

Models of distributed software systems

Agenda

Physical models of distributed systems

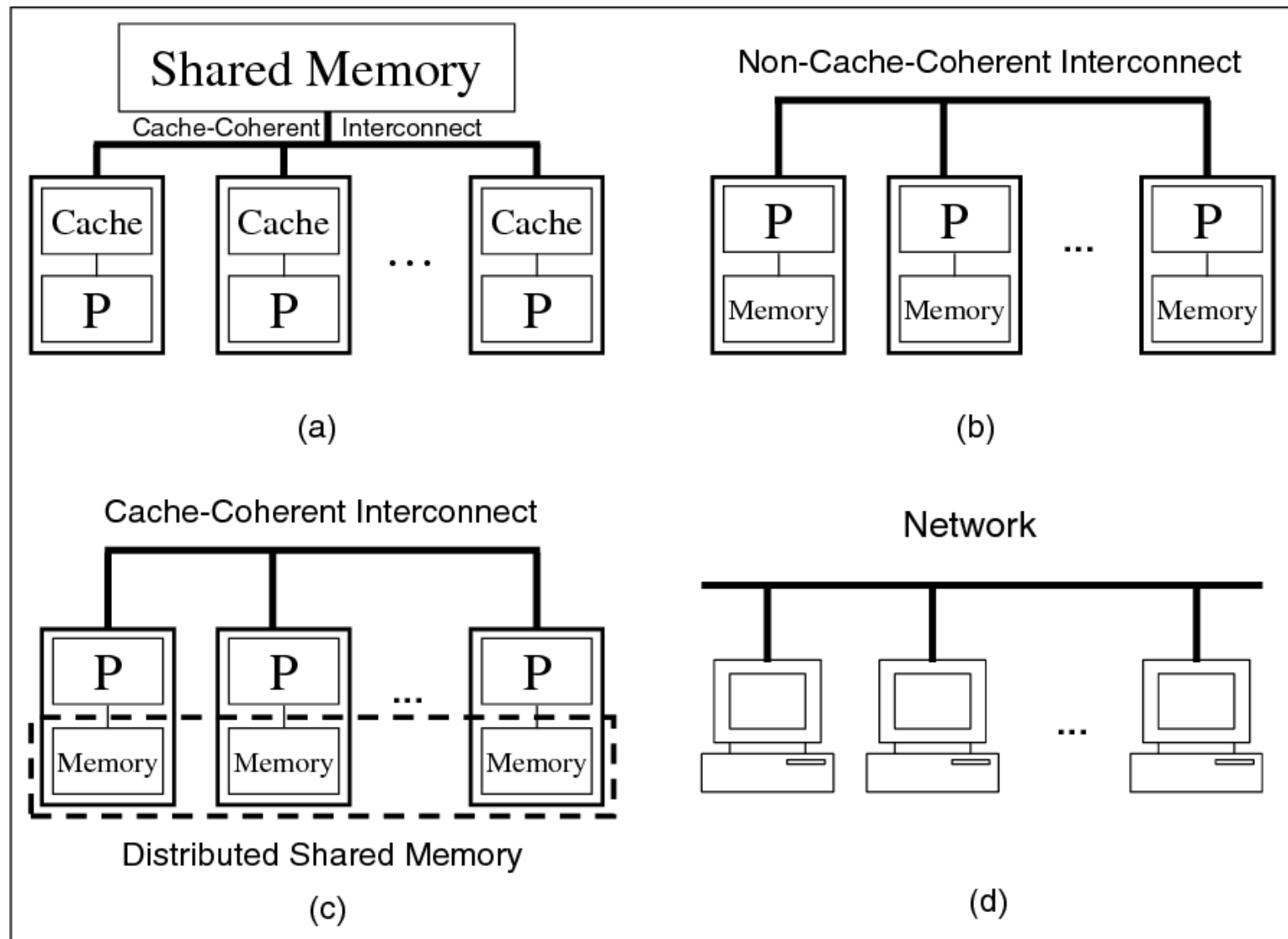
Fundamental models (for distributed algorithms)

Architectural models (eg. for middleware)

Models of distributed systems

1. **Physical models** consider the types of computers and devices that constitute a system and their **interconnectivity**, without details of specific technologies. Eg. Shared memory or no shared memory
2. **Fundamental models** take an **abstract** perspective in order to describe solutions to issues faced by most distributed systems: eg. Models for clock synchronization, or for system partitioning
3. **Architectural models** describe a system in terms of the **communication tasks** performed by its computational elements (computers, devices or network interconnections);
Client-server and *peer-to-peer* are two of the most commonly used forms of architectural models for distributed systems.

Physical models: Shared memory vs distributed memory



The machine in (a) has physically shared memory, whereas the others have distributed memory. However, the shared memories in (c) are accessible to all processors₄

Fundamental Models: time and interactions

How do we handle time?

Do we need in our design time limits or bounds on process execution, message delivery, and clock drifts?

Answers to these questions are very different in the following two cases:

- Synchronous distributed systems
- Asynchronous distributed systems

Synchronous Distributed Systems

Main features:

- Lower and upper bounds on execution time of processes can be set.
- Transmitted messages are received within a known bounded time.
- Drift rates between local clocks have a known bound.

Important consequences:

1. In a synchronous system we can exploit a notion of global physical time (with some known relative precision depending on the drift rate).
2. Only synchronous systems are predictable in terms of timing.
Only such systems can be used for hard real-time applications.
3. In a synchronous system it is possible and safe to use timeouts in order to detect failures of a process or communication link.

REMARK It is difficult and expensive to implement synchronous systems

Asynchronous Distributed Systems

Many distributed systems (including those on the Internet) are asynchronous:

- No bound on process execution time (nothing can be assumed about speed, load, reliability of computers).
- No bound on message transmission delays (nothing can be assumed about speed, load, reliability of interconnections)
- No bounds on drift rates between local clocks.

Asynchronous Distributed Systems

Many distributed systems (including those on the Internet) are asynchronous:

- ❑ No bound on process execution time (nothing can be assumed about speed, load, reliability of computers).
- ❑ No bound on message transmission delays (nothing can be assumed about speed, load, reliability of interconnections)
- ❑ No bounds on drift rates between local clocks.

Important consequences:

1. In an asynchronous system there is no global time.
Reasoning can be only in terms of logical time.
2. Asynchronous systems are unpredictable in terms of timing.
3. Timeouts can be used, but are more challenging to exploit efficiently than in synchronous systems

Asynchronous Distributed Systems

Asynchronous systems are widely and successfully used in practice.

In distributed programming practice timeouts are used with asynchronous systems for *failure detection*.

However, additional measures have to be applied in order to avoid duplicated messages, duplicated execution of operations, etc.

Special theory models for distributed systems

In a distributed system:


- there is no global time
- all communication is by means of **messages**. They are affected by **delays**, can suffer from **failures** and are vulnerable to **security** attacks

These issues are addressed by three special models:

1. An **interaction** model deals with **performance** and with the difficulty of setting time limits to some operations, for example for message delivery.
Main models: synchronous, asynchronous
2. A **fault** (or failure) model aims at a precise specification of the faults that can be exhibited by processes and communication channels. It defines **reliable communication** and correct processes.
3. A **security** model describes some possible threats to processes and communication channels. It introduces the concept of a **secure channel**, which is secure against those threats.

Fault Models

What kind of faults can occur and what are their effects?

- ❑ Omission faults
 - ❑ Arbitrary faults
 - ❑ Timing faults
-
- Faults can occur both in processes and communication channels.
The reason can be both software and hardware.
 - Fault models are needed in order to build systems with predictable behaviour in case of faults (systems which are fault tolerant).
 - A *fault tolerant* system will function according to the predictions, only as long as the real faults behave as defined by the “fault model”. Otherwise 

Omission Faults (Fail Stop Model)

A processor or communication channel fails to perform actions it is supposed to do: the particular action is not performed by the faulty component!

- With omission faults:

- If a component is faulty it does not produce any output.
- If a component produces an output, this output is correct.

- With omission faults, in synchronous systems, faults are detected by timeouts.

- Since we are sure that messages arrive inside an interval, a timeout will indicate that the sending component is faulty.

Such a system has a *fail-stop* behaviour.

Arbitrary (Byzantine) Faults

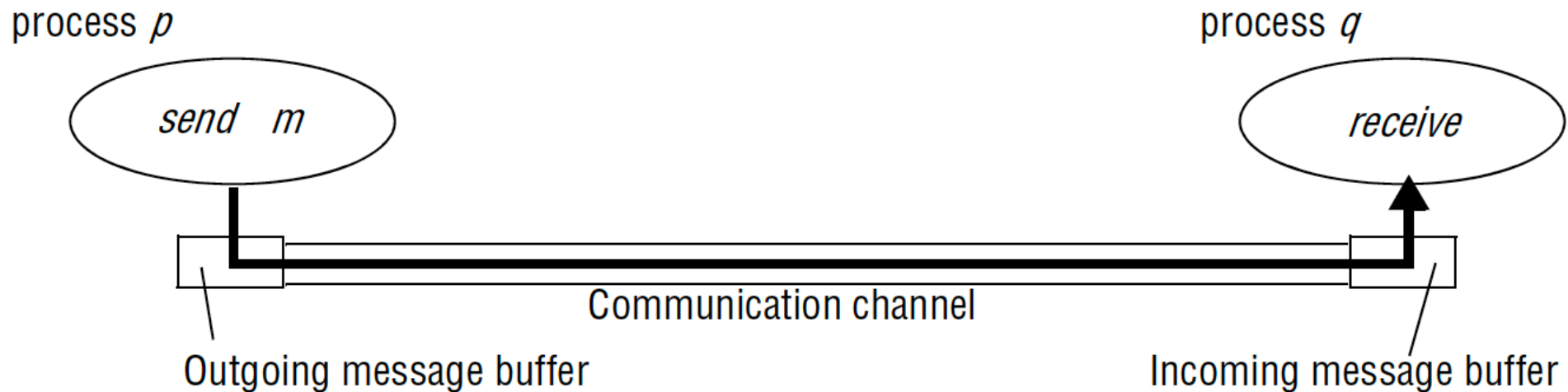
This is the most general and worst possible fault semantics:

- Intended processing steps or communications are omitted or/and unintended ones are executed. Results may not come at all or may come but carry wrong values.



