Slides for Chapter 2:
System Models

From **Coulouris, Dollimore and Kindberg**

Distributed Systems:

Concepts and Design

How to model a distributed system?

- Systems of different type still share many properties and give rise to similar design problems
- Models help to describe and deal with design issues
- There are (at least) three kind of models
  - **Physical models**: hardware compositions of devices, networks, etc
  - **Architectural models**: computational and communication tasks performed by system elements
  - **Fundamental models**: abstract models for examining individual aspects, in a detailed and formal setting
    - **Interaction models**: about structure and sequence of communications
    - **Failure models**: about failures, and correct behaviour thereof
    - **Security models**: about how the system is protected against attacks
  - see: Semantica e Concorrenza

Physical models

- Basic physical model: networked computers/ devices which communicate through message passing on a network
- Early distributed systems (’70-’80): first LANs, shared printers, file servers, email
- Internet-scale systems (‘90s): networks of networks. Need for cross-platform standard middleware: first CORBA, now web services
- Contemporary distributed systems:
  - mobile, ubiquitous systems
  - cloud computing and virtualized system

Generations of distributed systems

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

Architectural Models

- System is described in terms of separated components and their interrelationships
- An architectural design is also a reference framework for future extensions (read: it is difficult to move from an architecture to another, often a complete refectory is needed)
- Choosing the right architecture is very important
  - a wrong design can hinder important features
- Often a trade-off among different parameters (e.g. performance vs effectiveness, scalability vs redundancy...)

- Before deciding the architecture, we have to identify the architectural elements (the building blocks)
  - Communication Entities
  - Communication Paradigms
  - Roles and responsibilities
  - Placement (i.e. where entities are physically located)

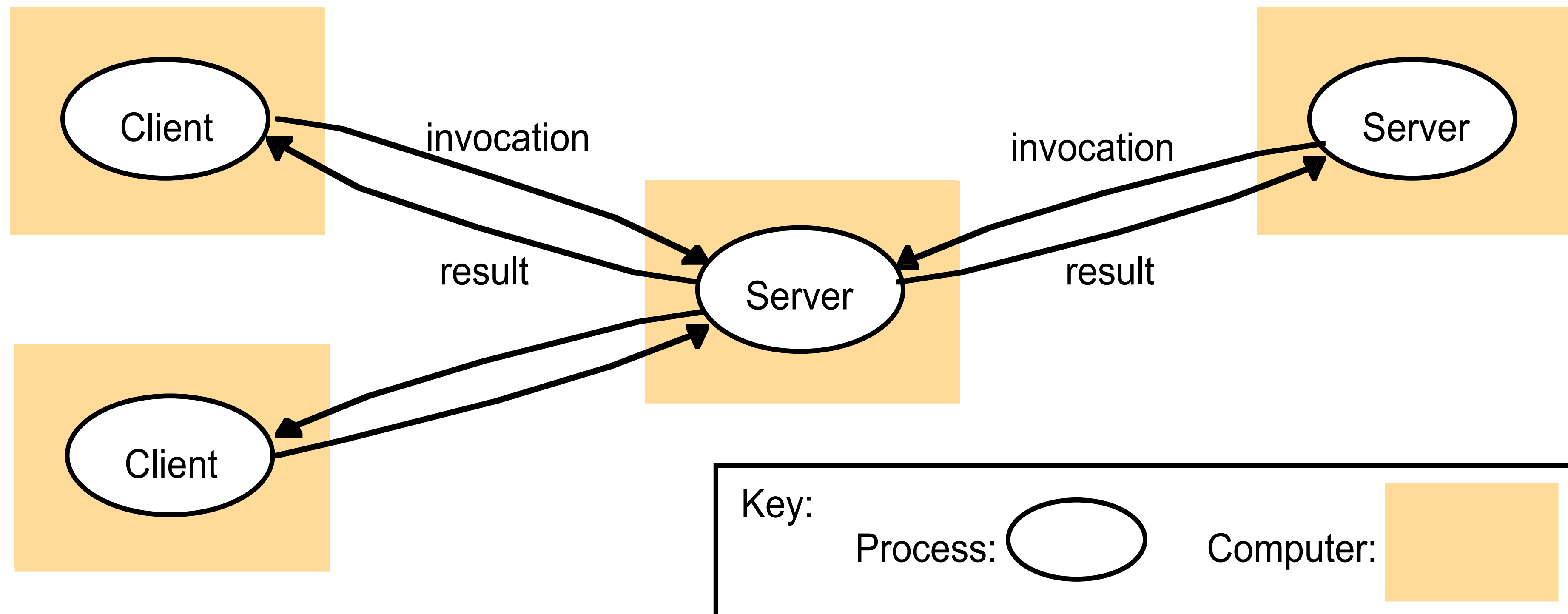
- Communication Entities from a System Perspective
  - whole systems running just one process (e.g. sensors), called *nodes*
  - or processes in a normal OS
  - or single threads
- Communication Entities from a Programmer (i.e. Problem) Perspective
  - Objects, like in Java, CORBA, D-Com...
  - Components (objects with contracts)
  - Web services: encapsulating services in standardized interfaces, allowing loosely coupled integration

Communication paradigms

- Interprocess communication
  - low-level support, like sockets, multicast, broadcast
- Remote Invocation: two-way exchange (call-reply)
  - Request-Reply (e.g., DNS, HTTP)
  - Remote Procedure Calls
  - Remote Method Invocation
- Group Communication
  - Publish-subscribe (e.g. FIX)
  - Message queues (e.g. AMQP)
  - Tuple spaces (e.g. JavaSpace)
  - Distributed Shared Memory

