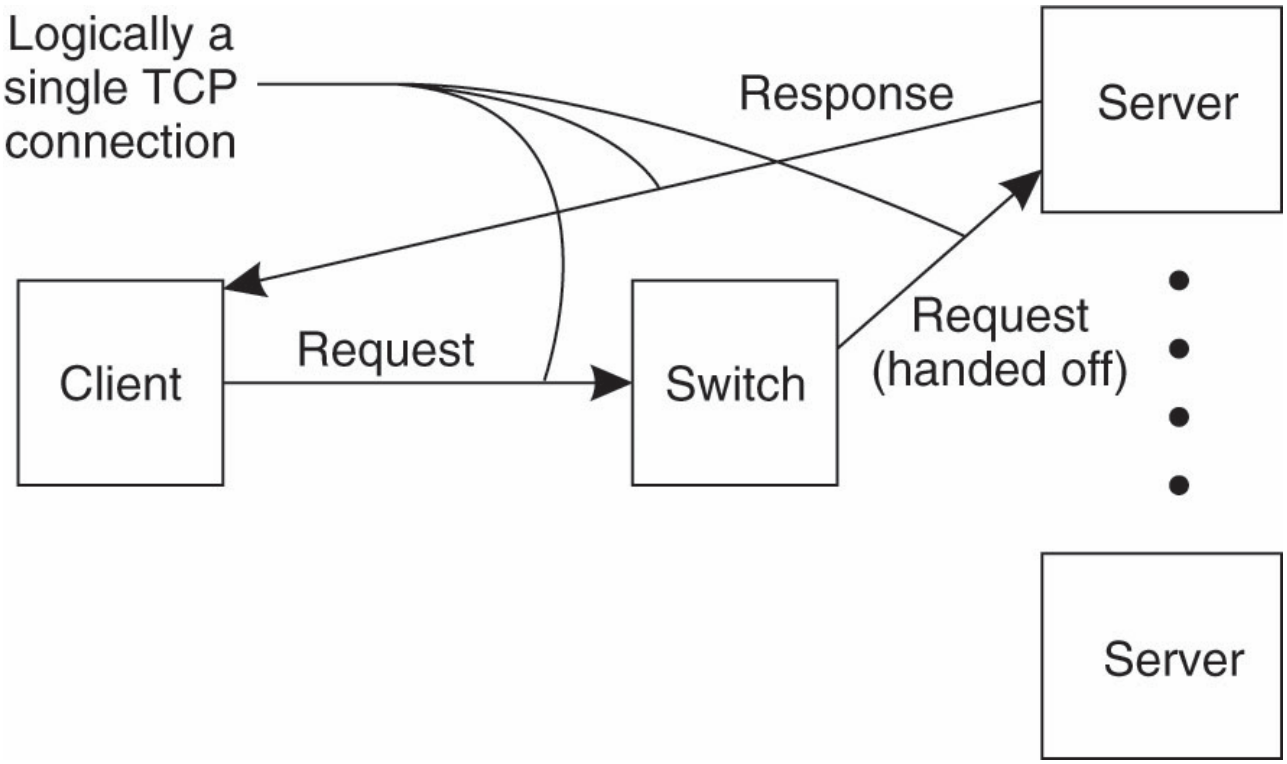


Figure 3-13. The principle of TCP handoff.

- ***The switch can play an important role in distributing the load among the various servers***.
- By deciding where to forward a request to, the switch also decides which server is to handle further processing of the request.
- *The simplest load-balancing policy that the switch can follow is* round robin : each time it picks the next server from its list to forward a request to.

- ***More advanced server selection criteria can be deployed as well.***
- For example, assume multiple services are offered by the server cluster.
- *If the switch can distinguish those services when a request comes in, it can then take informed decisions on where to forward the request to.*

Distributed Servers

- Most server clusters offer a single access point.
- When that point fails, the cluster becomes unavailable.
- To eliminate this potential problem, several access points can be provided, of which the addresses are made publicly available
- This approach still requires clients to make several attempts if one of the addresses fails.
- Moreover, this does not solve the problem of requiring static access points.