Everything, including the worst, can happen!

Processes and channels



A communication channel produces an **omission failure** if it does not transport a message from p 's outgoing message buffer to q 's incoming message buffer. This is known as 'dropping messages' and is generally caused by lack of buffer space at the receiver or at an intervening gateway, or by a network transmission error, detected by a checksum carried with the message data

Omission and arbitrary failures

<i>Class of failure</i>	<i>Affects</i>	<i>Description</i>
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 <i>send</i> , 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.

Timing Faults

- Timing faults can occur in synchronous distributed systems, where time limits are set to process execution, communications, and clock drifts or clock skew.
- A timing fault results in any of these time limits being exceeded.
- Asynchronous distributed systems cannot be said to have timing failures as guarantees are not provided for response times.

Timing failures

<i>Class of Failure</i>	<i>Affects</i>	<i>Description</i>
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.

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

Any one of these failures may result in responses being unavailable to clients within a specified time interval.

Ordering events in a distributed system

The execution of activities in an asynchronous distributed system can be described in terms of events and their ordering, despite the lack of global clocks

We will be often interested in knowing whether an event (eg.: message sent/message received) at one process occurred before, after or concurrently with another event at another process

Example: consider the following set of exchanges between a group of email users, X, Y, Z and A, on a mailing list:

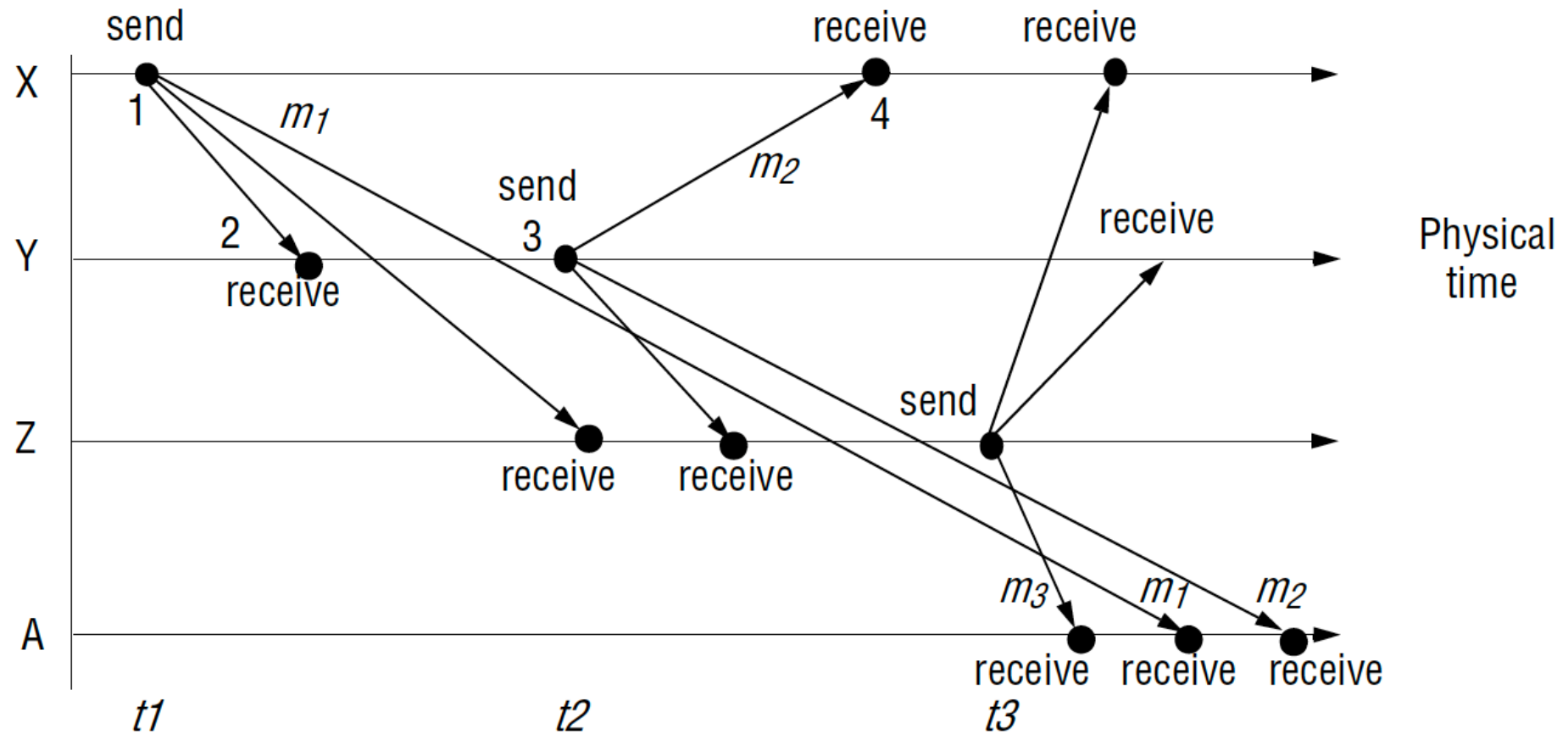
1. User X sends a message with the subject «Meeting.»

2. Users Y and Z reply by sending a message with the subject «Re: Meeting.»

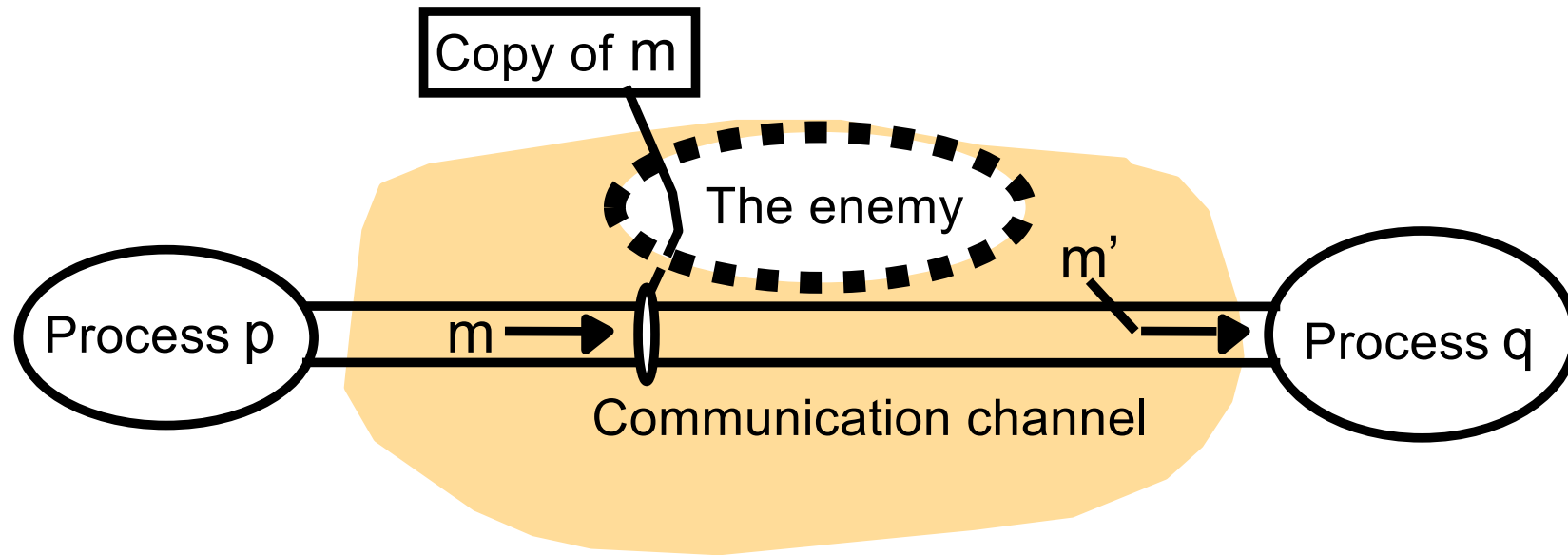
In real time, X's message is sent first, and Y reads it and replies; Z then reads both X's message and Y's reply and sends another reply, which references both X's and Y's messages.