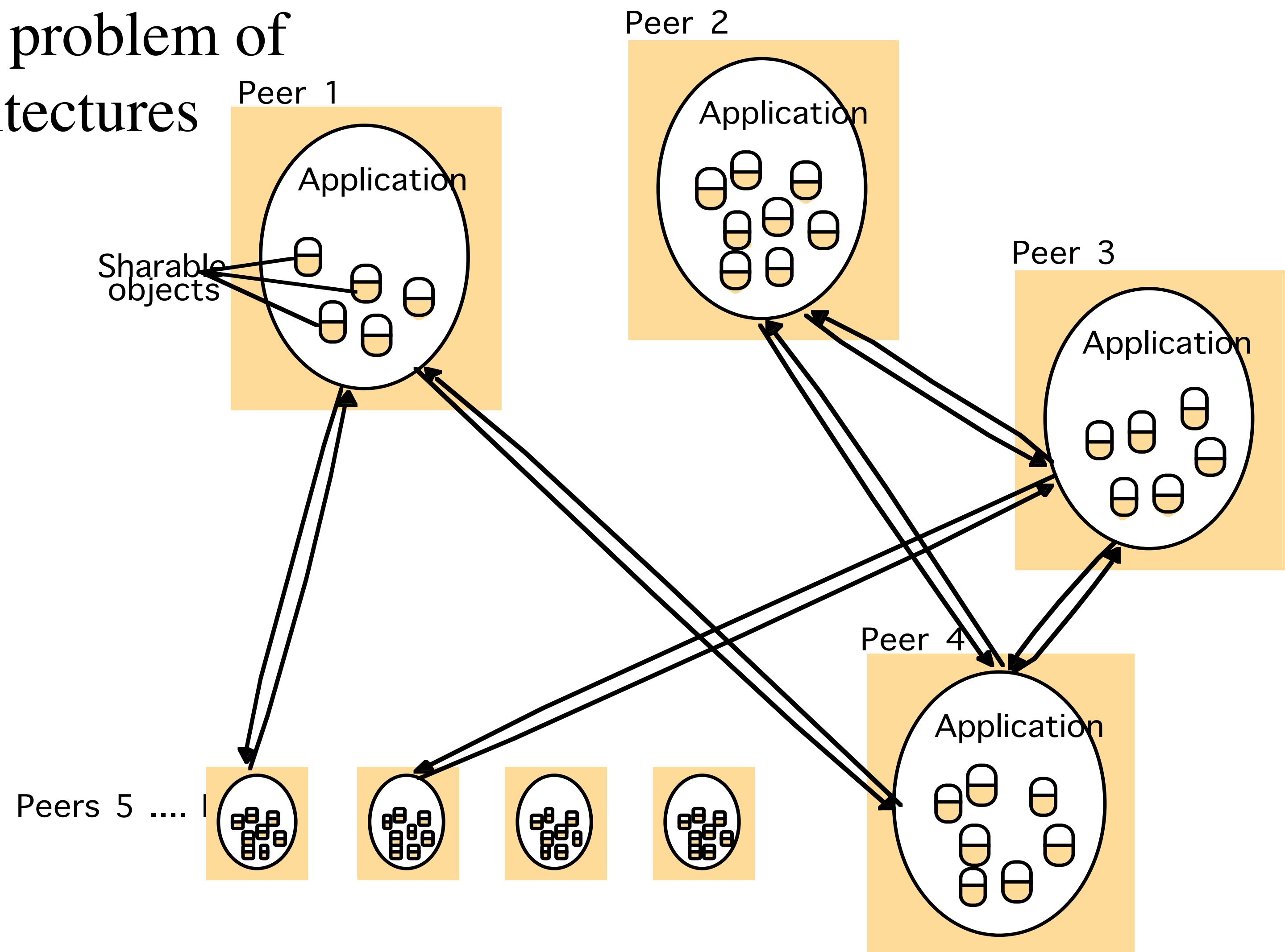
Roles and responsibilities: client/server

- each architectural element has some duties in order to achieve the global goal
- Most common: client/server model



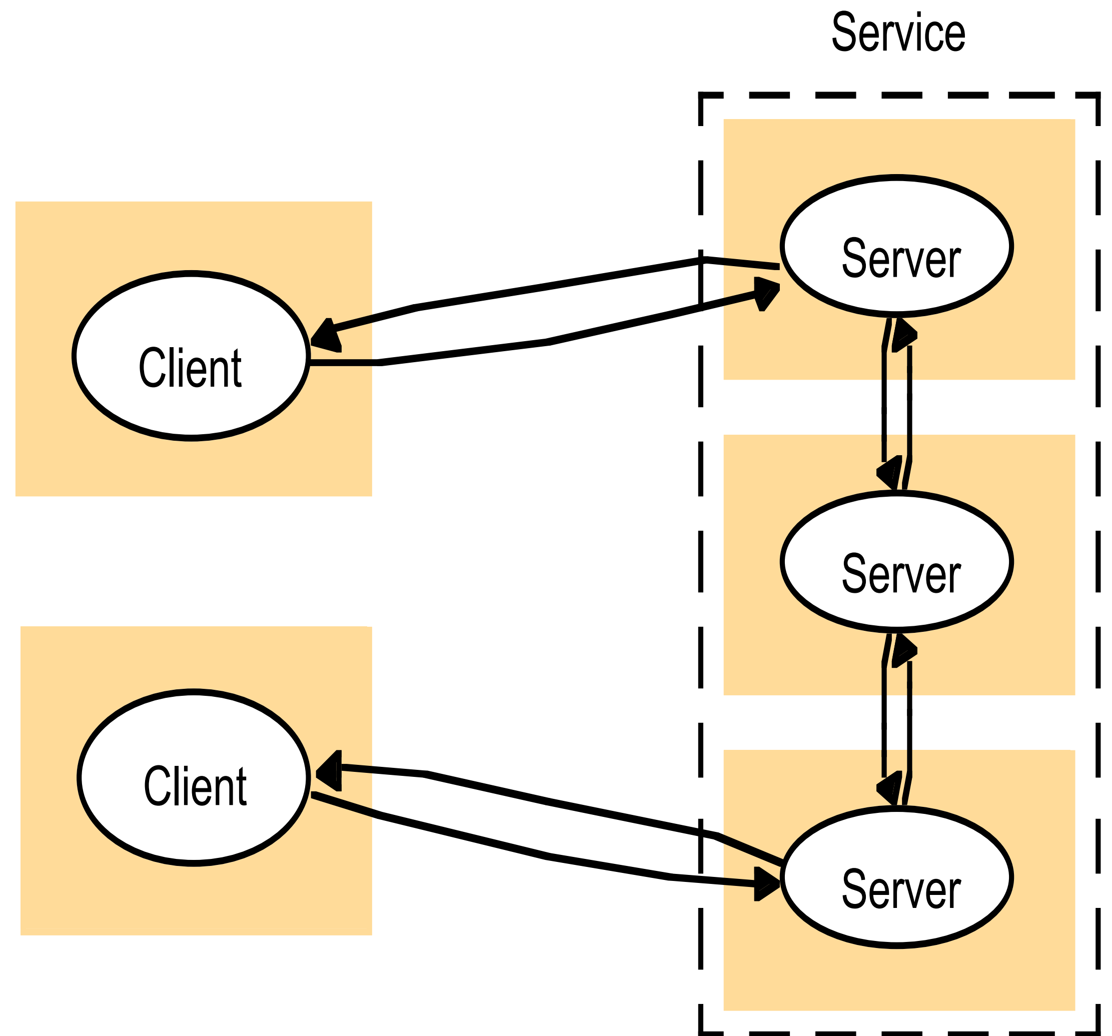
Roles and responsibilities: peer processes

