DISTRIBUTED SYSTEMS
Principles and Paradigms

Second Edition

ANDREW S. TANENBAUM
MAARTEN VAN STEEN

Chapter 1
Introduction

Definition of a Distributed System

A collection of independent computers that appears to its users as a single coherent system.

Definition of a Distributed System

Several important aspects:

1. Consists of components that are autonomous.
2. Users think they are dealing with a single system.
3. No assumptions made regarding the types of computers or the way they are interconnected.

Definition of a Distributed System

Distributed systems are often organized by means of a layer of software -

- logically placed between a higher layer consisting of users and applications, and a lower layer consisting of operating systems and basic communication facilities
- **middleware.**

Definition of a Distributed System

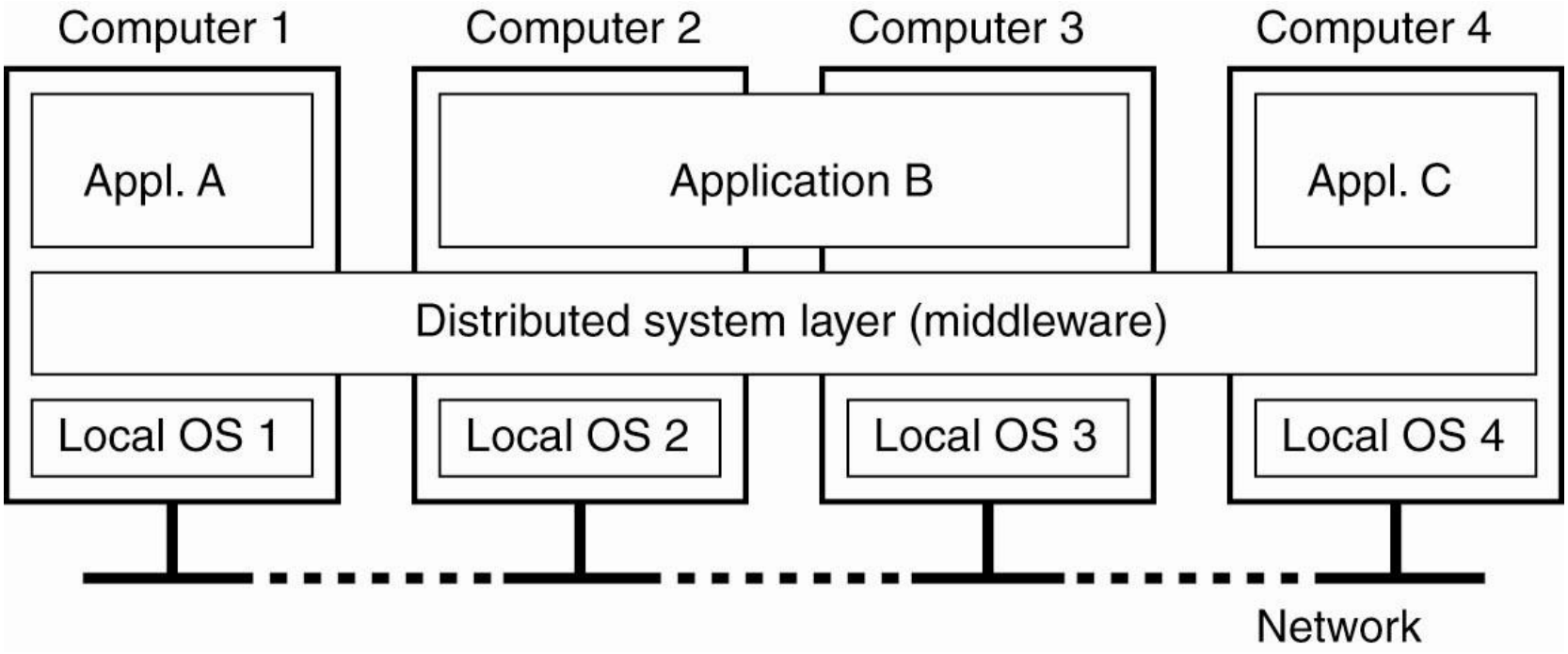


Figure 1-1. A distributed system organized as middleware. The *middleware* layer extends over multiple machines, and offers each application the same interface.

Goals of Distributed Systems

- Making resources accessible
- Distribution transparency
- Openness
- Scalability

Making Resources Accessible

- Making it easy for users and applications to access remote resources
- Share remote resources in a controlled and efficient manner

Making Resources Accessible

- Benefits of sharing remote resources
  - Better economics by sharing expensive resources
  - Easier to collaborate and exchange information
  - Connectivity of the Internet has lead to numerous virtual organizations where geographically dispersed people can work together using ***groupware***
  - Connectivity has enabled *electronic commerce*
  - However, as connectivity and sharing increase …
- Security problems
  - Eavesdropping or intrusion on communication
  - Tracking of communication to build up a preference profile of a specific user

Distribution Transparency

An important goal - hide the fact that the processes and resources are physically distributed across multiple computers.

Transparent - A distributed system that presents itself to users and applications as if it were only a single computer system.

Types of Transparency

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 (ISO, 1995).

Types of Transparency

Access transparency - hide differences in data representation and the way the resources are accessed

Ex: a distributed system may have computer systems that run different operating systems, each having their own file-naming conventions.

Types of Transparency

Location transparency - users cannot tell where a resource is physically located in the system. Achieved by assigning only logical names to resources.

Ex: *http://www.prenhall.com/index.html*

Migration transparency - resources can be moved without affecting how those resources can be accessed.

Types of Transparency

Relocation transparency - resources can be relocated *while* they are being accessed without the user or application noticing anything.

Ex: when mobile users can continue to use their wireless laptops while moving from place to place.

Replication transparency - hide the fact that several copies of a resource exist.

Types of Transparency

Sharing of resources can also be done in a competitive way.

Ex: 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.

This phenomenon is called concurrency transparency.

Failure transparency - a user does not notice that a resource fails to work properly, and that the system subsequently recovers from that failure.

Degree of Transparency

Complete hiding the distribution aspects from users is not always a good idea.

- Attempting to mask a server failure before trying another one may slow down the system
- Requiring several replicas to be always consistent means a single update operation may take seconds to complete
- For mobile and embedded devices, it may be better to *expose* distribution rather than trying to hide it
- Signal transmission is limited by the speed of light as well as the speed of intermediate switches.

Openness

An *open* distributed system offers services according to standard rules that describe the syntax and semantics of those services.

- Services are generally specified through *interfaces*, which are often described in an *Interface Definition Language (IDL).*

Openness

- An interface definition - allows an arbitrary process that needs a certain interface to talk to another process that provides that interface.
  allows two independent parties to build completely different implementations of those interfaces.
- Proper specifications are complete and neutral.
- Completeness and neutrality are important for interoperability and portability.

Openness

Interoperability - characterizes the extent by which two implementations of systems or components from different manufacturers can co-exist and work together by merely relying on each other's services as specified by a common standard

Portability characterizes to what extent an application developed for a distributed system *A* can be executed. without modification, on a different distributed system *B that* implements the same interfaces as *A.*

Openness

Extensibility

- It should be easy to configure the system out of different components
- It should be easy to add new components or replace existing ones.

Scalability

Scalability can be measured against three dimensions.

- *Size*: be able to easily add more users and resources to a system
- *Geography*: be able to handle users and resources that are far apart
- *Administrative*: be easy to manage even if it spans many independent administrative organizations

Scalability Problems

Consider scaling w.r.t. size - we are often confronted with the limitations of centralized services, data and algorithms.

Scalability Problems

Concept	Example
Centralized services	A single server for all users
Centralized data	A single on-line telephone book
Centralized algorithms	Doing routing based on complete information

Figure 1-3. Examples of scalability limitations.

Scalability Problems

Only decentralized algorithms should be used.

Characteristics of *decentralized* algorithms:

- No machine has complete information about the system state
- Machines make decisions based only on local information
- Failure of one machine does not ruin the algorithm
- There is no implicit assumption that a global clock exists

Scalability Problems

LANs use synchronous communication. Designing WANs using synchronous communication is much more difficult

Communication in WANs is inherently unreliable, and virtually always point-to-point. LANs use broadcasting.
Ex: this makes it very easy to locate a service.