But due to the independent delays in message delivery, the messages may be delivered in a wrong order (see next slide)

Real-time ordering of events

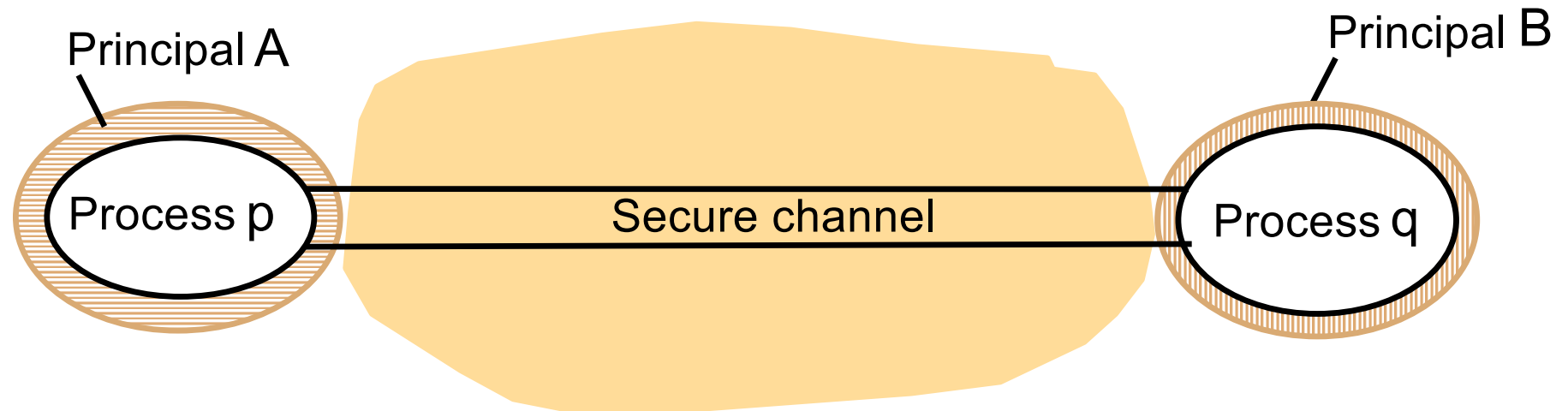


Security models: The enemy



To model and study security threats, we postulate an enemy (adversary) that is capable of sending any message to any process and reading or copying any message sent on a channel between a pair of processes

Secure channels



A **secure channel** (encrypted, authenticated) has the following properties:

- Each of the processes knows reliably the identity of the principal on whose behalf the other process is executing.
- A secure channel ensures the privacy and integrity (protection against tampering) of the data transmitted across it.
- Each message includes a physical or logical timestamp to prevent messages from being replayed or reordered

Models based on architectural styles

An architectural **style** is formulated in terms of (replaceable, composable) components with well-defined interfaces

- the way that components are connected to each other
- the data formats exchanged between components
- how these components and connectors are jointly configured into a system.

Connector

A (software) mechanism that mediates communication, coordination, or cooperation among components.

Example : facilities for (remote) procedure call, messaging, or streaming.

Generations of distributed systems

<i>Distributed systems:</i>	<i>Early</i>	<i>Internet-scale</i>	<i>Contemporary</i>
<i>Scale</i>	Small	Large	Ultra-large
<i>Heterogeneity</i>	Limited (typically relatively homogenous configurations)	Significant in terms of platforms, languages and middleware	Added dimensions introduced including radically different styles of architecture
<i>Openness</i>	Not a priority	Significant priority with range of standards introduced	Major research challenge with existing standards not yet able to embrace complex systems
<i>Quality of service</i>	In its infancy	Significant priority with range of services introduced	Major research challenge with existing services not yet able to embrace complex systems

Coordinable entities in distributed systems

- What entities do they communicate in a distributed system? Eg. Processes? Agents? Objects?
- How do they communicate? which communication paradigm is used? Eg. Channel or ether based?
- What roles and responsibilities do the coordinable entities have in the overall architecture?
- How are they mapped on to the physical distributed infrastructure (what is their deployment)?

Coordinable entities in distributed systems: example

- What entities do they communicate in a distributed system? **Java (remote) objects**
- How do they communicate, or, more specifically, what communication paradigm is used? **RMI middleware**
- What (potentially changing) roles and responsibilities do they have in the overall architecture? **Client-server**
- How are they mapped on to the physical distributed infrastructure (what is their deployment)? **Pack the server code and run it on the remote machine**

(<https://stackoverflow.com/questions/10459057/java-rmi-remote-deployment/11150596>)

Architectural style example: client-server

Components: clients, servers

Data format interchanged: XML or JSON

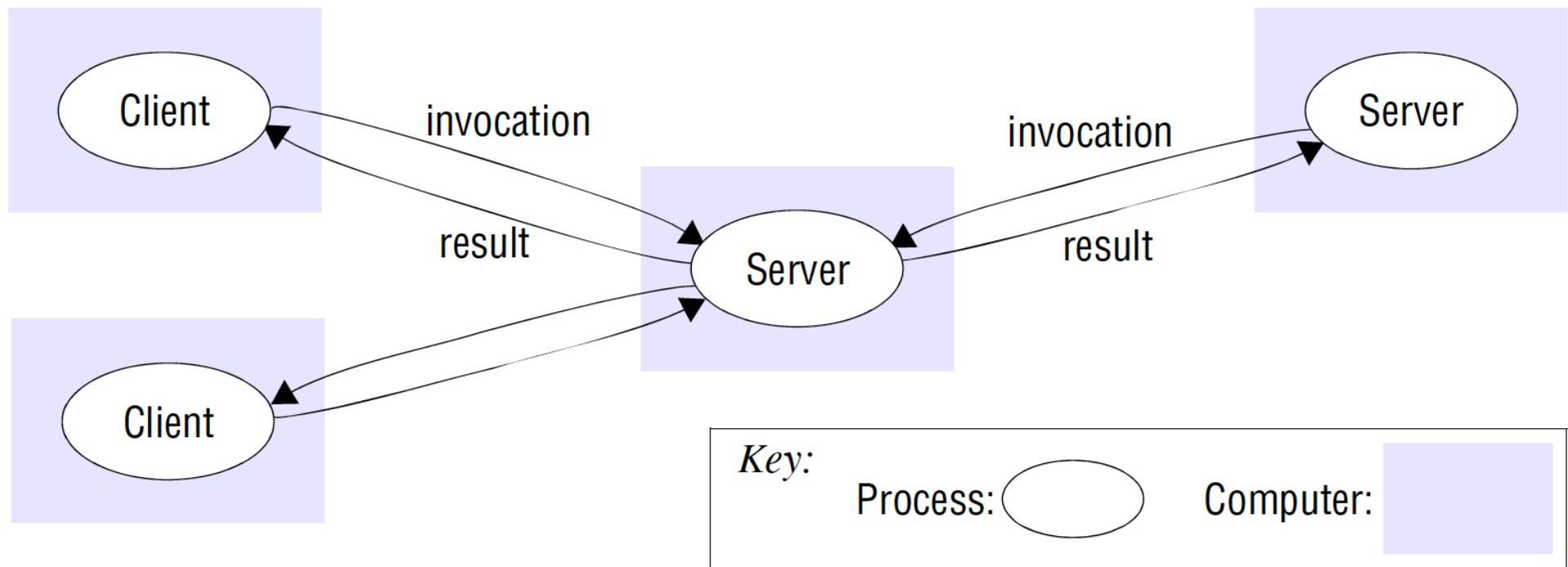
Configuration: (example Web) HTTP + URL

Connector: (example Web) HTTP

Communicating entities and communication paradigms

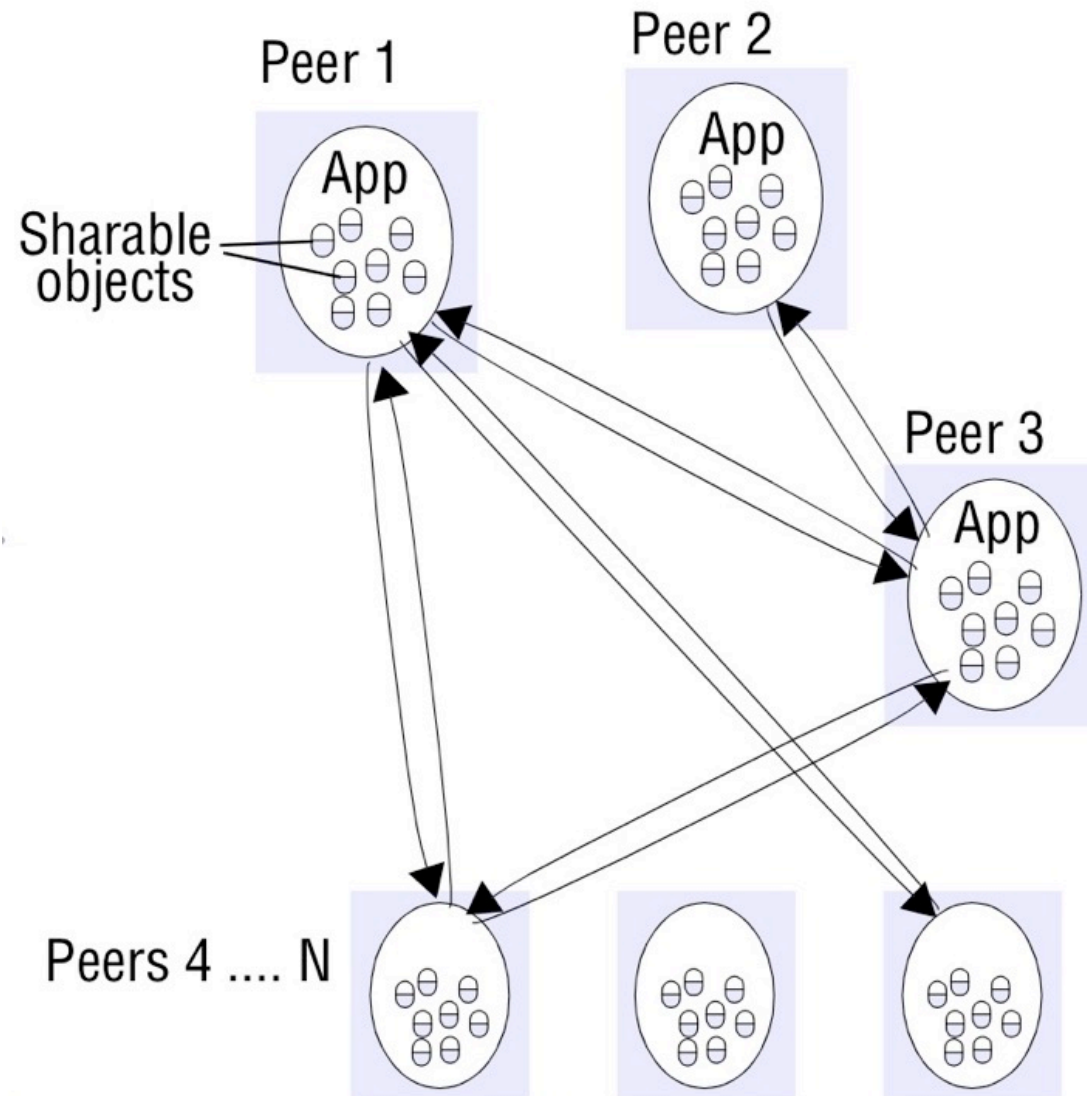
<i>Communicating entities (what is communicating)</i>		<i>Communication paradigms (how they communicate)</i>		
<i>System-oriented entities</i>	<i>Problem- oriented entities</i>	<i>Interprocess communication</i>	<i>Remote invocation</i>	<i>Indirect communication</i>
Nodes	Objects	Message passing	Request- reply	Group communication
Processes	Components	Sockets	RPC	Publish-subscribe
	Web services	Multicast	RMI	Message queues
				Tuple spaces
				DSM