Peers play similar roles
No distinction of responsibilities
Solves scalability problem of client/server architectures



Mixed models: client/server with p2p servers

- Objects are Service partitioned/replicated
  - Web: each server manages its objects
  - NIS: replicated login/ password info
  - Cluster: closely coupled, scalable (search engines)
- Servers are peers
- Example: eMule, Skype, Napster

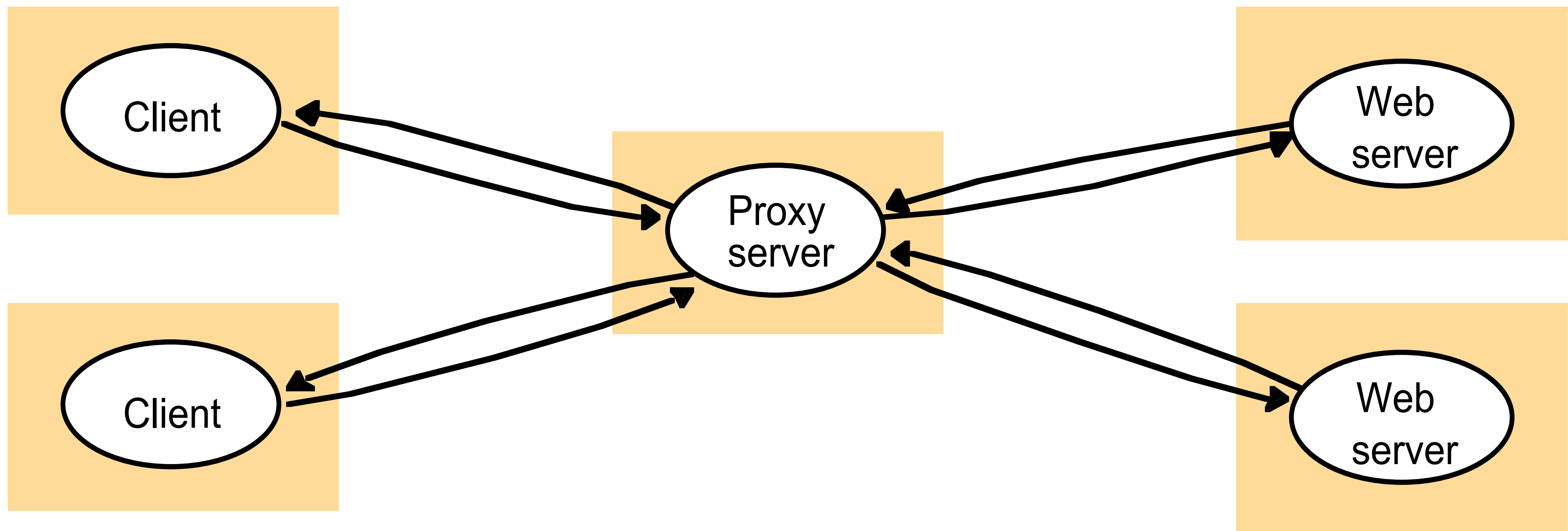


Placement

- Placement considers where entities are physically located
- Crucial in terms of performance but also security
  - e.g. a server: in LAN, in DMZ or "outside"?
- Parameters to take into account:
  - patterns of communications
  - reliability of machines
  - speed and reliability of connections
- Usually very specific for each situation
- Some examples

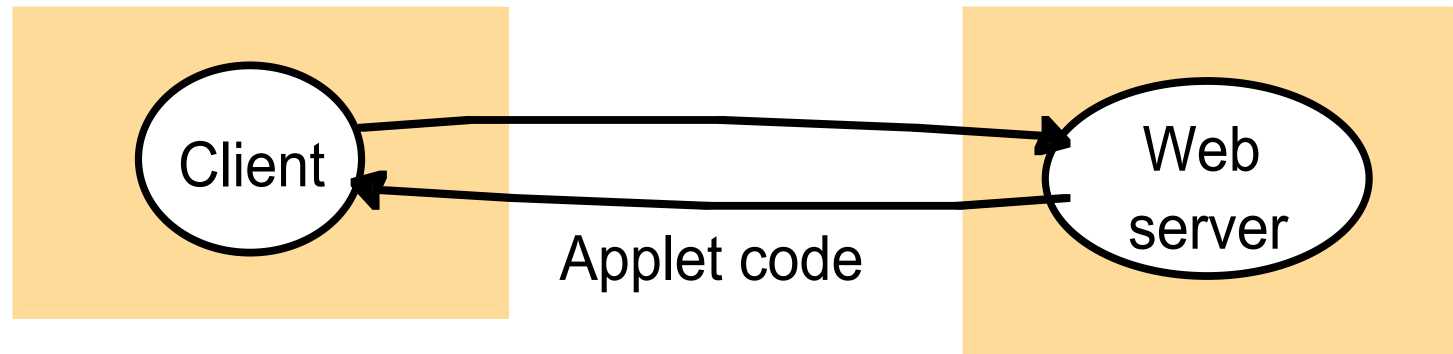
Placement examples: multiple servers, caching