- *This observation has lead to a* ***design of a distributed server*** which effectively is nothing but a possibly dynamically changing set of machines, with also possibly varying access points, but which nevertheless-appears to the outside world as a single, powerful machine.
- ***The basic idea behind a distributed server*** is that clients benefit from a robust, high-performing, stable server.
- By grouping simpler machines transparently into a cluster, and not relying on the availability of a single machine, it may be possible to achieve a better degree of stability than by each component individually.

- How a stable access point can be achieved in such a system?
- The main idea is to make use of available networking services, notably mobility support for IP version 6 (MIPv6).
- In MIPv6, **a mobile node is assumed to have a home network** where it normally resides and f**or which it has an associated stable address,** known as its **home address (HoA).**
- This **home network has a special router attached**, known as the **home agent**, **which will take care of traffic to the mobile node when it is away**.

https://www.ipv6.com/mobile/what-is-mobile-ipv6/

- To this end, when a mobile node attaches to a foreign network, it will receive a temporary **care-of address (CoA)** where it can be reached.
- This care-of address is reported to the node's home agent who will then see to it that all traffic is forwarded to the mobile node.
- Note that, applications communicating with the mobile node will only see the address associated with the node's home network.
- *They will never see the care-of address.*

- *This principle can be used to offer a stable address of a distributed server.*
- In this case, a single unique contact address is initially assigned to the server cluster.
- The contact address will be the server's life-time address to be used in all communication with the outside world.

- At any time, one node in the distributed server will operate as an access point using that contact address, but this role can easily be taken over by another node.
- What happens is that the access point records its own address as the care-of address at the home agent associated with the distributed server.
- At that point, all traffic will be directed to the access point, who will then take care in distributing requests among the currently participating nodes.
- If the access point fails, a simple fail-over mechanism comes into place by which another access point reports a new care-of address.

- *This simple configuration would make the home agent as well as the access point a potential bottleneck as all traffic would flow through these two machines.*
- *This situation can be avoided by using* an MIPv6 feature known as route optimization.

- *Route optimization works as follows* :
- Whenever a mobile node with home address HA reports its current care-of address, say CA, the home agent can forward CA to a client.
- The latter will then locally store the pair (HA, CA).
- From that moment on, communication will be directly forwarded to CA.
- Although the application at the client side can still use the home address, the underlying support software for MIPv6 will translate that address to CA and use that instead.

- *Route optimization can be used to make different clients believe they are communicating with a single server*, where, in fact, each client is communicating with a different member node of the distributed server, as shown in **Fig. 3-14**.