Clients invoke individual servers

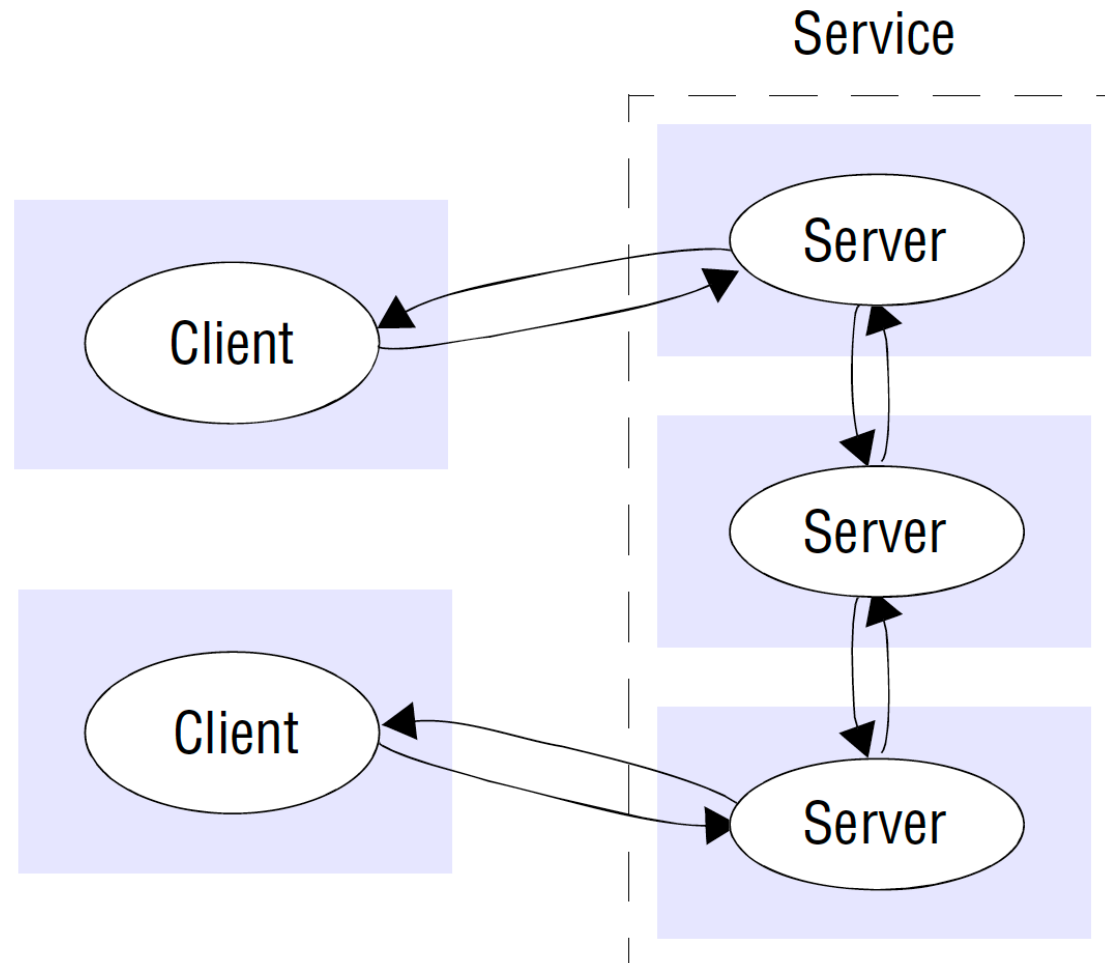


a web server is often a client of a local file server that manages the files in which the web pages are stored. Web servers and most other Internet services are clients of the DNS service, which translates Internet domain names to network addresses

Peer-to-peer architecture

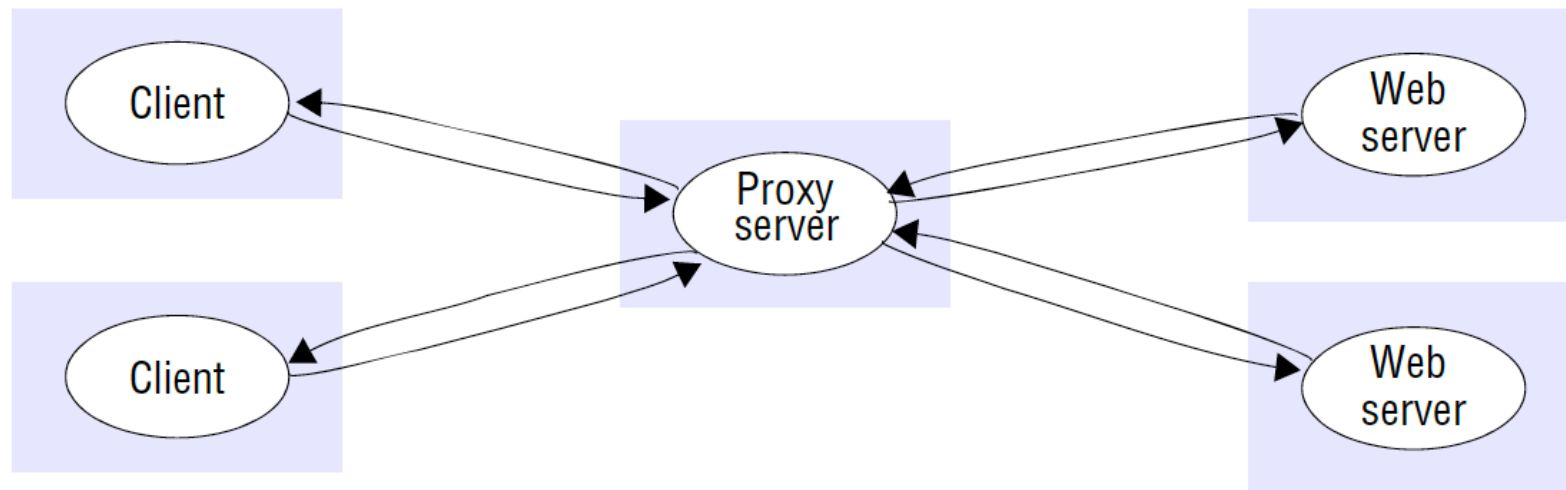


A service provided by multiple servers



The servers may partition the set of objects on which the service is based and distribute those objects between themselves, or they may maintain replicated copies of them on several hosts.

Web proxy server

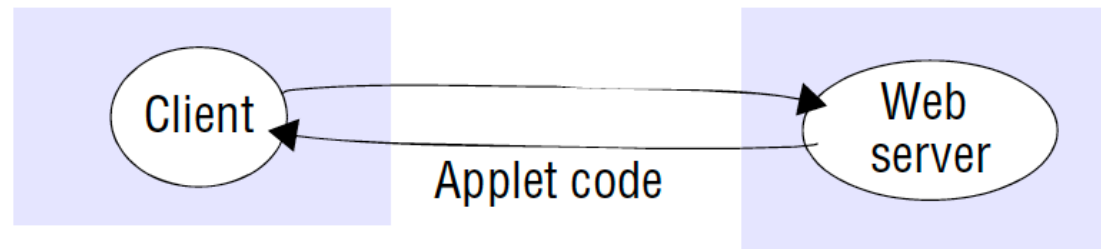


Web proxy servers provide a shared cache of web resources for the client machines at a site or across several sites.

The purpose of proxy servers is to increase the availability and performance of the service by reducing the load on both the network and the servers.