- Multiple servers
  - data not replicated: partitioning of data (e.g. the Web)
  - data replicated: good for load balancing
- Caching



Placement example: mobile code

a) client request results in the downloading of applet code

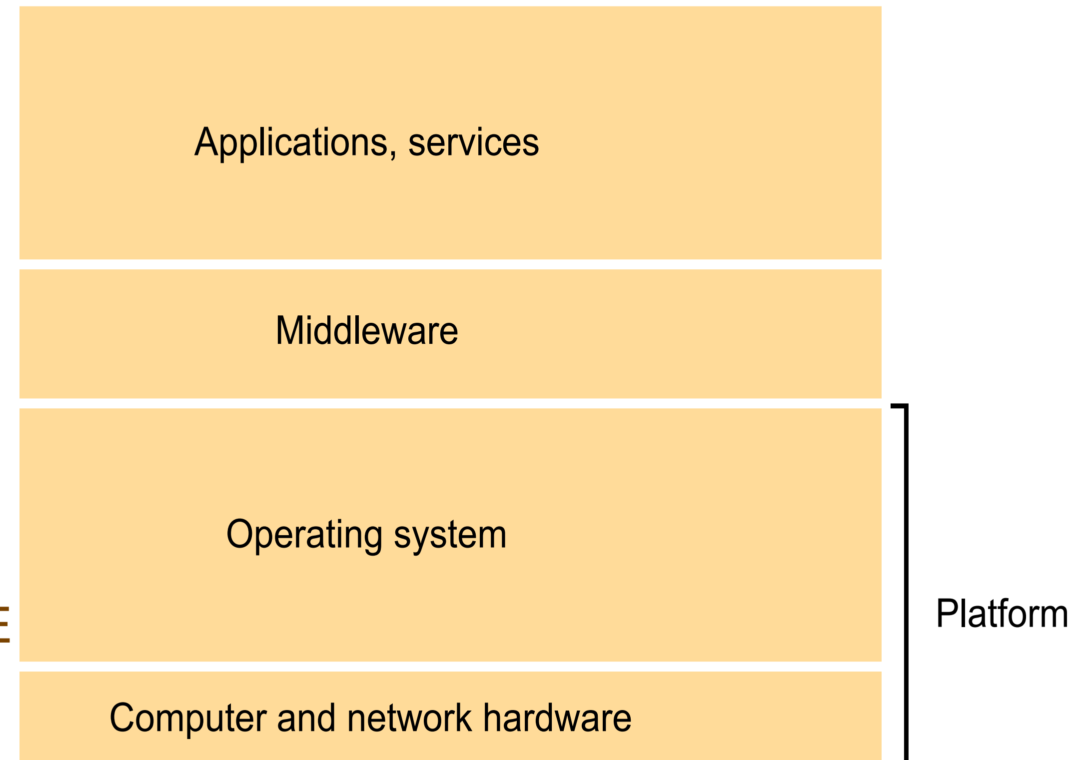


b) client interacts with the applet

Mobile code is a potential secure threat -> browser must adopt specific security schema (sandboxing)