Scaling across multiple, independent administrative domains leads to conflicting policies w.r.t. resource usage, payment, management and security.

Scalability Problems

Security issues:

Many components of a distributed system that resides within a single domain, may not be trusted by users in other domains.

Scaling Techniques

How can the scalability problems be solved?

Three techniques for scaling:

- Hiding communication latencies
- Distribution
- Replication

Scaling Techniques

Hiding communication latencies:

Basic idea: Try to avoid waiting for responses to remote service requests as much as possible.

In applications that cannot make effective use of asynchronous communication, a better solution is to reduce the overall communication.

Scaling Techniques

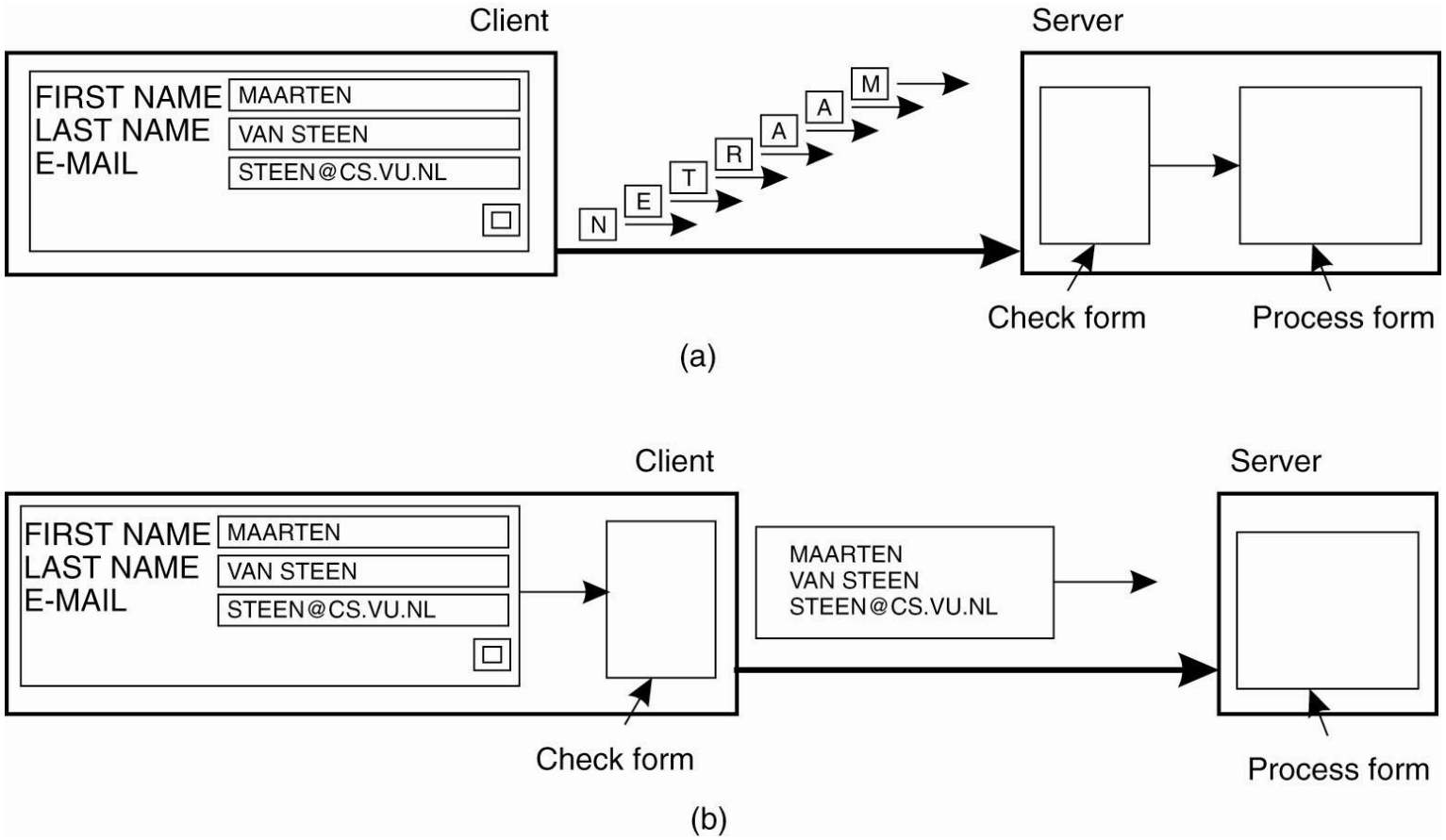


Figure 1-4. The difference between letting (a) a server or (b) a client check forms as they are being filled.

Scaling Techniques

- *Distribution*: Taking a component, splitting into smaller parts, and subsequently spreading them across the system.
  Ex: the Internet Domain Name System (DNS).
  - The DNS namespace is hierarchically organized into a tree of **domains,** which are divided into nonoverlapping **zones**.

Scaling Techniques

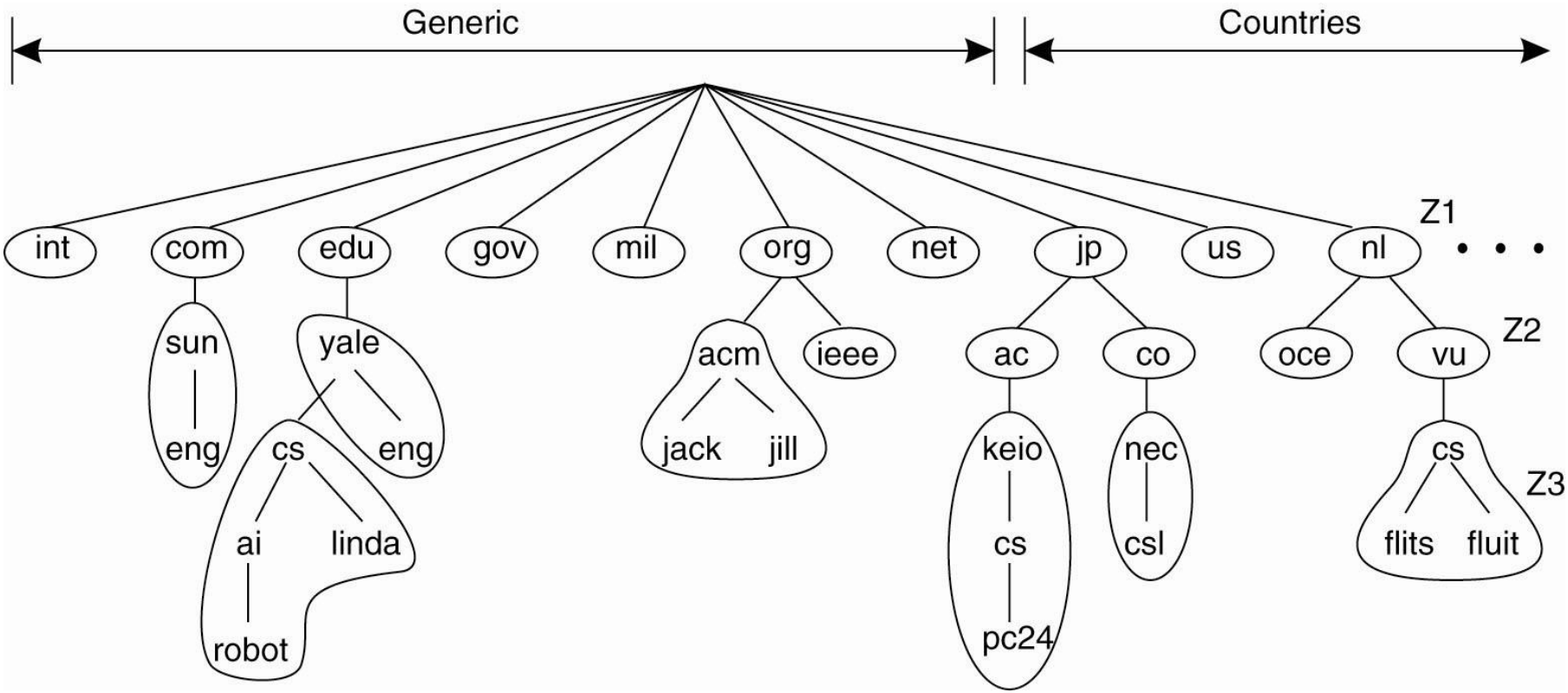


Figure 1-5. An example of dividing the DNS name space into zones.