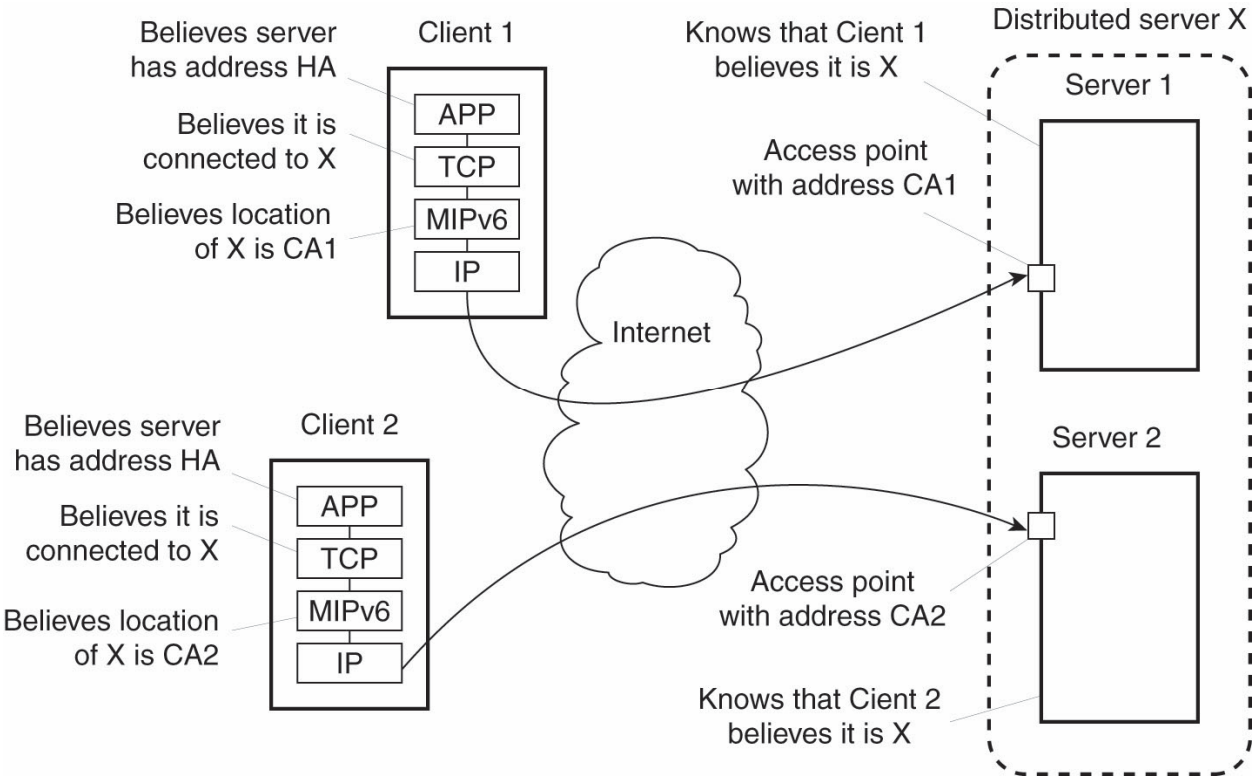


Figure 3-14 : Route optimization in a distributed server.

3.4.3 Managing Server Clusters

Common Approaches

- The most common approach to managing a server cluster is to extend the traditional managing functions of a single computer to that of a cluster.
- ***In its most primitive form***, this means that an administrator can log into a node from a remote client and execute local managing commands to monitor, install, and change components .
- ***More advanced is to hide the fact that*** you need to login into a node and instead provide an interface at an administration machine that allows to collect information from one or more servers, upgrade components, add and remove nodes, etc.

- *The main advantage of the latter approach is that* collective operations, which operate on a group of servers, can be more easily provided.
- This type of managing server clusters is widely applied in practice
- However, as soon as clusters grow beyond several tens of nodes, this type of management is not the way to go.

3.5 CODE MIGRATION

- We have been mainly concerned with distributed systems in which communication is limited to passing data.
- However, there are situations in which passing programs, sometimes even while they are being executed, simplifies the design of a distributed system.

3.5.1 Approaches to Code Migration

- Let us first consider why it may be useful to migrate code.

Reasons for Migrating Code

- Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another
- Moving a running process to a different machine is a costly and intricate task, and there had better be a good reason for doing so.
- *That reason has always been performance*.
- The basic idea is that overall system performance can be improved if processes are moved from heavily-loaded to lightly-loaded machines.

- 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 maybe 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

- Besides improving performance, there are other reasons for supporting code migration as well.
- The most important one is that of flexibility.

- *If code can move between different machines, it becomes possible to dynamically configure distributed systems*.
- The server provide the client's implementation when the client binds to the server.
- At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server. *This principle is shown in* **Fig. 3-17**.
- This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing code is standardized.
- Also, it is necessary that the downloaded code can be executed on the client's machine.