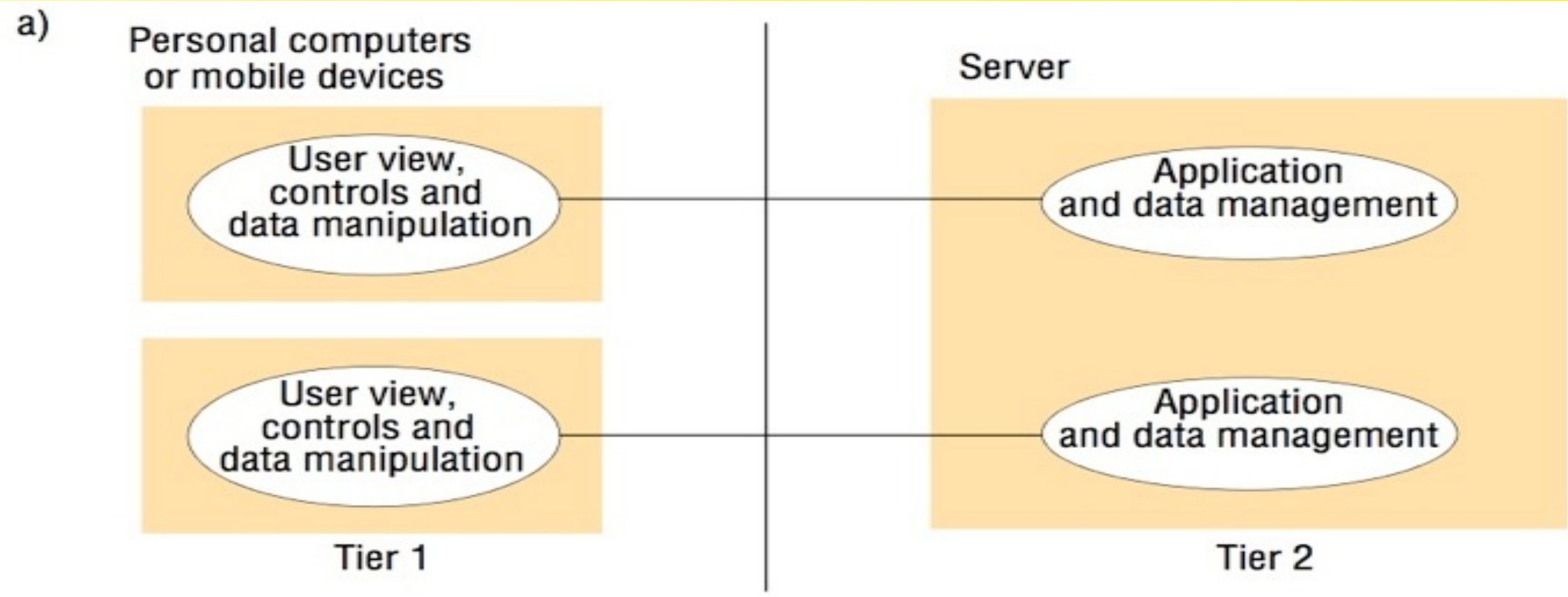
Architectural Patterns: Layering

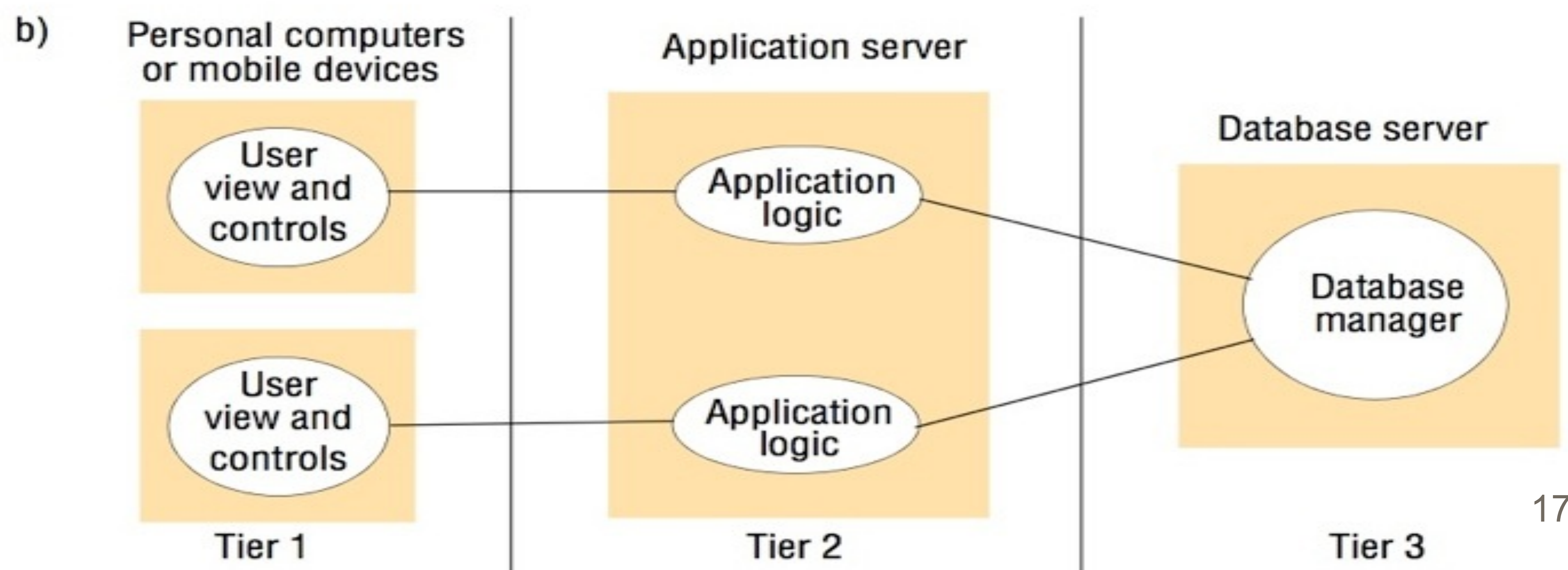
- Platform: hardware, OS
- Middleware
  - masking heterogeneity of underlying platform, hence providing a common "platform"
  - Examples:
    - Sun RPC, Java RMI
    - CORBA, Microsoft .NET, Java J2EE
  - reliable services in the middle layer, still need application- specific reliability
- Applications, services



- Tiering is complementary to layering
  - each layer can be organized in tiers
  - “horizontal” vs “vertical”
- Two- and three-tiers are common
  - presentation/application/data logics
  - Example: AJAX
- Can be generalized to n-tiered architectures

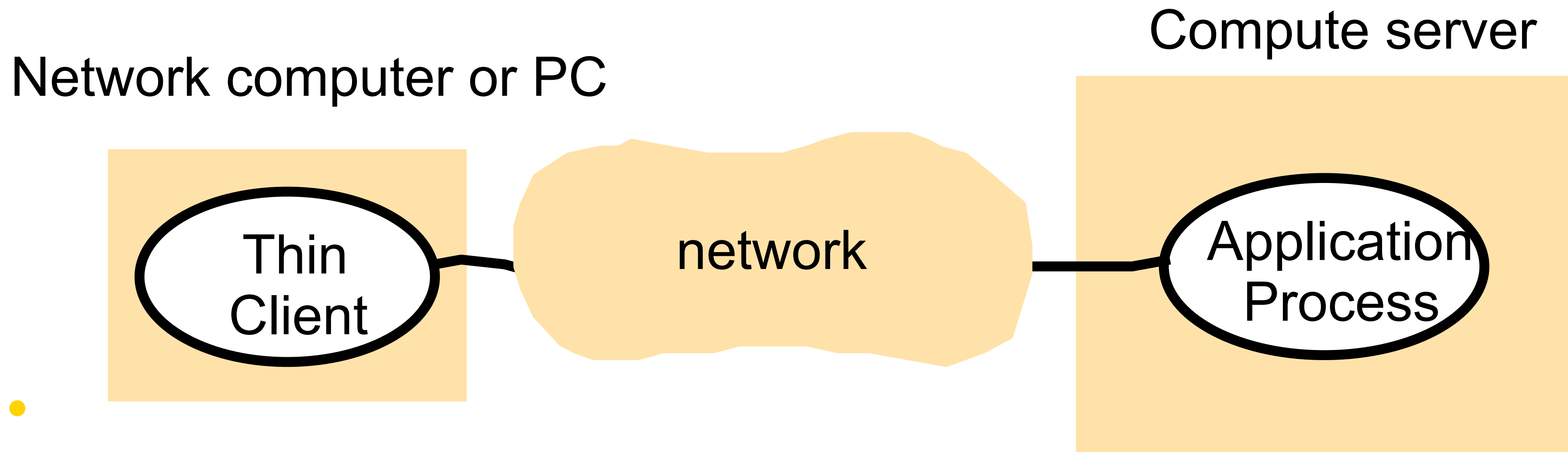
Architectural Patterns: Tiered Architecture