Scaling Techniques

- *Replication*: increases availability and helps balance the load between components leading to better performance.
- *Caching:* special form of replication - making a copy of the resource, generally in the proximity of the client accessing that resource.

Scaling Techniques

One serious drawback to caching and replication - consistency problems.

Scaling Techniques

Size scalability - least problematic from a technical point of view.

Geographical scalability is a much tougher problem

Administrative scalability is the most difficult one, partly also because we need to solve nontechnical problems

Pitfalls when Developing Distributed Systems

False assumptions made by first time developer:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

System Models

An architectural model of a distributed system:

- concerned with the placement of its parts and the relationships between them.

Fundamental models – concerned with a more formal description of the properties that are common in all of the architectural models.

Architectural Models

An architectural model considers:

- the placement of the components across a network of computers
- the interrelationships between the components - i.e., their functional roles and the patterns of communication between them.

Architectural Models

There are different ways on how to view the organization of a distributed system.

Software architecture – tells us how the various software components are to be organized and how they should interact.

There are different ways on how we instantiate and place software components on real machines.

System architecture – the final instantiation of a software architecture.

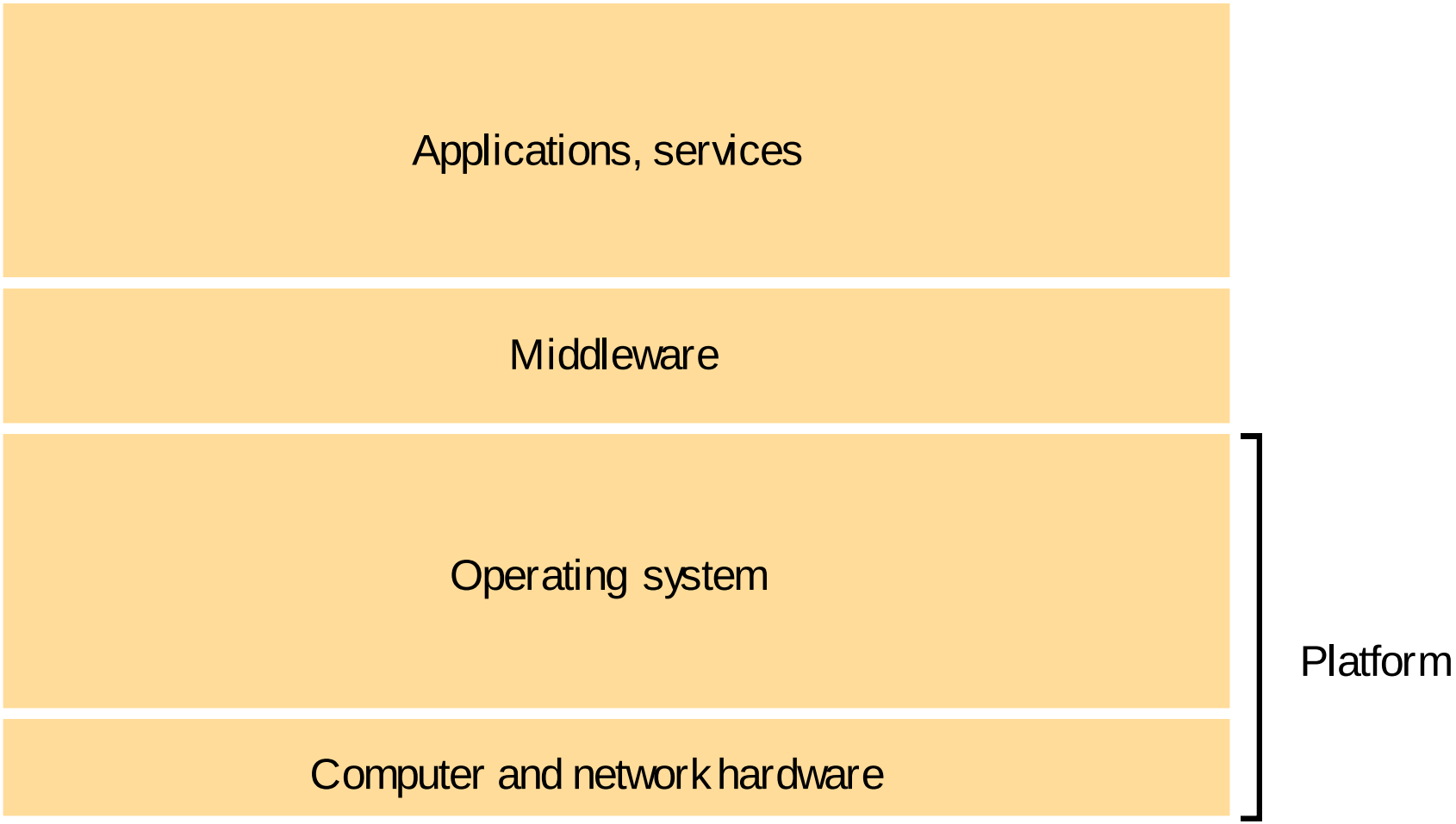
Software Layers

Software Architecture

- refers to services offered and requested between processes located in the same or different computers.
- This process- and service-oriented view can be expressed in terms of *service layers*.

A distributed service can be provided by one or more server processes, interacting with each other and with client processes.

Software and hardware service layers in distributed systems



Software Layers

Platform - the lowest-level hardware and software layers

Middleware

- a layer of software whose purpose is to mask heterogeneity
- provides a convenient programming model to application programmers.
- represented by processes or objects.
- raises the level of the communication activities of application programs through the support of abstractions.

Software Layers

Earliest instances of middleware - remote procedure calling packages - Sun RPC

Object-oriented middleware products and standards:

- CORBA
- Java RMI
- web services
- Microsoft's Distributed Component Object Model (DCOM)
- the ISO/ITU-T's Reference Model for Open Distributed Processing (RM-ODP)

Limitations of middleware

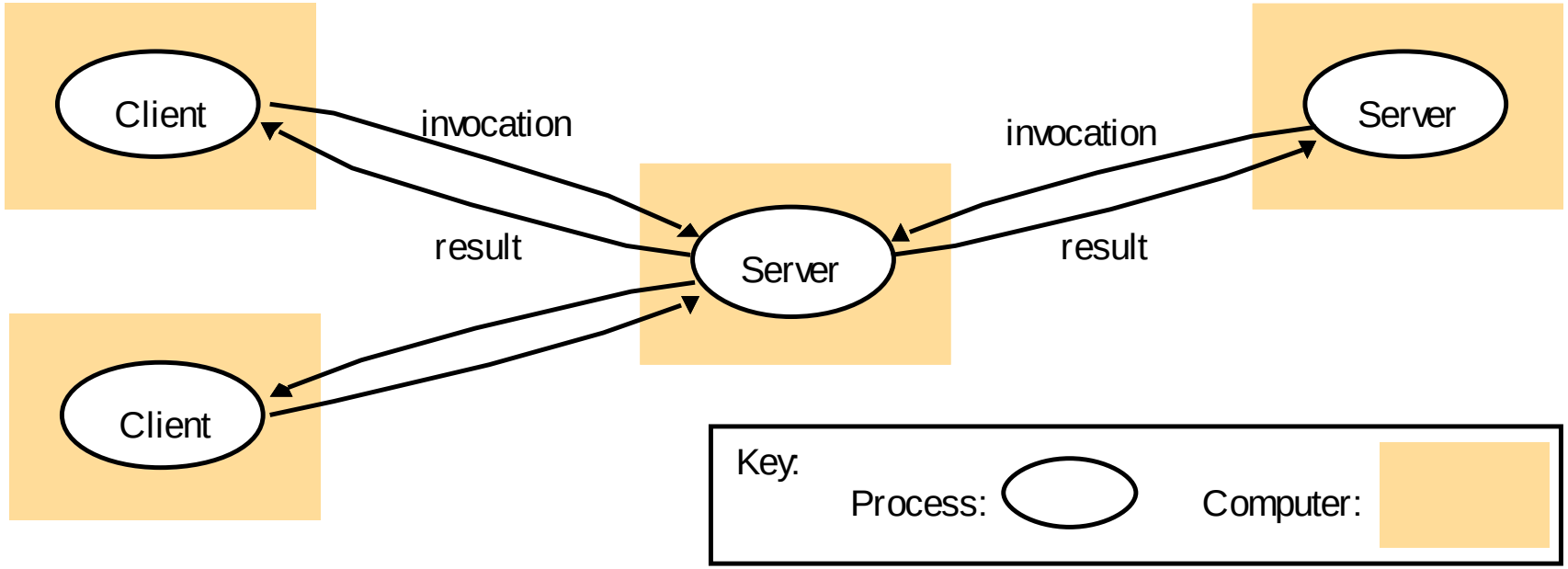
System architectures

The two main types of architectural models - client-server model and the peer-to-peer model.