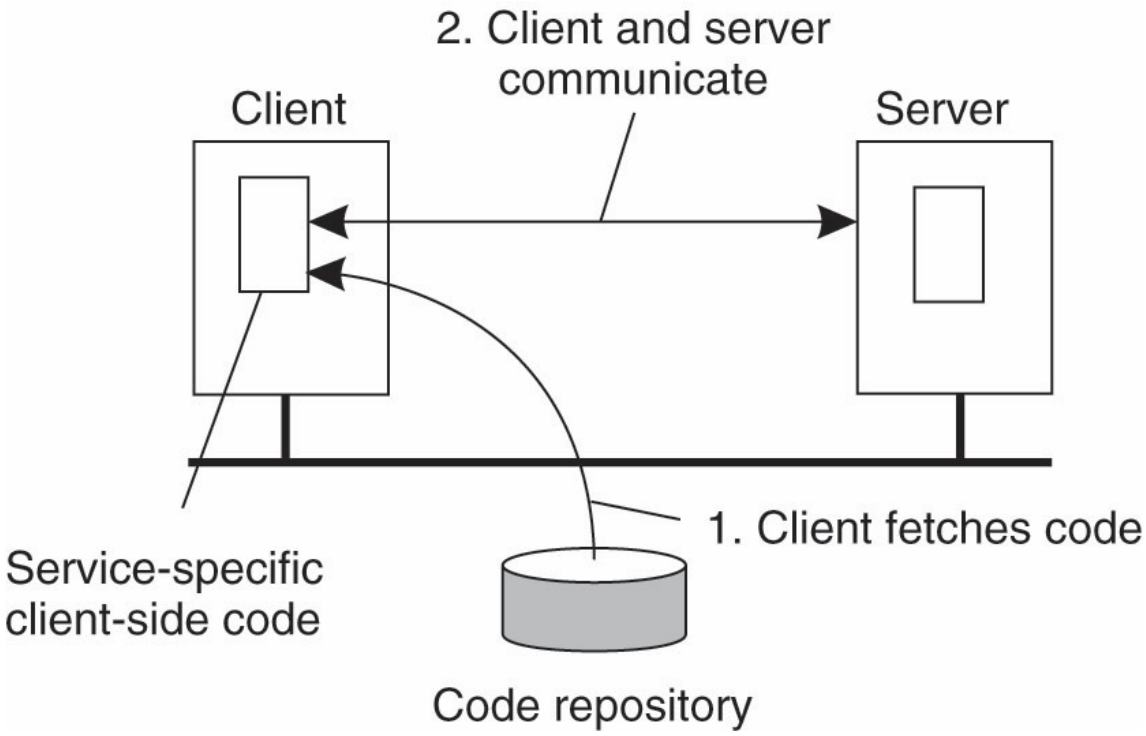


Figure 3-17 : The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

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

- *Disadvantages :* The most serious one has to do with security.
- Blindly trusting that the downloaded code implements only the advertised interface while accessing your unprotected hard disk and does not send the juiciest parts to heaven-knows-who may not always be such a good idea

Models for Code Migration

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

- The bare minimum for code migration is to provide only **weak mobility**.
- In this model, it is possible to transfer only the code segment, along with perhaps some initialization data.
- *A characteristic feature of weak mobility is* that a transferred program is always started from one of several predefined starting positions.
- This is what happens, for example, with Java applets, which always start execution from the beginning.
- *The benefit of this approach is its simplicity*.

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

- *Irrespective of whether mobility is weak or strong, a further distinction can be made between* **sender-initiated** and **receiver-initiated migration**.
- In sender initiated migration, migration is initiated at the machine where the code currently resides or is being executed.
- Typically, sender-initiated migration is done when uploading programs to a compute server.
- Another example is sending a search program across the Internet to a Web database server to perform the queries at that server.

- In *receiver-initiated migration*, the initiative for code migration is taken by the target machine.
- Java applets are an example of this approach.
- Receiver-initiated migration is simpler than sender-initiated migration.