Web applets

a) client request results in the downloading of applet code



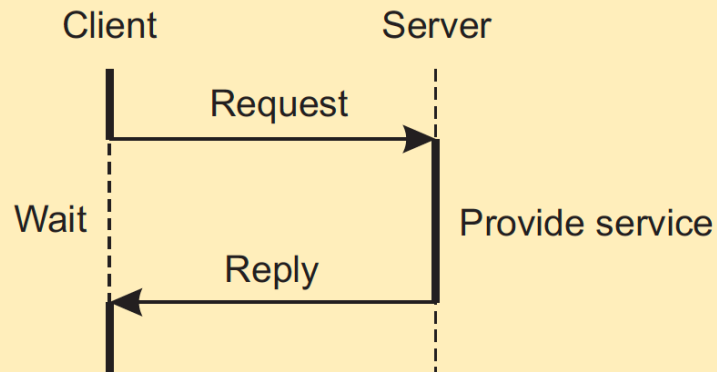
b) client interacts with the applet



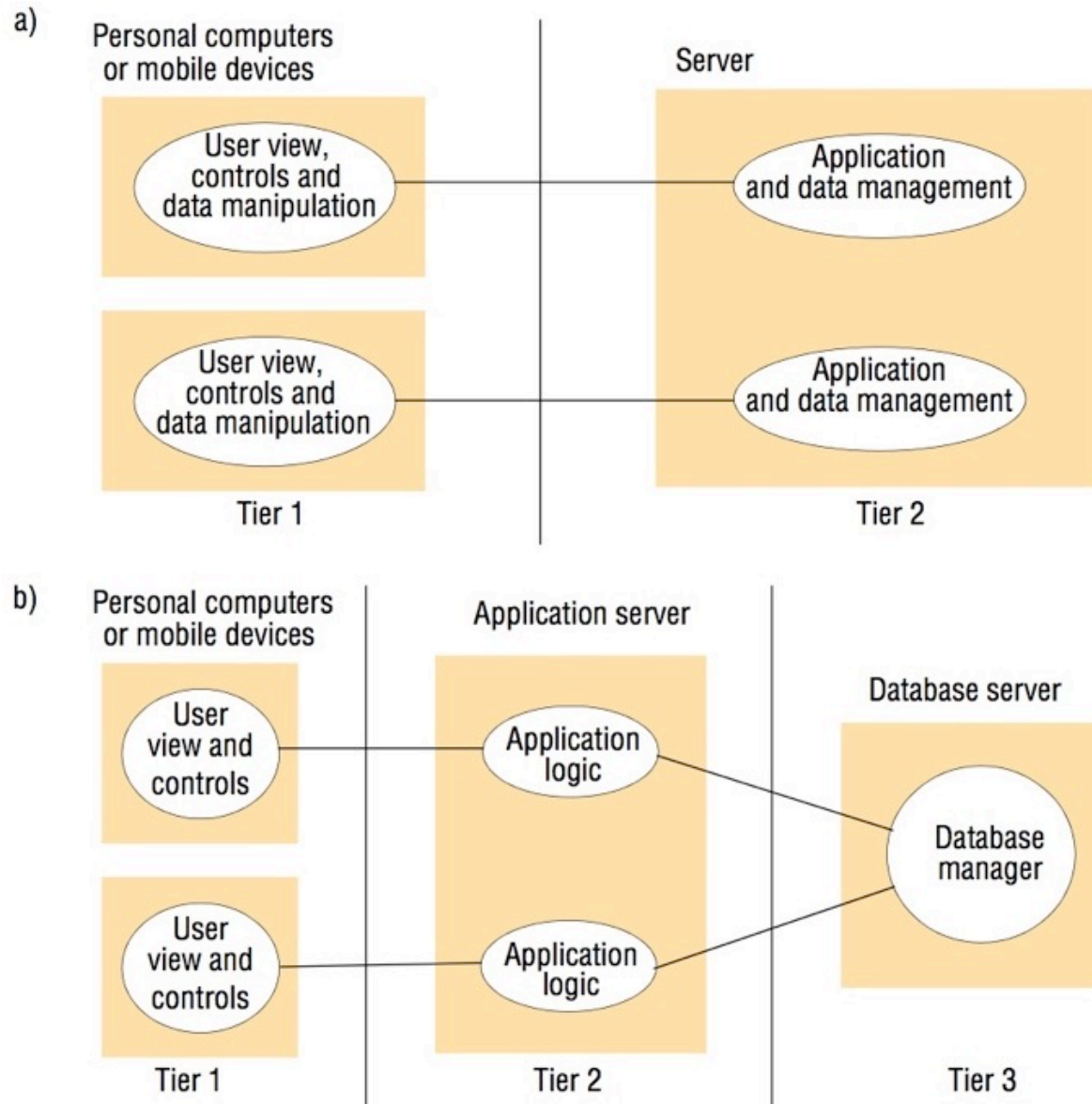
Basic client-server style

Characteristics:

- There are processes offering services (**servers**)
- There are processes that use services (**clients**)
- Clients and servers can be on different machines
- Clients follow request/reply model with respect to using services



Two-tier and three-tier architectures

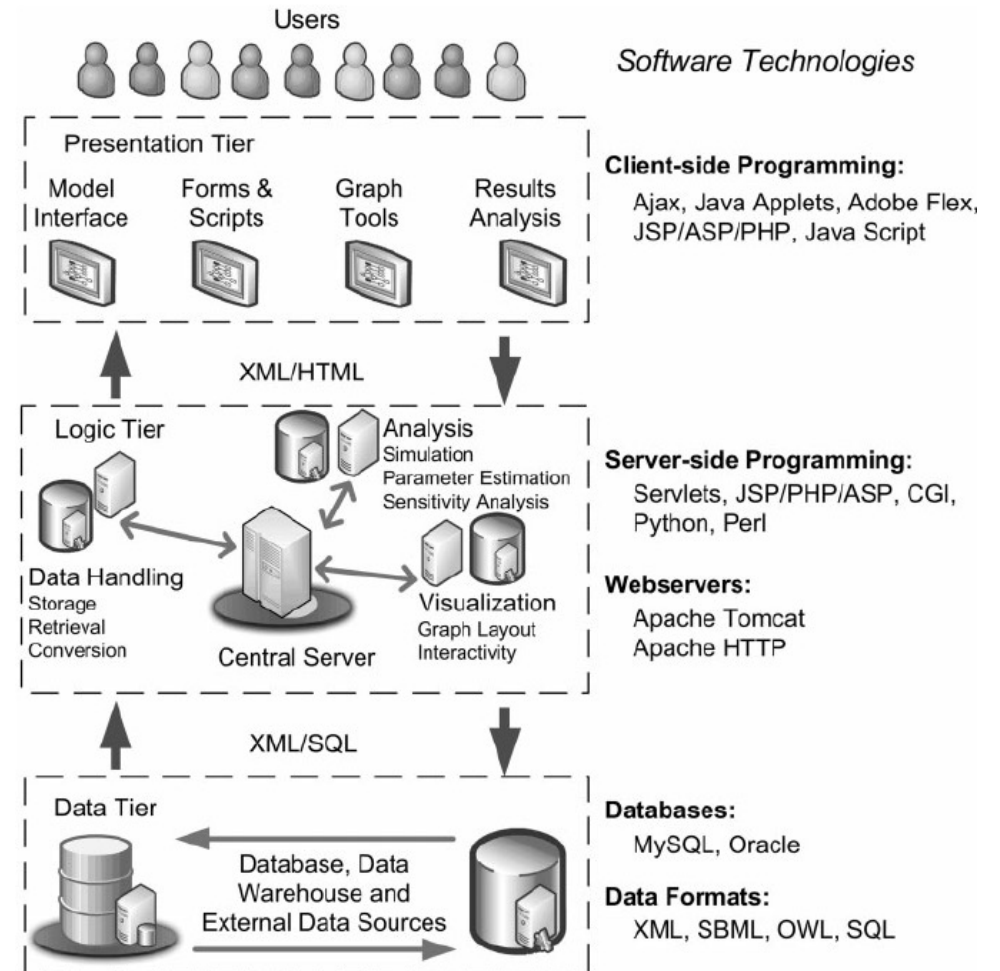


3- tier architecture with AJAX

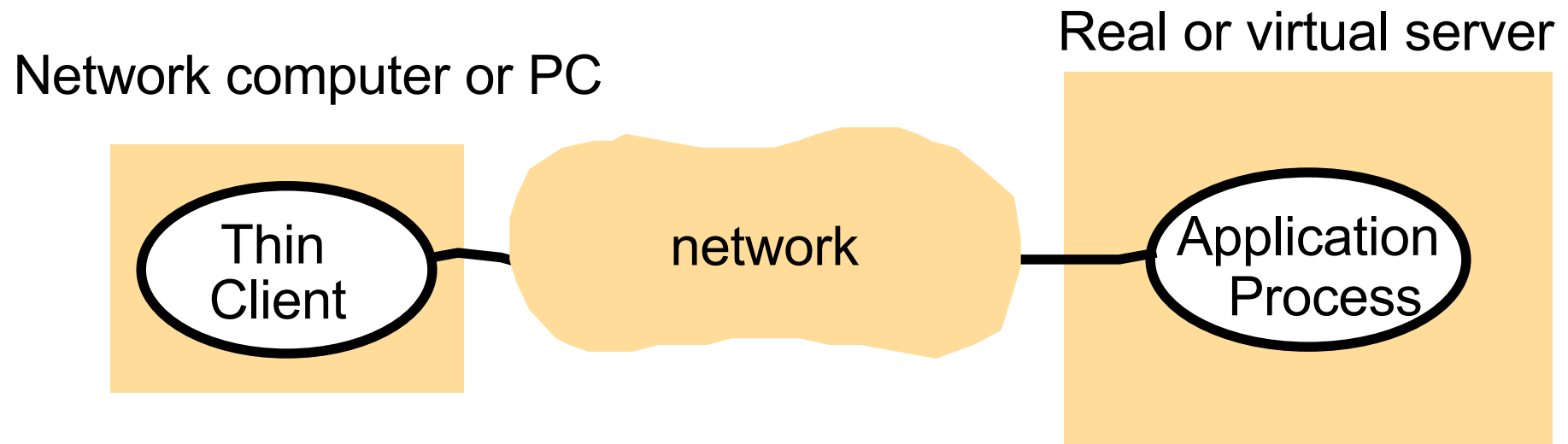
The 3-tier architecture is used in most web applications where AJAX is used client-side