Client-server:

- the architecture that is most often cited when discussing distributed systems
- historically the most important and remains the most widely employed

Clients invoke individual servers



Client-server

Servers may in turn be clients of other servers.

Ex: A web server is often a client of a local file server that manages the files in which the web pages are stored.

- - - another example.
Search engines - enable users to look up summaries of information available on web pages at sites throughout the Internet.
- made by programs called web crawlers.

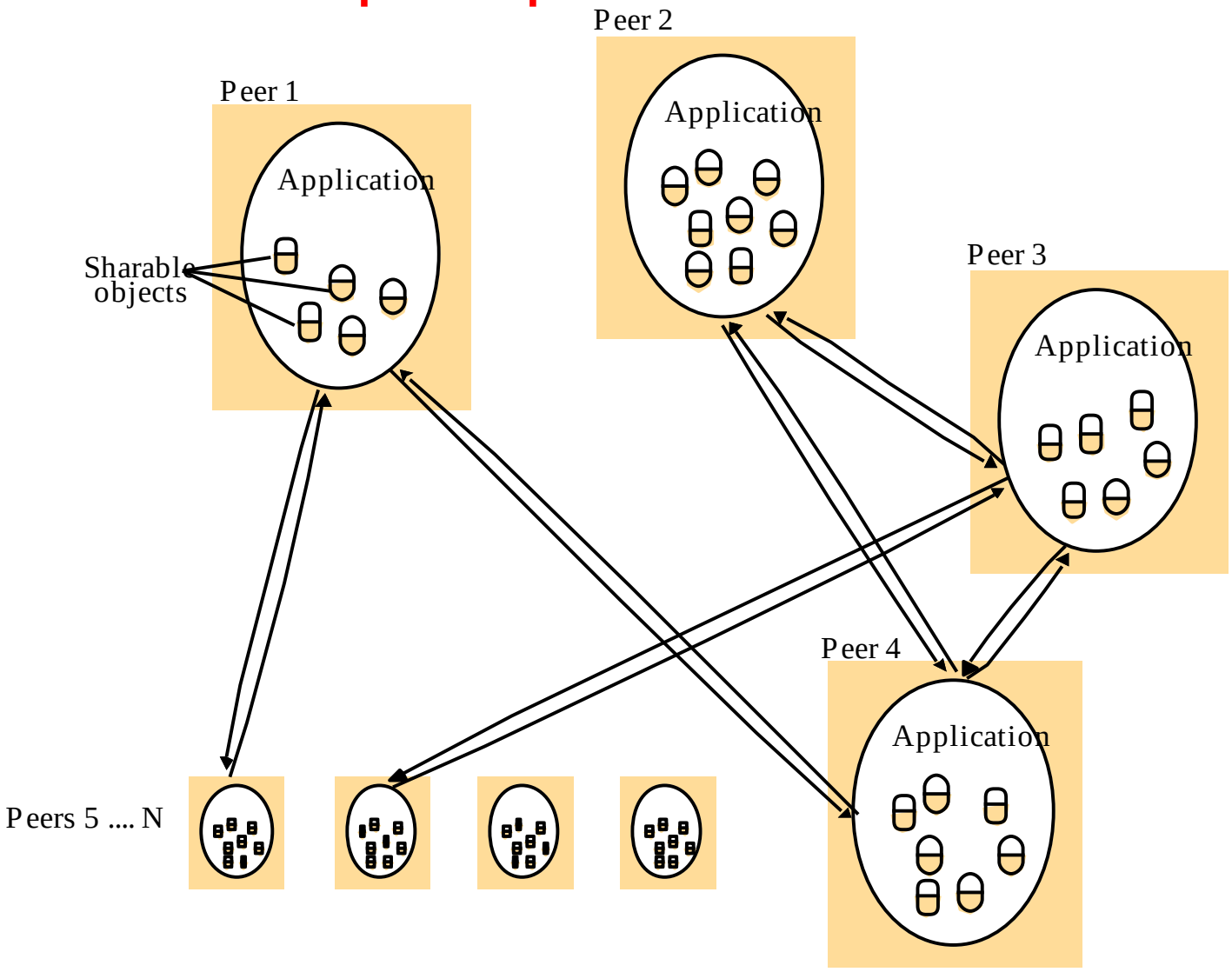
System architectures

Disadvantage of client-server model: It scales poorly.

Peer-to-peer:

- All the processes involved in a task play similar roles, interacting cooperatively as *peers*.
- The aim is to exploit the resources in a large number of participating computers for the fulfillment of a given task or activity.