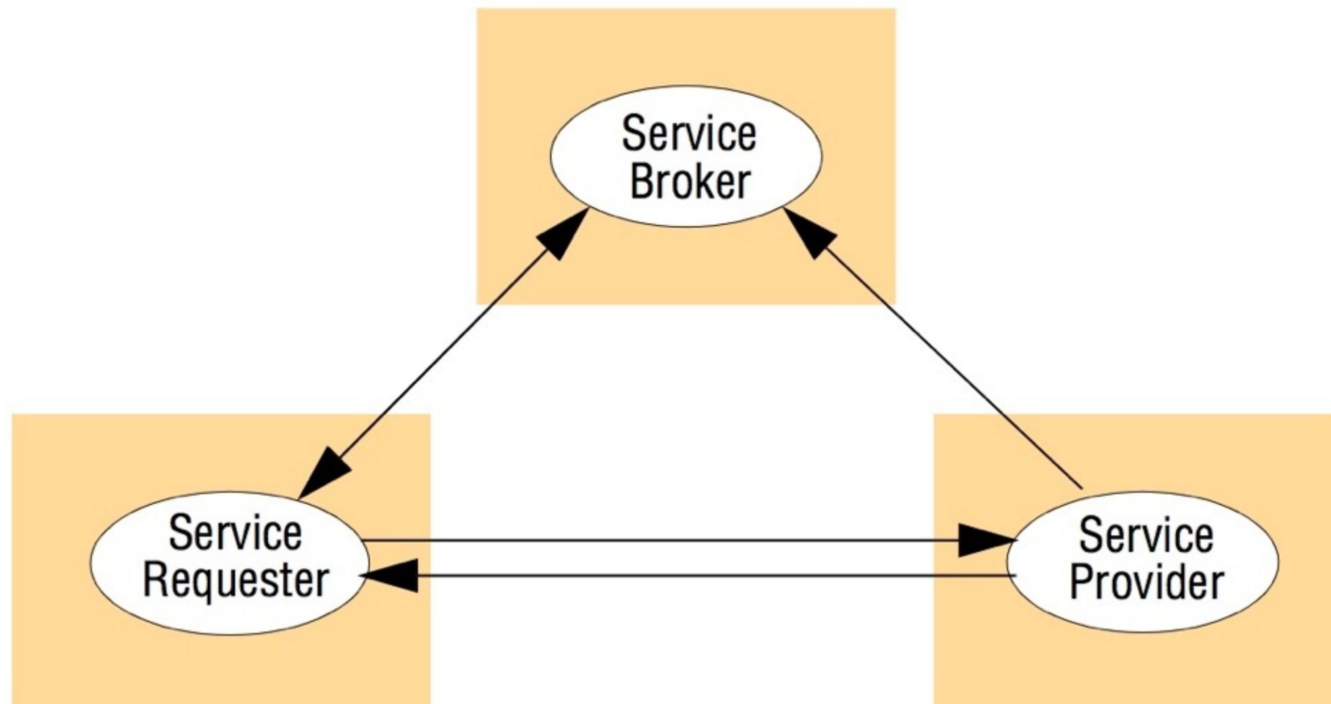
Architectural Patterns: Thin Clients

- Moving complexity from the end-user device towards services in the Internet
- Thin client: accessing to advanced network services (e.g. whole servers) from small, lightweight devices, supporting just a windows-based interface