- In the case of *weak mobility*, it also makes a difference if the migrated code is executed by the target process, or whether a separate process is started.
- For example, Java applets are simply downloaded by a Web browser and are executed in the browser's address space.
- The benefit of this approach is that there is no need to start a separate process, thereby avoiding communication at the target machine.
- The main drawback is that the target process needs to be protected against malicious or inadvertent code executions.
- A simple solution is to let the operating system take care of that by creating a separate process to execute the migrated code

- *Instead of moving a running process, also referred to as process migration, strong mobility* can also be supported by remote cloning.
- In contrast to process migration, cloning yields an exact copy of the original process, but now running on a different machine.
- The cloned process is executed in parallel to the original process.
- The benefit of cloning is that the model closely resembles the one that is already used in many applications.
- The only difference is that the cloned process is executed on a different machine.

- In this sense, migration by cloning is a simple way to improve distribution transparency.
- The various alternatives for code migration are summarized in **Fig. 3-18**.

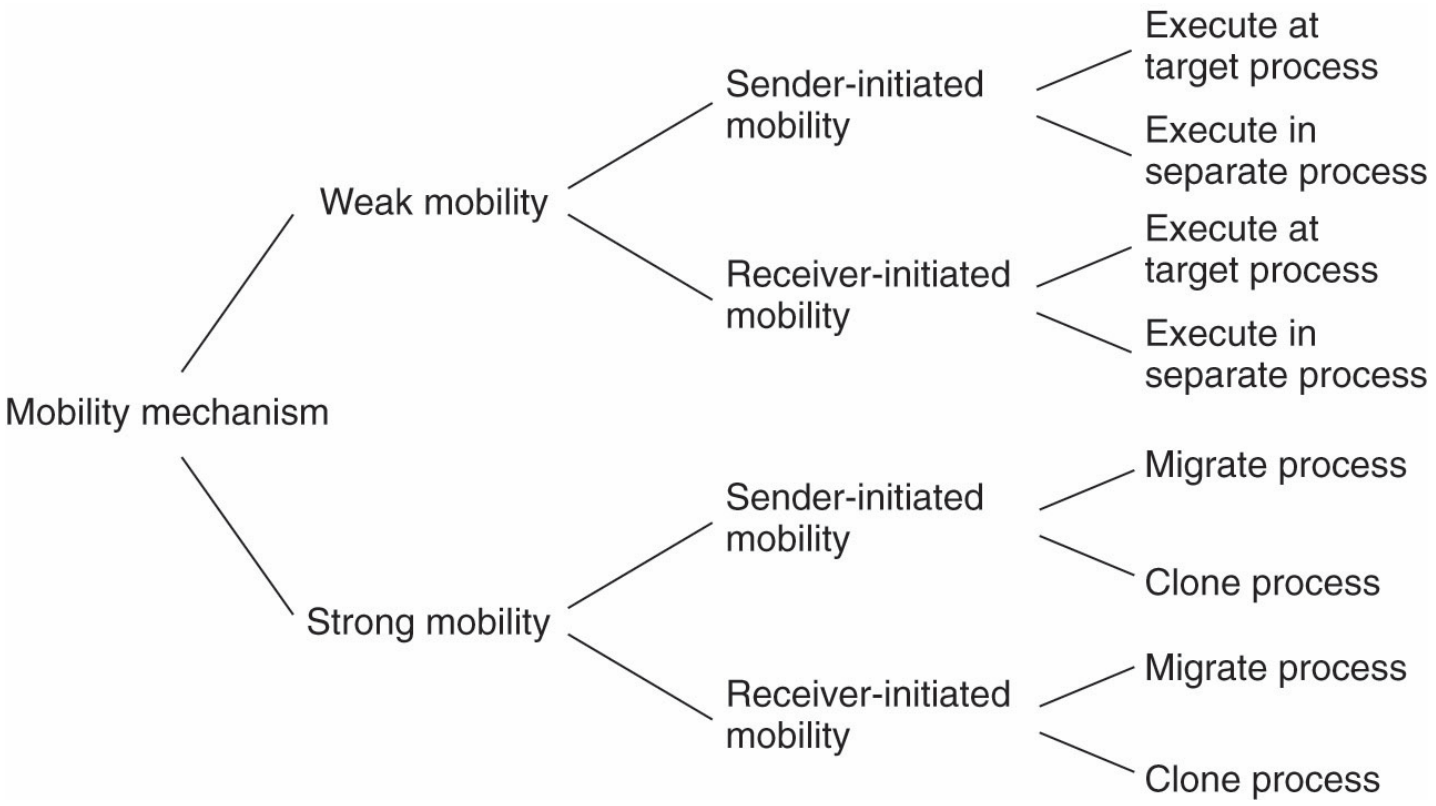


Figure 3-18. Alternatives for code migration.

3.5.2 Migration and Local Resources

- To understand the implications that code migration has on the resource segment we distinguish *three types of process-to-resource bindings.*
- *The strongest binding is when a process refers to a resource* ***by its identifier***.
- In that case, the process requires precisely the referenced resource, and nothing else.

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

- *Finally, the weakest form of binding is when a process indicates it needs only a resource of a **specific type***.
- This binding by type is exemplified by references to local devices, such as monitors, printers, and soon.
- When migrating code, we often need to change the references to resources, but cannot affect the kind of process-to-resource binding.
- If, and exactly how a reference should be changed, depends on whether that resource can be moved along with the code to the target machine.

- More specifically, *we need to consider the* resource-to-machine bindings, and *distinguish the following cases*.
- Unattached resources can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated.
- In contrast, moving or copying a fastened resource may be possible, but only at relatively high costs.
- Typical examples of fastened resources are local databases and complete Web sites.
- Finally, fixed resources are intimately bound to a specific machine or environment and cannot be moved.
- Fixed resources are often local devices.

- *Combining three types of process-to-resource bindings, and three types of resource-to-machine bindings, leads to nine combinations that we need to consider when migrating code*.