A distributed application based on peer processes



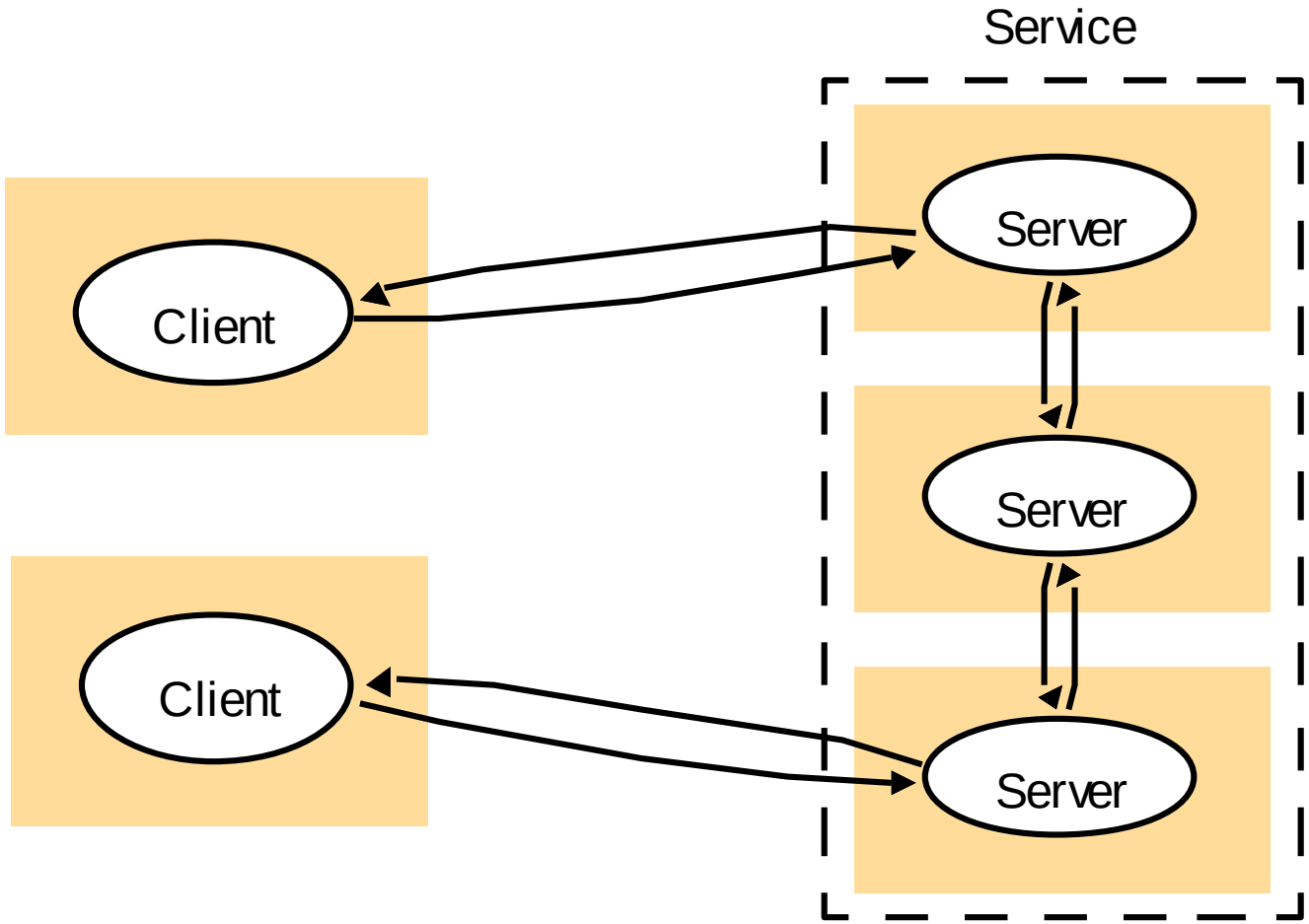
Peer-to-peer

- Applications are composed of large numbers of peer processes running on separate computers
- the pattern of communication between them depends entirely on application requirements.
- A large number of data objects are shared.

Variations

- the use of multiple servers and caches to increase performance and resilience;
- the use of mobile code and mobile agents;
- users' need for low-cost computers with limited hardware resources that are simple to manage;
- the requirement to add and remove mobile devices in a convenient manner.

A service provided by multiple servers



Variations

Services provided by multiple servers:

- Services may be implemented as several server processes in separate host computers.
- The servers may partition the set of objects on which the service is based and distribute them between themselves, or they may maintain replicated copies of them on several hosts.

Variations

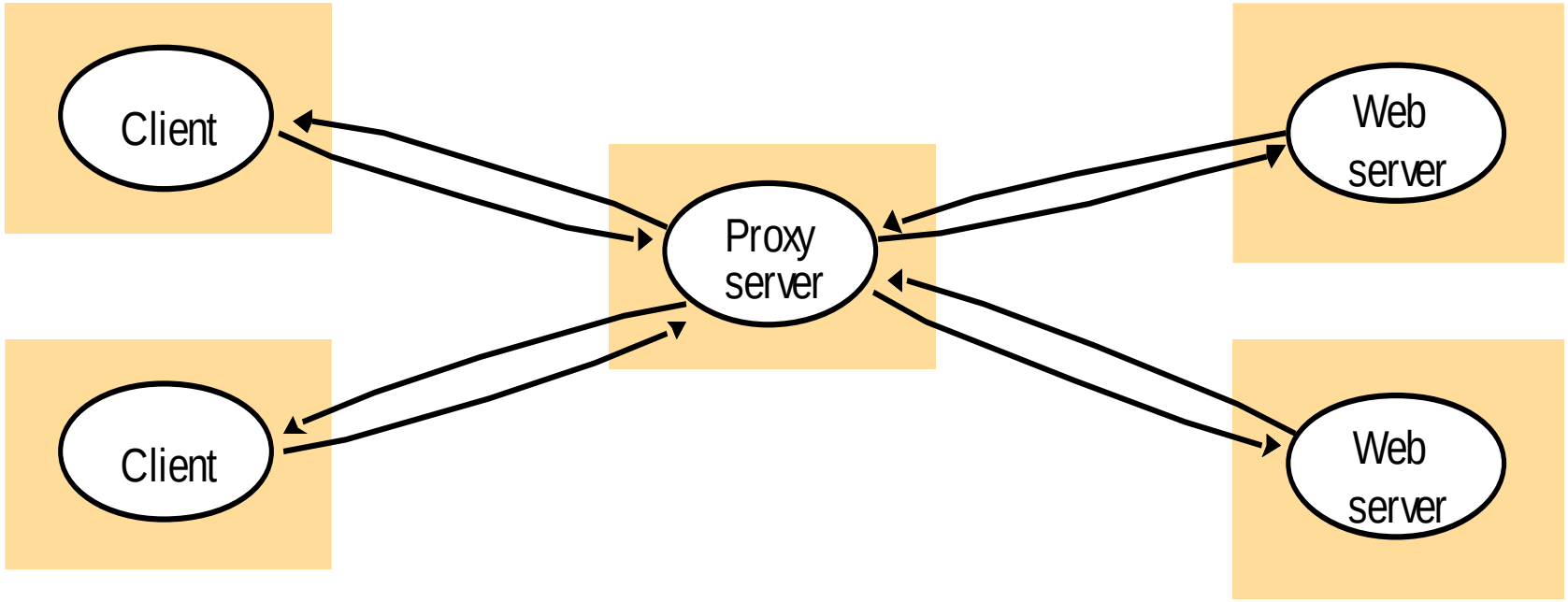
examples:

The Web provides an example of partitioned data in which each web server manages its own set of resources.

Sun NIS – an example of a service based on replicated data.

A more closely-coupled type of multiple-server architecture is the cluster, which is used for highly scalable web services such as search engines and online stores.

Web proxy server



Variations

Proxy servers and caches:

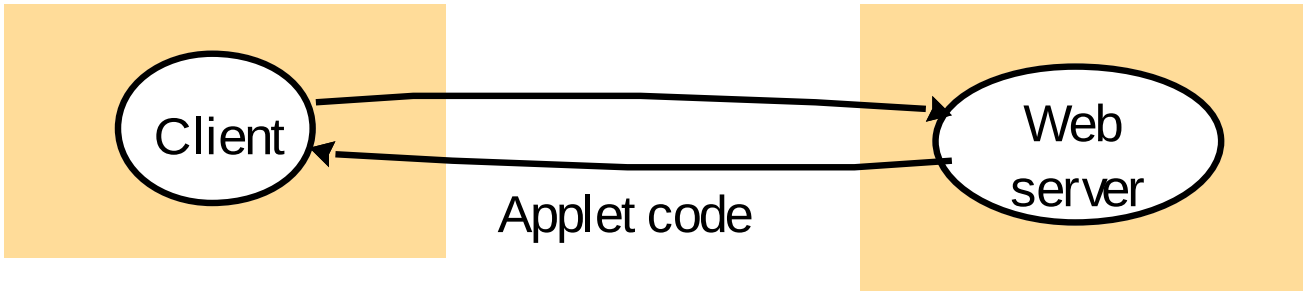
Web browsers maintain a cache of recently visited web pages and other web resources in the client's local file system.

Web proxy servers

- provide a shared cache of web resources for the client machines at a site or across several sites.
- The purpose is to increase availability and performance of the service by reducing the load on the wide-area network and web servers.

Web applets

a) client request results in the downloading of applet code



b) client interacts with the applet

Variations

Mobile code: Applets

- the user running a browser selects a link to an applet whose code is stored on a web server;
- the code is down loaded to the browser and runs there.
- advantage - good interactive response.

Why mobile code? Some web sites use functionality not found in standard browsers and require the downloading of additional code.

Ex: a stockbroker provides a service to notify customers of changes in the prices of shares.

Variations

Mobile agent:

- a running program that travels from one computer to another carrying out a task on someone's behalf.
- may make many invocations to local resources at each site it visits.
- reduction in communication cost and time when compared with a static client making remote invocations to some resources.