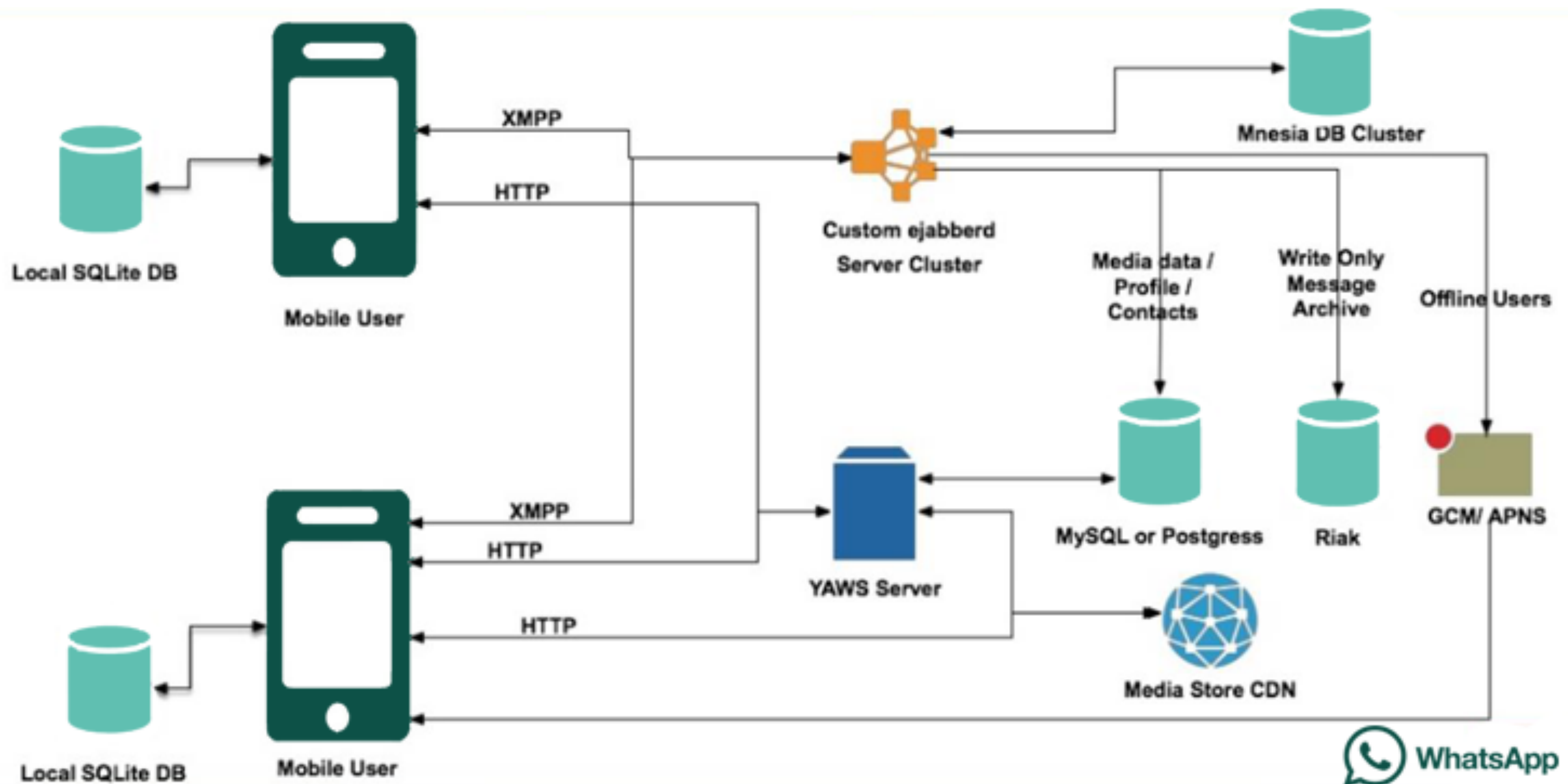
Architectural Patterns: Brokerage

- An intermediate process is a “meta-server”: allows for finding the right server offering the service needed by the client
  - used in WSDL, CORBA, Java RMI



Example: Whatsapp Architecture

- LYME: Linux-Yaws-Mnesia-Erlang



Middleware solutions

- A middleware has to provide
  - a higher-level programming abstraction for the development of distributed systems
  - and (through layering) to abstract over heterogeneity in the underlying infrastructure to promote interoperability and portability
- Many middlewares, depending on which transparency (abstraction) they aim to implement
- Based on the architecture models seen before

Categories of Middlewares

Major categories:	*Subcategory*	*Example systems*
Distributed objects (Chapters 5, 8)	Standard	RM-ODP
	Platform	CORBA
	Platform	Java RMI
Distributed components (Chapter 8)	Lightweight components	Fractal
	Lightweight components	OpenCOM
	Application servers	SUN EJB
	Application servers	CORBA Component Model
	Application servers	JBoss
Publish-subscribe systems (Chapter 6)	-	CORBA Event Service
	-	Scribe
	-	JMS
Message queues (Chapter 6)	-	Websphere MQ
	-	JMS
Web services (Chapter 9)	Web services	Apache Axis
	Grid services	The Globus Toolkit
Peer-to-peer (Chapter 10)	Routing overlays	Pastry
	Routing overlays	Tapestry
	Application-specific	Squirrel
	Application-specific	OceanStore
	Application-specific	Ivy
	Application-specific	Gnutella

Limitation of middlewares

- Many applications rely entirely on the services offered by the middleware (e.g. RPC for client/server connections)
- However middleware cannot (and it is not even desired) achieve ALL the needs of ALL applications
- Something has always to be left "end-to-end"
  - e.g.: failure tolerance email transfer
- "correct behavior in distributed programs depends upon checks, error-correction mechanisms, security measures at many levels, some of which require access to data in application's address space"
- Read [Saltzer, Reed, Clarke 1984]
- Boundaries of middleware layer are not fixed

Fundamental Models

- Fundamental models focus on few or just one characteristic of the distributed system
- Contain only the essential aspects, abstracting from unnecessary details
  - Allows to make explicit which are the relevant assumptions about the system we are modelling
  - Allows to make general reasonings, general results, general algorithms, and possibly mathematical proofs of the desired properties
- Many fundamental models have been proposed

Interaction Models

- Local systems are described as algorithms (steps to be executed)
- Distributed systems can be described as "distributed algorithms":
  - a definition of steps to be taken by each of the processes
  - *including the transmission of messages among them*
  - Called also *protocols*
- Two key factors:
  - Communication performance is a limiting characteristic
    - latency, bandwidth, jitter
  - It is impossible to maintain a single global time
    - clock drift, synchronization impossible/costly

- **Synchronous distributed systems**: with some precise bounds:
  - Process execution speed has known lower and upper bounds
  - Message transmission delay has a known bound
  - Each process has a local clock whose drift rate from real time has a known bound
- Sometimes can be realizable, by requiring precise features and speeds on processors, links, etc.
- SDSs allow for several algorithms not possible in asynchronous system (e.g. failure detection)

- **Asynchronous distributed systems**: there are NO bounds on:
  - Process execution speed: each process runs at its own pace
  - Message transmission delay: each message can take as long as it wants
  - Clock drift rate
- This exactly models the Internet and systems implemented over it

Interaction models: Cause and effect

- No simultaneity in an asynchronous system
  - => we need something else in order to study the events in such a system.
- We do have the notion of **cause** and **effect**:
- If one event e_i caused another event e_j to happen, then e_i and e_j could never have happened simultaneously:

$$e_i \text{ happened before } e_j$$

- This defines a **causal order** among events
  - within a process: by local time
  - among processes: by send/receive