This structure is the 'glue' that supports the construction of such applications

it provides a communication mechanism enabling front-end components running in a browser to issue requests and receive results from back-end components running on a server

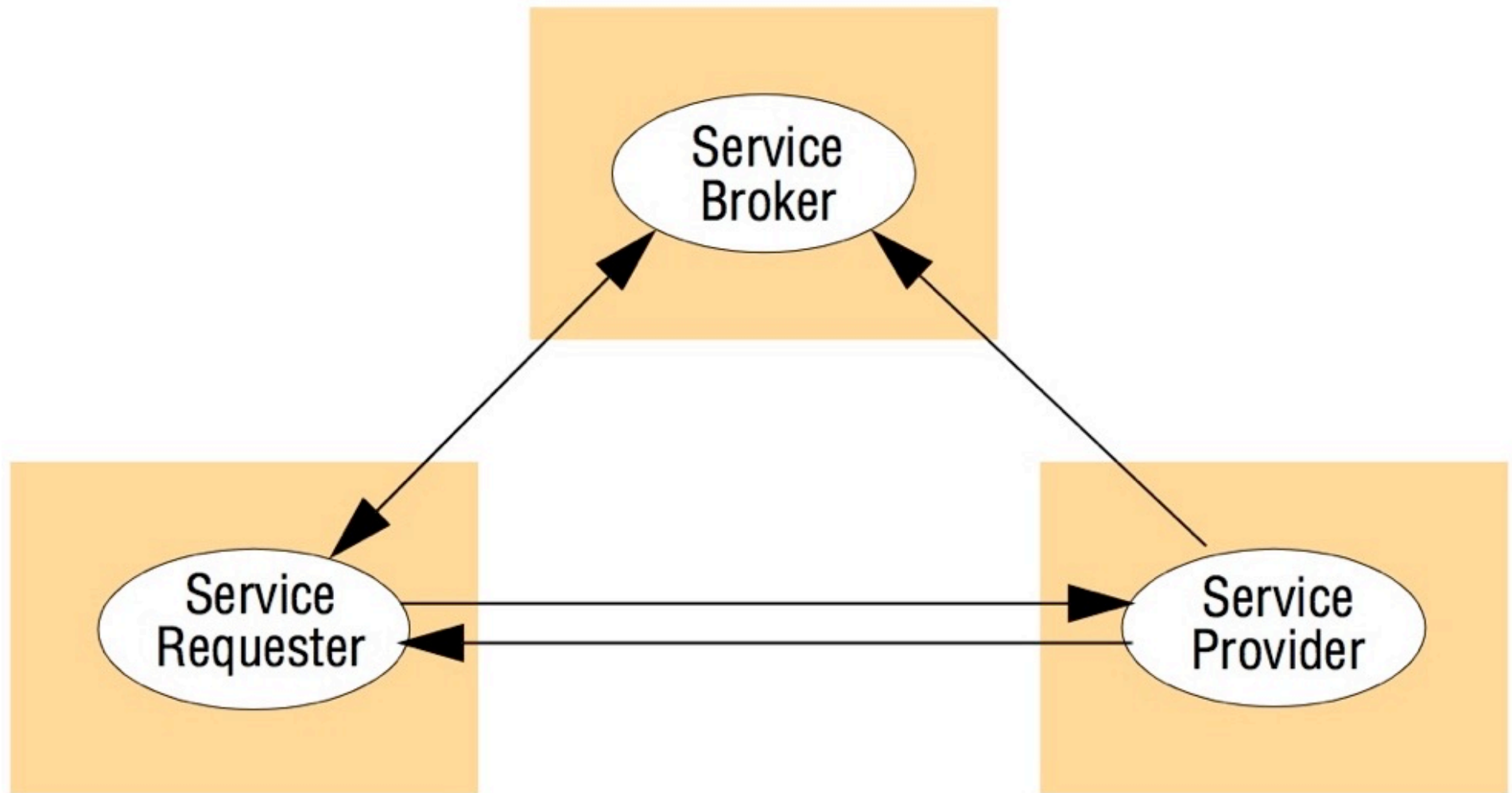


Thin clients and compute servers



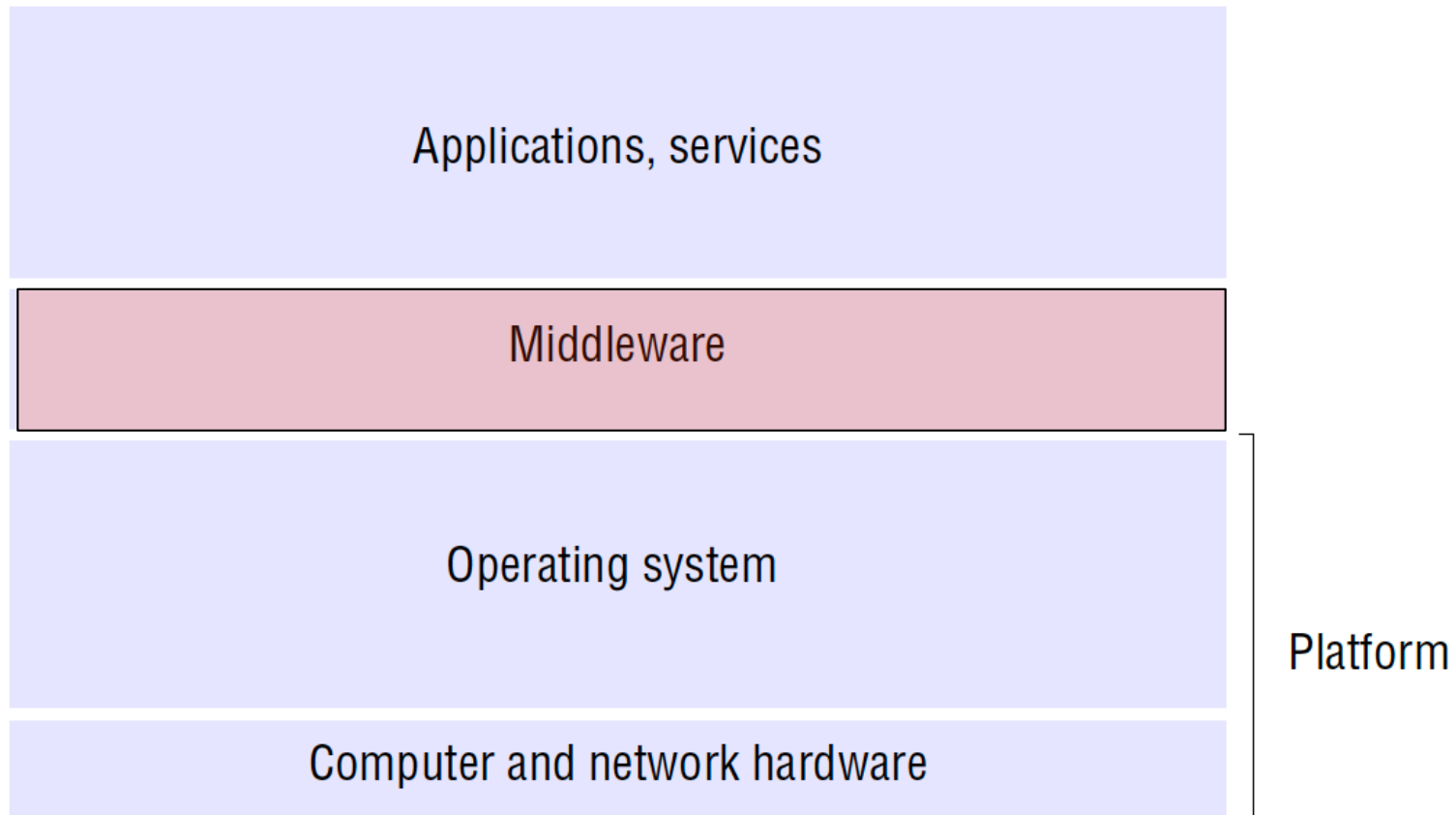
The trend in distributed computing is towards moving complexity away from the end-user device towards services in the Internet
This is most apparent in the move towards cloud computing

The web service architectural pattern for middleware



This middleware pattern is replicated in many areas of distributed systems, for example with the registry in Java RMI and the naming service in CORBA

Software and hardware service layers in distributed systems



Categories of middleware

Category	subcategory	Example systems
Distributed objects	Standard	RM-ODP
	Platform	CORBA
	Platform	JavaRMI
Distributed components	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	Enterprise Java Beans
	Application servers	CORBA component model
	Application servers	Jboss, Wildfly
Publish-subscribe systems		CORBA event service
		Apache Camel
Message queues		Websphere MQ
		RabbitMQ
Web services	Web services	Apache Axis
	Grid services	The Globus Toolkit
Peer-to-peer	Routing overlays	Pastry, Tapestry
	Distributed ledger	Apache ZooKeeper
	Application-specific	Squirrel
	Application-specific	Oceanstore
	Application-specific	Ivy
	Application-specific	Gnutella
Data streaming	Platform	Apache Kafka, Pulsar