Variations

Mobile agents:

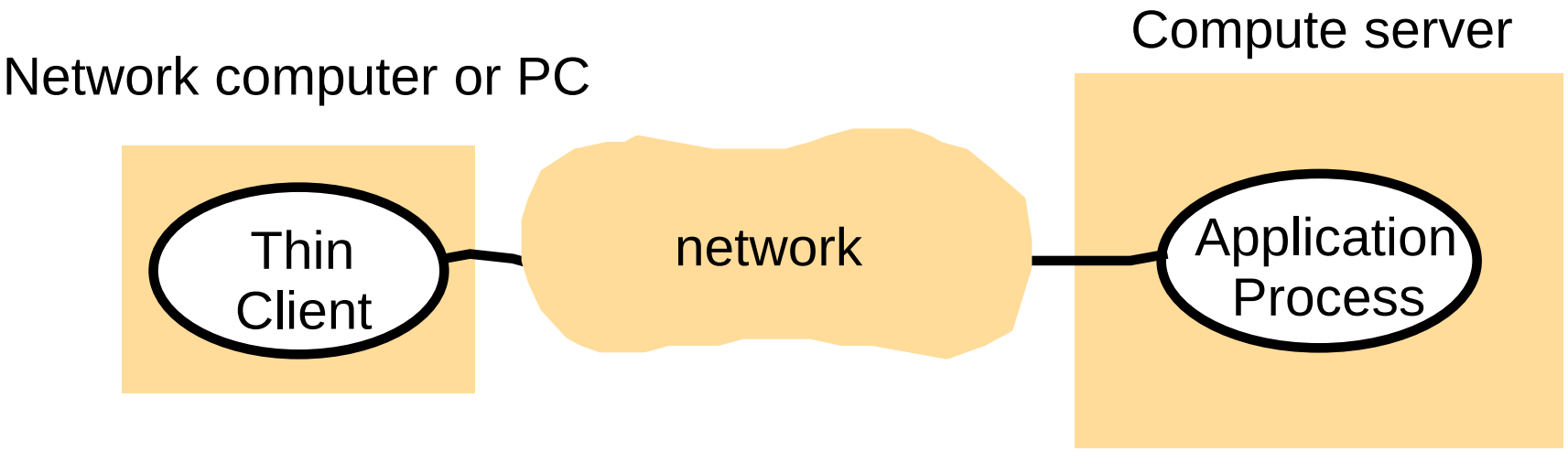
- mobile agents (like mobile code) are a potential security threat to the resources in computers that they visit.
- mobile agents can themselves be vulnerable.

Variations

Network computers:

- downloads the OS and any application software needed by the user from a remote file server.
- applications are run locally but the files are managed by a remote file server.
- the users may migrate from one network computer to another.

Thin clients and compute servers



Variations

Thin client:

- a software layer that supports a window-based user interface on a computer that is local to the user while executing application programs on a remote computer.
- has the same low management and hardware costs as the network computer scheme
- instead of downloading the code of applications into the user's computer, it runs them on a *compute server*.

Variations

Thin clients:

- Main drawback is in highly interactive graphical activities such as CAD and image processing

Design requirements for distributed architectures

Performance Issues:

Responsiveness - Users of interactive applications require a fast and consistent response, but client programs need to access shared resources;

Throughput - ability of a distributed system to perform work for all its users;

Balancing computational loads - enable applications and service processes to proceed concurrently without competing for the same resources and to exploit the available computational resources.

Design requirements for distributed architectures

Quality of service:

functional and non-functional requirements
reliability, security, performance, adaptability
ability to meet timeliness guarantees
applications handling *time-critical data*

Design requirements for distributed architectures

Use of caching and replication:

overcomes obstacles to successful deployment of distributed systems

Web-caching protocol – both web browsers and proxy servers cache responses to client requests

- a browser or proxy can *validate* a cached response

Design requirements for distributed architectures

Dependability issues:

Fault tolerance – Dependable applications should continue to function correctly in the presence of faults in hardware, software and networks.

- reliability is achieved through redundancy

Security – the architectural impact of the requirements for security concerns the need to locate sensitive data and other resources only in computers that can be secured effectively against attack.

Fundamental models

– concerned with a more formal description of the properties that are common in all of the architectural models.

All of the architectural models

- are composed of processes that communicate with one another
- are concerned with the performance & reliability characteristics of processes and networks
- are concerned with the security of the resources in the system

Fundamental models

Fundamental models are presented to discuss about:

Interaction – processes interact by passing messages, resulting in communication & coordination between processes

Failure – the correct operation of a distributed system is threatened whenever a fault occurs in any of the computers or in the network

Security – the modular nature of distributed systems & their openness exposes them to attack

Interaction model

Concept of an algorithm
- single process

Distributed systems are composed of multiple processes.

Behaviour and state of a distributed system can be described by a *distributed algorithm*
- a definition of the steps to be taken by each of the processes of which the system is composed, *including the transmission of messages between them.*

Interaction model

Interacting processes perform all of the activities in a DS.

Each process has its own state.

Two significant factors affecting interacting processes in a distributed system:

- communication performance is a limiting characteristic;
- impossible to maintain a single global notion of time.

Interaction model

Performance of communication channels

Communication channels are realized in a variety of ways.

Communication has the following characteristics:

Latency - The delay between the start of a message's transmission from one process and the beginning of its receipt by another.

B**andwidth** of a computer network - the total amount of information that can be transmitted over it in a given time.

Jitter - the variation in the time taken to deliver a series of messages.

Interaction model

Computer clocks and timing events

Each computer in a distributed system has its own internal clock.

Used by processes to associate timestamps with their events.

Computer clocks drift from perfect time.

Clock drift rate.

Interaction model

Two variants of the interaction model

- It is hard to set time limits on the time taken for process execution, message delivery or clock drift.

- Two opposing extreme positions provide a pair of simple models:

- *Synchronous distributed systems*
- *Asynchronous distributed systems*

Interaction model

Synchronous distributed systems:

- the time to execute each step of a process has known lower and upper bounds;
- each message transmitted over a channel is received within a known bounded time;
- each process has a local clock whose drift rate from real time has a known bound.

It is possible to suggest likely upper and lower bounds …

But it is difficult to provide realistic values …

Interaction model

Asynchronous distributed systems:

There are no bounds on

- process execution speeds;
- message transmission delays;
- clock drift rates.

Actual distributed systems are very often asynchronous because of the need for processes to share the processors and for communication channels to share the network.

But there are many design problems that cannot be solved for an asynchronous system, without using some aspects of time.

Interaction model

Event ordering

- In many cases, we are interested in knowing whether an event at one process occurred before, after or concurrently with another event at another process.

- The execution of a system can be described in terms of events and their ordering despite the lack of accurate clocks.