- *These nine combinations are* shown in **Fig. 3-19**.
- Consider the possibilities when a process is bound to a resource by identifier.
- When the resource is unattached, it is generally best to move it along with the migrating code.
- However, when the resource is shared by other processes, an alternative is to establish a global reference, that is, a reference that can cross machine boundaries.

- When the resource is fastened or fixed, the best solution is also to create a global reference.

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

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

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

- Another example of where establishing a global reference is not always that easy is when migrating a process that is making use of a local communication end point.
- In that case, we are dealing with a fixed resource to which the process is bound by the identifier.
- *The situation is different when dealing with* ***bindings by value*****.**
- Consider first a fixed resource. The combination of a fixed resource and binding by value occurs, for example, when a process assumes that memory can be shared between processes.
- Establishing a global reference in this case would mean that we need to implement a distributed form of shared memory

- Fastened resources that are referred to by their value, are typically runtime libraries.
- Normally, copies of such resources are readily available on the target machine, or should otherwise be copied before code migration takes place.
- Establishing a global reference is a better alternative when huge amounts of data are to be copied.
- The easiest case is when dealing with unattached resources.
- The best solution is to copy (or move) the resource to the new destination, unless it is shared by a number of processes.

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

3.5.3 Migration in Heterogeneous Systems

- In general, however, distributed systems are constructed on a heterogeneous collection of platforms, each having their own operating system and machine architecture.
- Migration in such systems requires that each platform is supported, that is, that the code segment can be executed on each platform.
- Also, we need to ensure that the execution segment can be properly represented at each platform.

- Let us consider one specific example of migrating virtual machines.
- Migration involves two major problems: *migrating the entire memory image* and *migrating bindings to local resources.*

- As to the first problem, there are, in principle, *three ways to handle migration :*
- 1. Pushing memory pages to the new machine and resending the ones that are later modified during the migration process.
- 2. Stopping the current virtual machine; migrate memory, and start the new virtual machine.
- 3. Letting the new virtual machine pull in new pages as needed, that is, let processes start on the new virtual machine immediately and copy memory pages on demand.