RESTful architectures

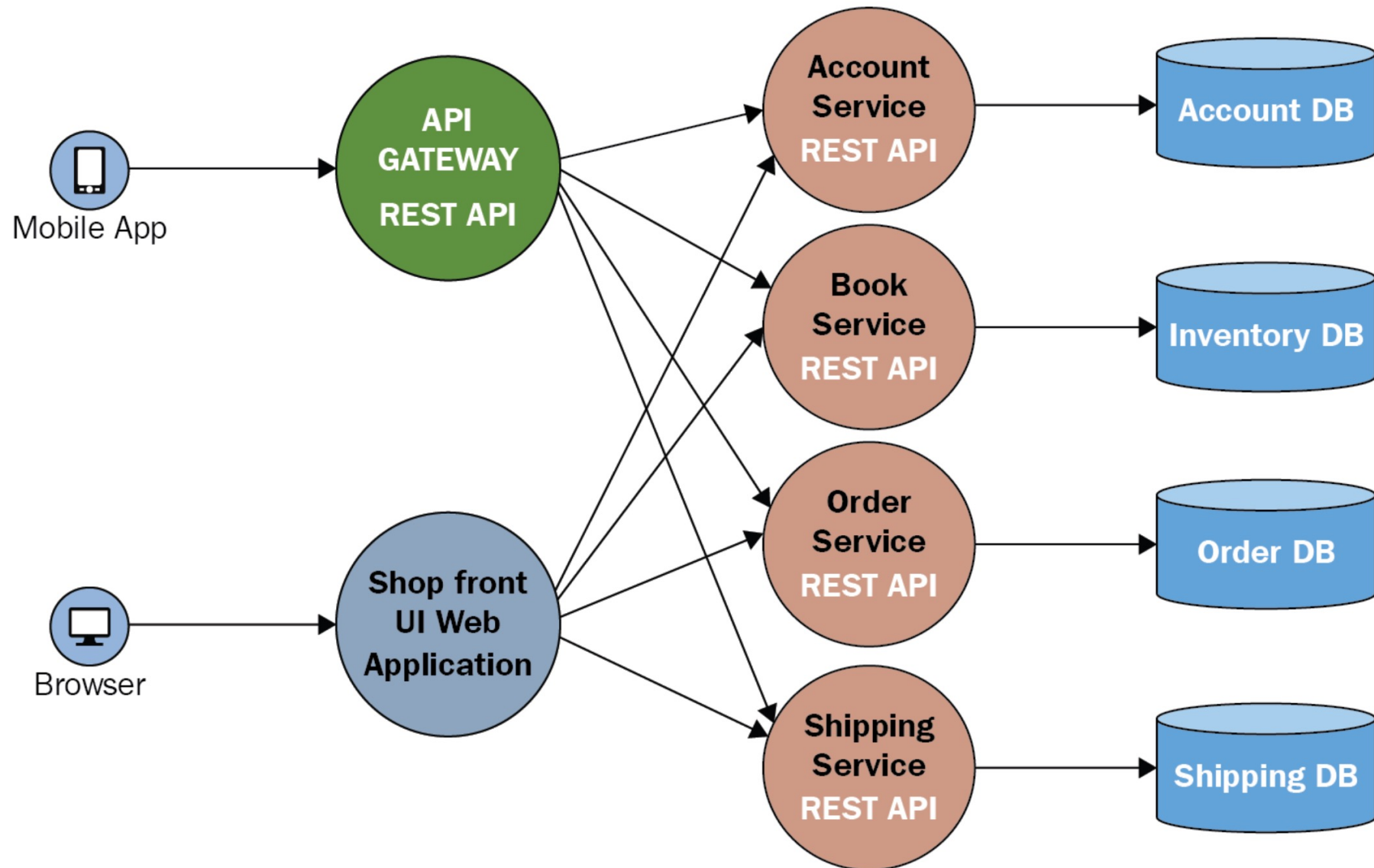
RESTful architectures view a distributed system as a collection of resources, individually managed by components. Resources may be added, removed, retrieved, and modified by (remote) applications.

- 1 Resources are identified through a single naming scheme
- 2 All services offer the same interface
- 3 Messages sent to or from a service are fully self-described
- 4 After executing an operation at a service, that component forgets everything about the caller

Basic operations

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource in some representation
DELETE	Delete a resource
POST	Modify a resource transferring to a new state

Microservice architecture



How to scale a web site

Suppose we have two services: inventory service (handles product descriptions and inventory management) and user service (handles user information, registration, login, etc.).

Step 1 - With the growth of the user base, one single application server cannot handle the traffic anymore. We put the application server and the database server into two separate servers.

Step 2 - The business grows, and a single application server is no longer enough. Deploy a cluster of application servers.

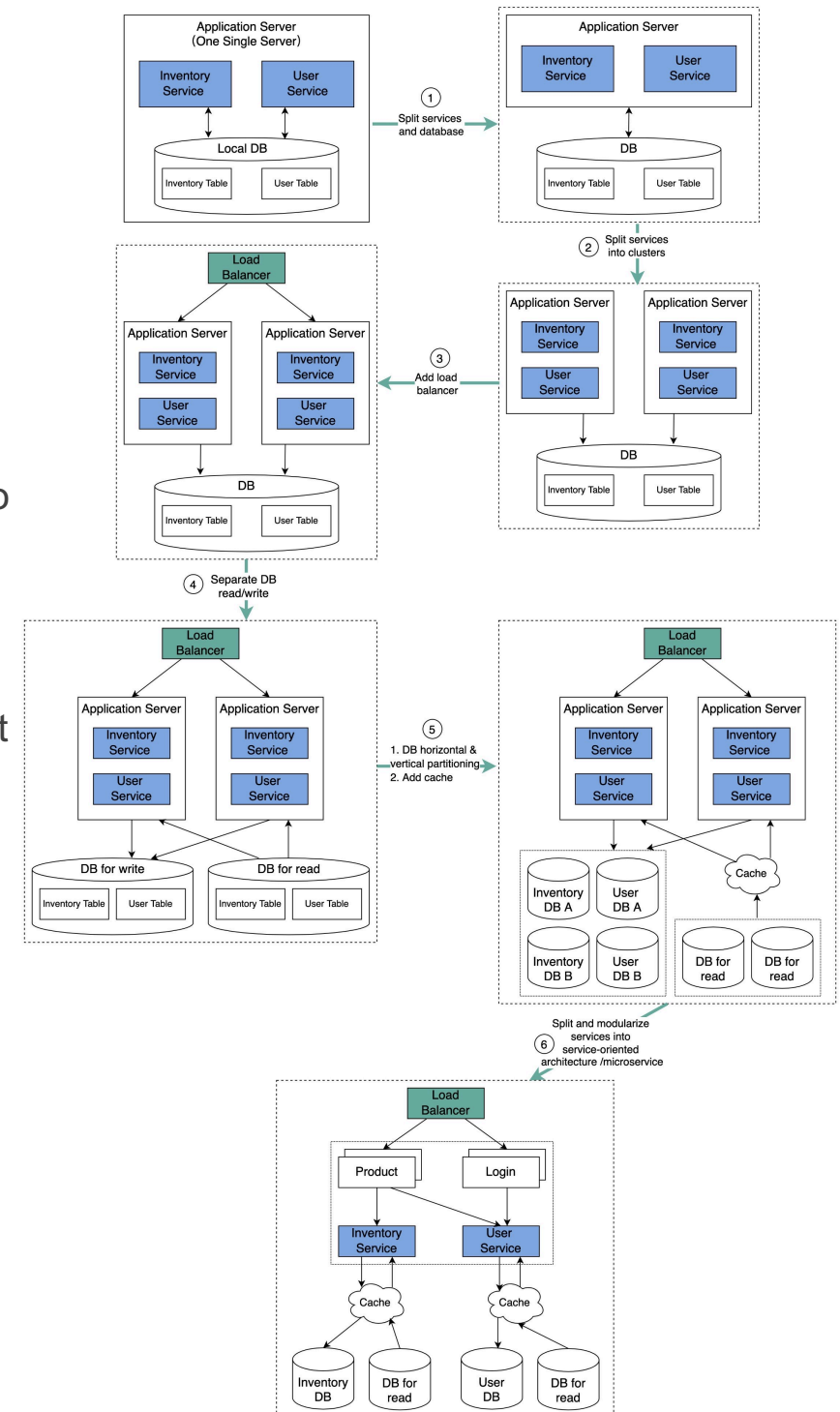
Step 3 - Now the incoming requests have to be routed to multiple application servers, how can we ensure each application server gets an even load? The load balancer handles this nicely.

Step 4 - With the business continuing to grow, the database might become the bottleneck. To mitigate this, we separate reads and writes in a way that frequent read queries go to read replicas. With this setup, the throughput for the database writes can be greatly increased.

Step 5 - Suppose the business continues to grow. One single database cannot handle the load on both the inventory table and user table. We have a few options:

1. Vertical partition. Adding more power (CPU, RAM, etc.) to the database server. It has a hard limit.
2. Horizontal partition by adding more database servers.
3. Adding a caching layer to offload read requests.