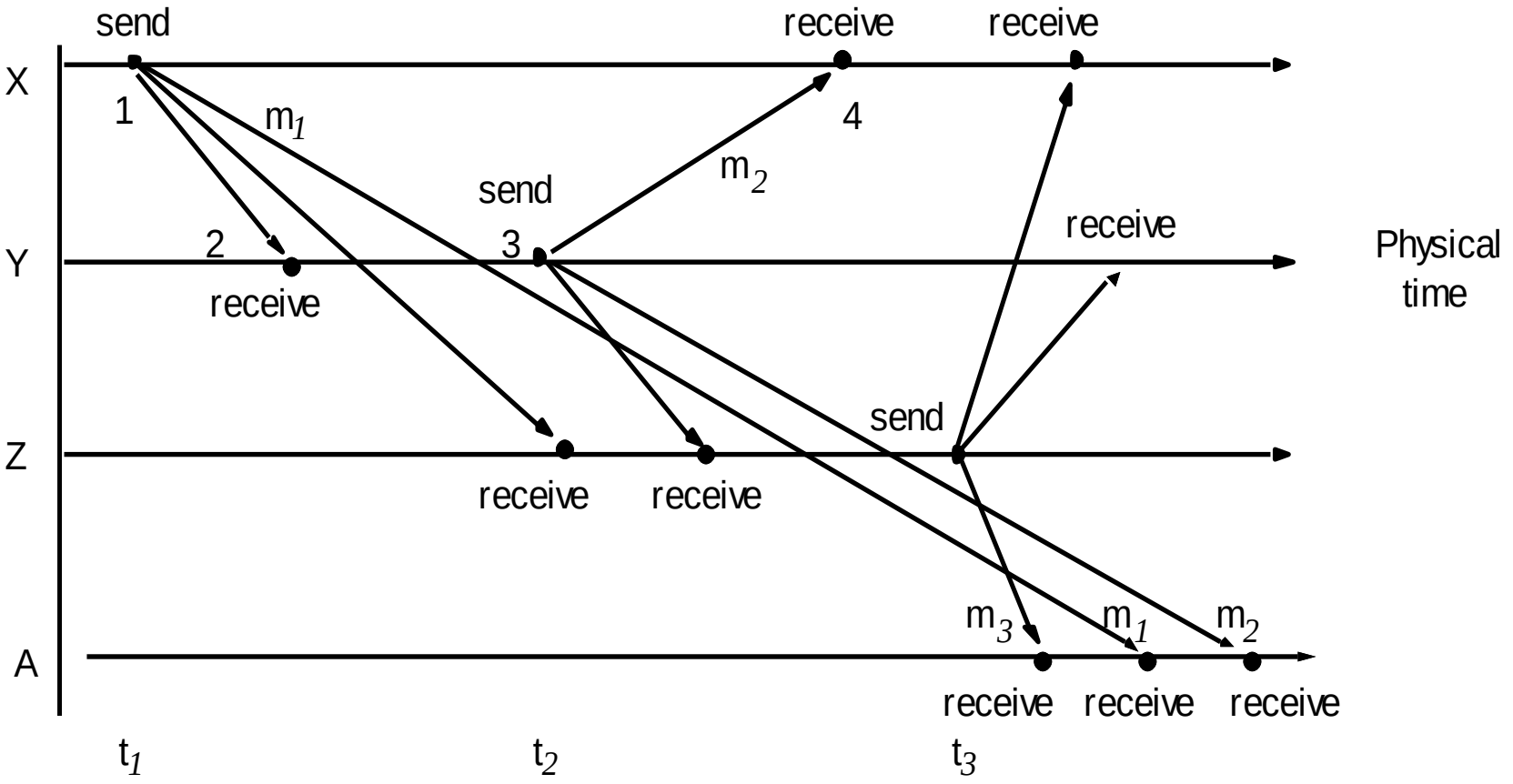
Interaction model

Event ordering example-

Following is a set of exchanges between a group of email users X, Y, Z and A on a mailing list:

1. user X sends a message with the subject *Meeting;*
2. users Y and Z reply by sending a message with the subject *Re: Meeting.*

Figure 2.8
Real-time ordering of events



Interaction model

user A might see:

		Inbox:
Item	*From*	*Subject*
23	Z	Re: Meeting
24	X	Meeting
25	Y	Re: Meeting

Interaction model

If the clocks on X's, Y's and Z's computers could be synchronized, then each message could carry the time on the local computer's clock when it was sent.

Interaction model

Clocks cannot be synchronized perfectly across a distributed system.

Lamport proposed a model of *logical time* that can be used to provide an ordering among the events at processes running in different computers.

Failure model

Defines the ways in which failure may occur in order to provide an understanding of the effects of failures.

Distinguishes between the failures of processes and communication channels.

Omission failures
Arbitrary failures
Timing failures

Failure model

Omission failures:

- when a process or communication channel fails to perform actions that it is supposed to do.

Process omission failures: The chief omission failure of a process is to crash.

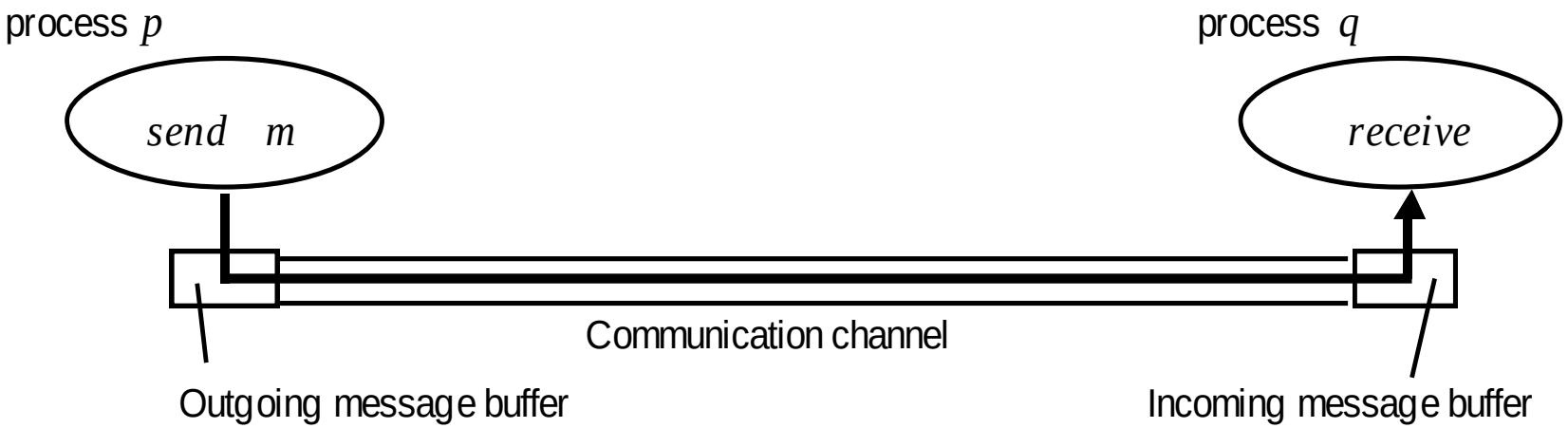
- The design of services that can survive in the presence of faults can be simplified …
- A process crash is called *fail-stop* if other processes can detect certainly that the process has crashed.

Failure model

Communication omission failures:

- communication primitives *send* and *receive*
- if the communication channel does not transport a message from *p's* outgoing message buffer to *q's* incoming message buffer.
- known as 'dropping messages'.
- *send-omission failures;*
- *receive-omission failures;*
- *channel-omission* failures.

Figure 2.9
Processes and channels



Failure model

Failures described so far are *benign* failures.

Failures can also be *arbitrary.*

Failure model

Arbitrary failures (Byzantine failures):

- used to describe the worst possible failure semantics.
- an arbitrary failure of a process is one in which it arbitrarily omits intended processing steps or takes unintended processing steps.
- communication channels can suffer from arbitrary failures.

Figure 2.10
Omission and arbitrary failures

Class of failure	*Affects*	*Description*
Fail-stop	Process	Process halts and remains halted. Other processes may detect this state.
Crash	Process	Process halts and remains halted. Other processes may not be able to detect this state.
Omission	Channel	A message inserted in an outgoing message buffer never arrives at the other end's incoming message buffer.
Send-omission	Process	A process completes a *send*, but the message is not put in its outgoing message buffer.
Receive-omission	Process	A message is put in a process's incoming message buffer, but that process does not receive it.
Arbitrary (Byzantine)	Process or channel	Process/channel exhibits arbitrary behaviour: it may send/transmit arbitrary messages at arbitrary times, commit omissions; a process may stop or take an incorrect step.

Failure model

Timing failures: applicable in synchronous distributed systems where time limits are set on process execution time, message delivery time and clock drift rate.

Figure 2.11
Timing failures

Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Failure model

Masking failures:

- each component is generally constructed from a collection of other components.
- possible to construct reliable services from components that exhibit failures.
- a service *masks* a failure, either by hiding it altogether or by converting it into a more acceptable type of failure.

Failure model

Reliability of one-to-one communication:

validity: any message in the outgoing message buffer is eventually delivered to the incoming message buffer;

integrity: the message received is identical to one sent, and no messages are delivered twice.

Failure model

Reliability of one-to-one communication:

The threats to integrity come from:

- Any protocol that retransmits messages but does not reject a message that arrives twice.
- Malicious users that may inject spurious messages, replay old messages or tamper with messages.

Security model

Security can be achieved by

- securing the processes and the channels used for their interactions
- protecting the objects that they encapsulate against unauthorized access.