Step 6 - Now we modularize the functions into different services. The architecture becomes service-oriented / microservice.



Conclusions

Distributed systems can be studied via the **architectural patterns** (abstractions and models) used to design and implement them

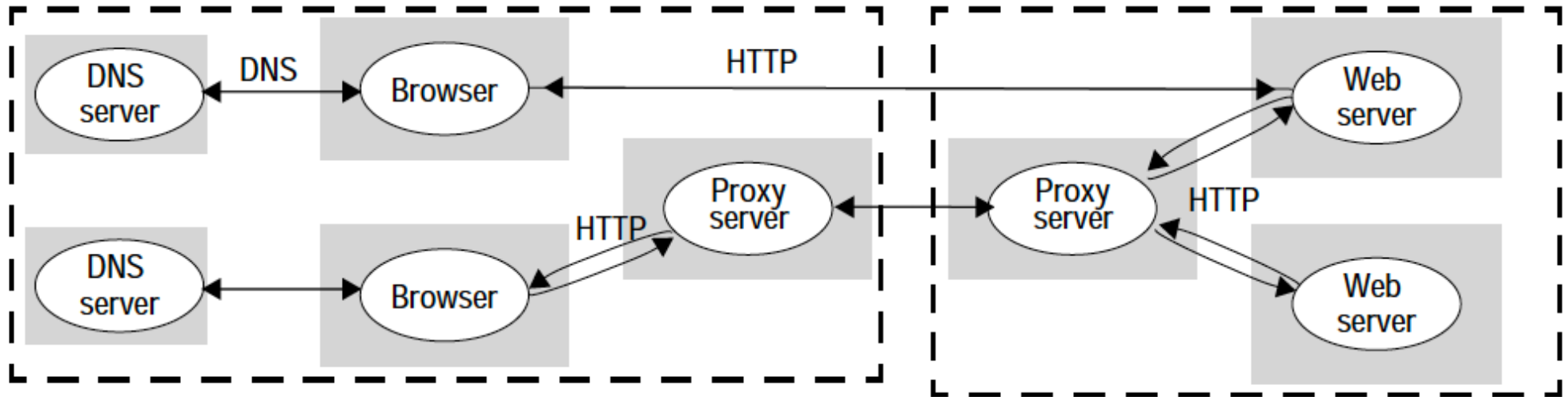
Client server and peer to peer are the most important architectural patterns for distributed systems

Microservices and serverless architectures are currently popular patterns for cloud computing

Exercises

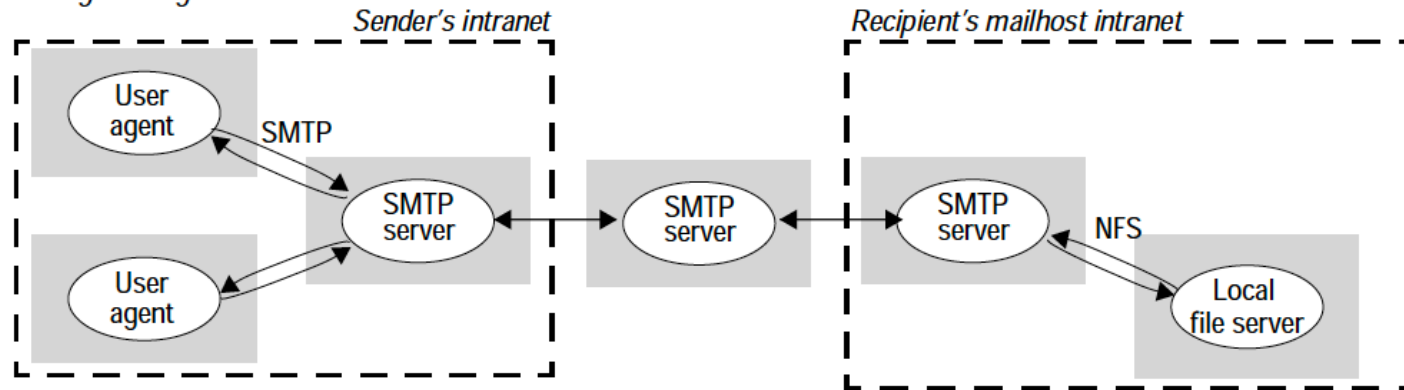
1. Describe the *software architecture* of the major Internet services for
 - Web,
 - Mail
 - news
2. Describe how servers cooperate in providing these services
3. How do the services exploit the partitioning and/or replication (or caching) of data amongst servers?

Web

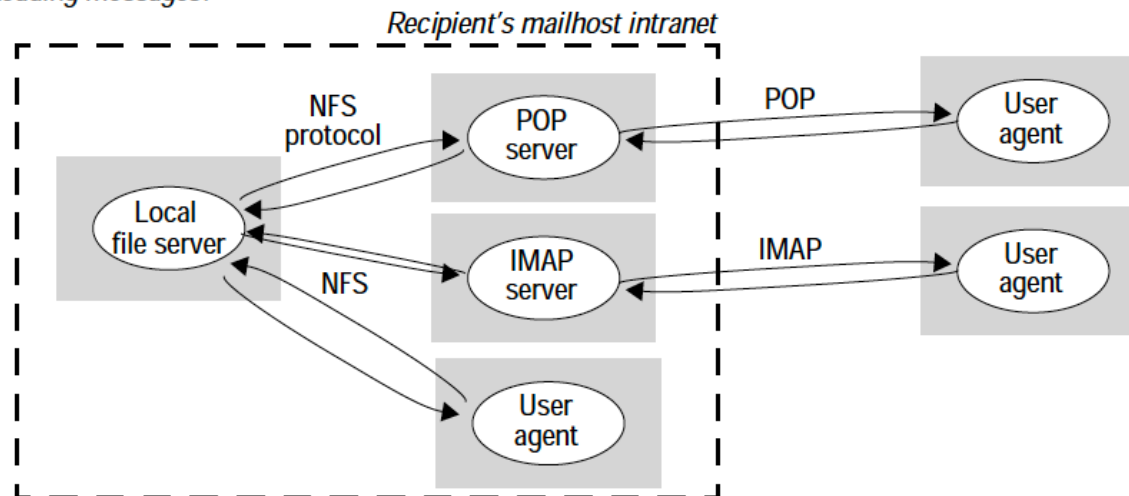


Mail

Sending messages:

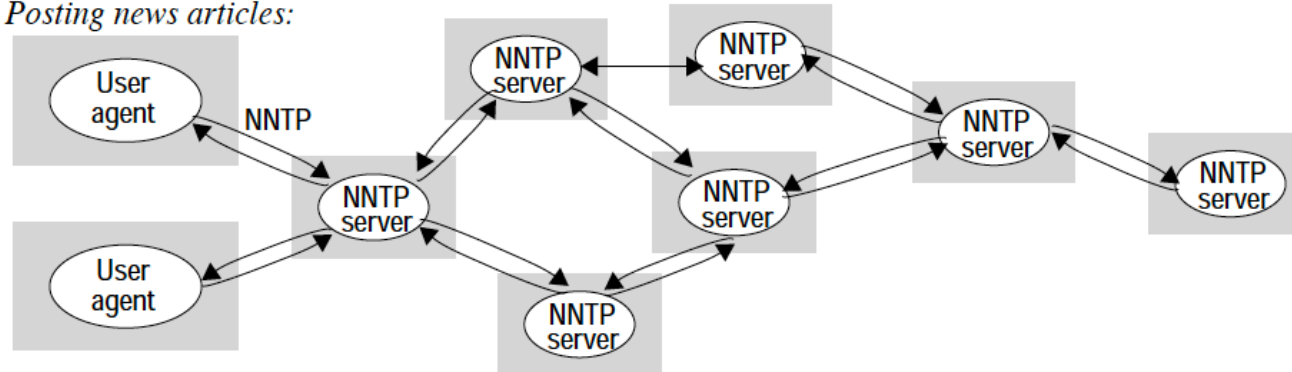


Reading messages:

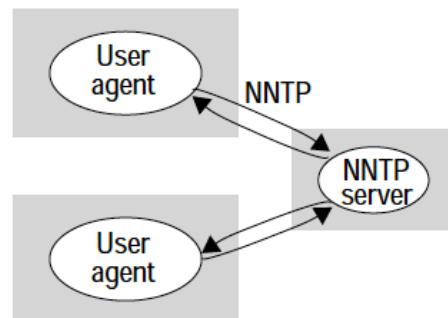


Netnews

Posting news articles:



Browsing/reading articles:



Server cooperation

Web: Web servers cooperate with Proxy servers to minimize network traffic and latency. Responsibility for consistency is taken by the proxy servers - they check the modification dates of pages frequently with the originating web server.

Mail: SMTP servers do not necessarily hold mail delivery routing tables to all destinations. Instead, they simply route messages addressed to unknown destinations to another server that is likely to have the relevant tables.

Netnews: All NNTP servers cooperate to provide the newsfeed mechanism.

Replication

Web: Web page masters are held in a file system at a single server.

The pages are therefore partitioned amongst many web servers.

Replication is not a part of the web protocols, but a heavily-used web site may provide servers with identical copies of the relevant file system. HTTP requests can be multiplexed amongst the identical servers using the DNS load sharing mechanism. In addition, web proxy servers support replication through the use of cached replicas of recently-used pages and browsers support replication by maintaining a local cache of recently accessed pages.

Mail: Messages are stored only at their destinations. That is, the mail service is based mainly on partitioning, although a message to multiple recipients is replicated at several destinations.

Netnews: Each group is replicated only at sites requiring it