Security model

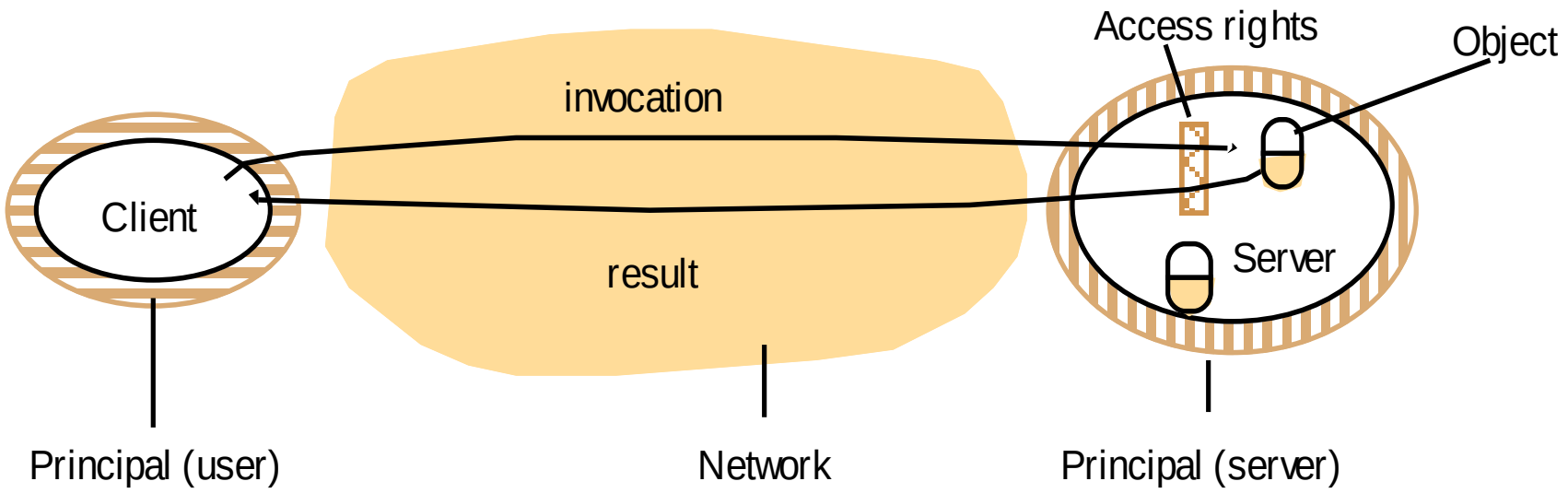
Protecting objects

Consider a server that manages a collection of objects on behalf of some users.

Objects are intended to be used in different ways by different users.

- *Access rights* specify who is allowed to perform the operations of an object
- *Principal* is the authority which is associated with each invocation and each result

Figure 2.12
Objects and principals



Security model

Securing processes and their interactions

Distributed systems are often deployed and used in tasks that are likely to be subject to external attacks by hostile users.

But how can we analyze these threats in order to identify and defeat them?

Security model

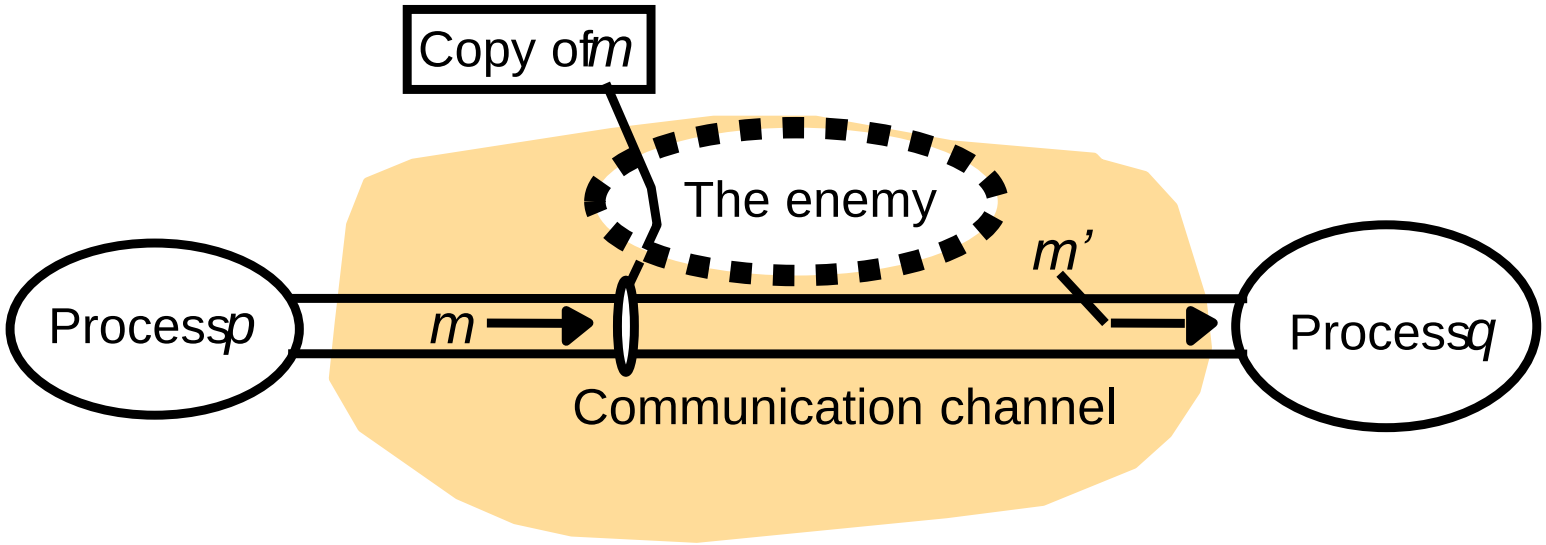
The enemy

To model security threats, we postulate an enemy …

The threats from a potential enemy include

- *threats to processes*
- *threats to communication channels*
- *denial of service*

Figure 2.13
The enemy



Security model

The enemy

Threats to processes:

- a process may receive a message from any other process in the distributed system
- Servers
- Clients

Threats to communication channels: An enemy can copy, alter or inject messages as they travel across the network and its intervening gateways.

Attempt to save copies of messages and to replay them at a later time.

Security model

Defeating security threats

Cryptography and shared secrets

Cryptography is the science of keeping messages secure

Encryption is the process of scrambling a message …

Security model

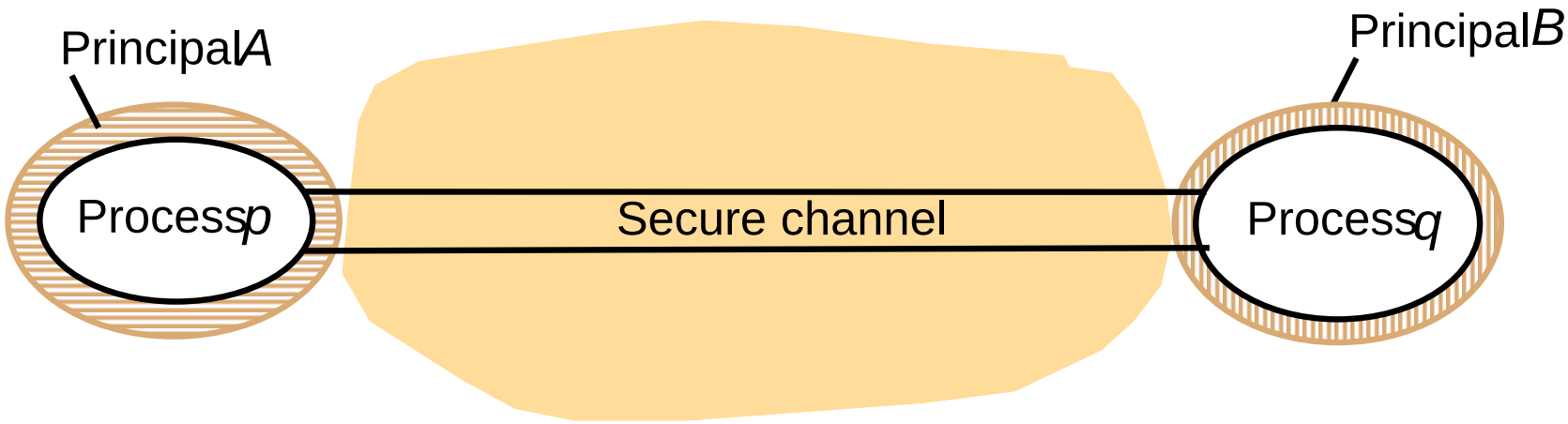
Defeating security threats

Authentication: the use of shared secrets and encryption provides the basis for authentication of messages

Secure channels: Encryption and authentication are used …

A communication channel which connects a pair of processes, each of which acts on behalf of a principal

Figure 2.14
Secure channels



Security model

Other possible threats from an enemy

Denial of service

Mobile code

The uses of security models

The use of security techniques incurs substantial processing and management costs.

Security model provides the basis for the analysis and design of secure systems …

Analysis involves the construction of a threat model …