Causal orders

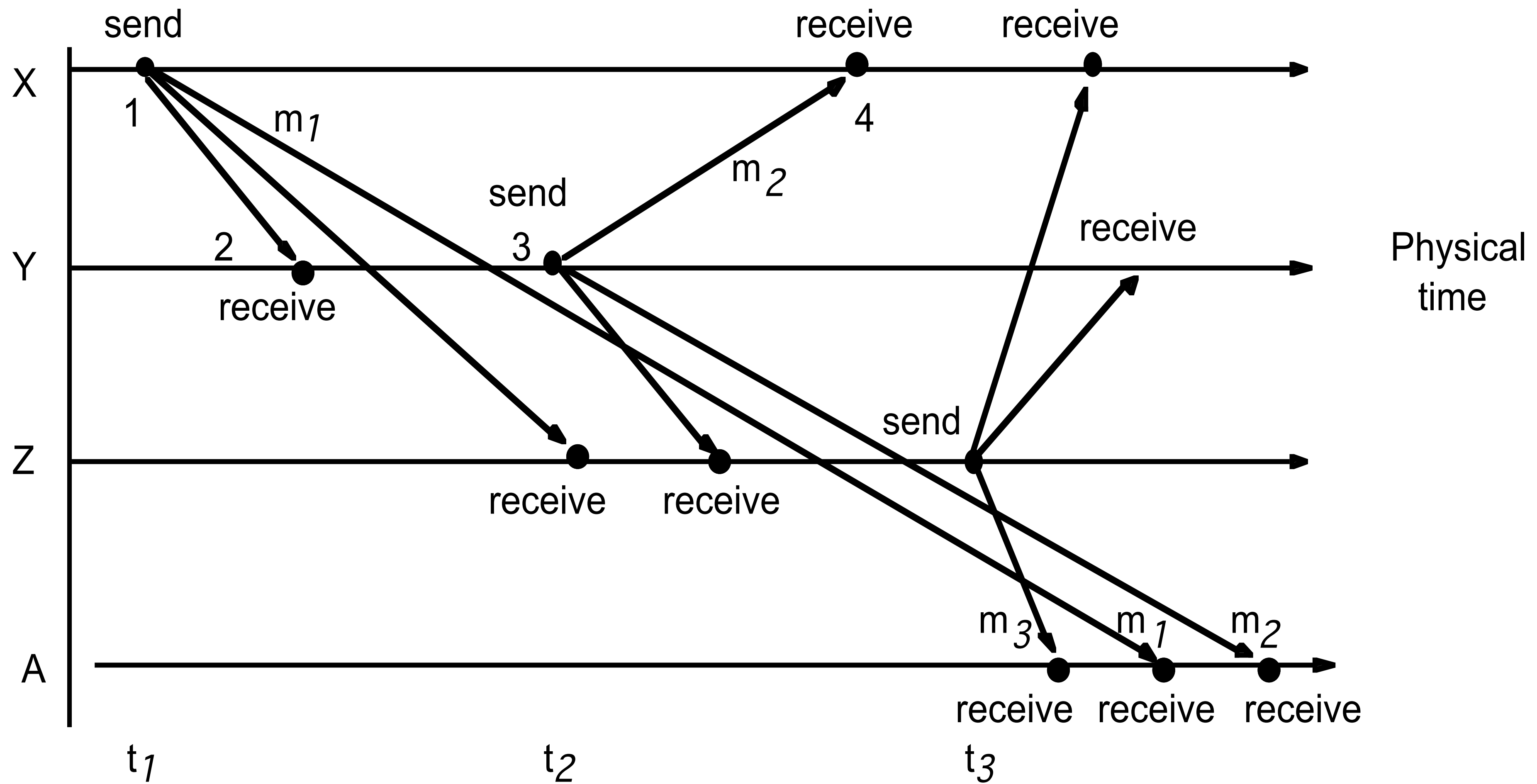
For each process i, let us denote h_i the set of its events e_i^k, ordered over k. We define a binary relation $\rightarrow$ over events, such that

- If $e_i^k, e_i^l \in h_i$ and $k < l$ then $e_i^k \rightarrow e_i^l$
- If $e_i = send(m)$ and $e_j = receive(m)$, then $e_i \rightarrow e_j$
- If $e \rightarrow e'$ and $e' \rightarrow e''$ then $e \rightarrow e''$

When $e \rightarrow e'$ we say that e *causally precedes* e', or *happens before* e'
We define *concurrent* as $e \parallel e' \equiv \neg(e \rightarrow e' \vee e' \rightarrow e)$

- These models have been studied in depth as “event structures” (see e.g. G. Winskel, “An introduction to event structures”, LNCS 354, 1989. Doi: 10.1007/BFb0013026)

Figure 2.8
Real-time ordering of events



Example: email reordering

- Suppose we (A) receive the following emails:
  - From Z: Re: Meeting
  - From X: Meeting
  - From Y: Re: Meeting
- Usually reordered by sender's timestamp - which is imprecise
- But we know that
  - Y receives a message from X after X sends it
  - Z receives a message from X after X sends it
  - Y sends a message after receives a message
  - Z sends a message after receives a message
- So a possible correct order is
  - From X: Meeting
  - From Z: Re: Meeting
  - From Y: Re: Meeting
- We cannot be sure about Z and Y messages order

- In distributed systems, both processes and communication channels may fail
  - Omission failures
    - message lost, process crash, ...
    - detected by means of timeouts
    - difficult to recover if process does not crash cleanly
  - Arbitrary failures (aka Byzantine)
    - anything in the system can behave in whatever mean
    - Arbitrary failure of channels are rare (see TCP/IP)
    - impossible to reach agreement in presence of communication failures.
  - Timing failures
    - applicable in distributed systems where time limits are set (e.g. multimedia, soft real-time applications)

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.

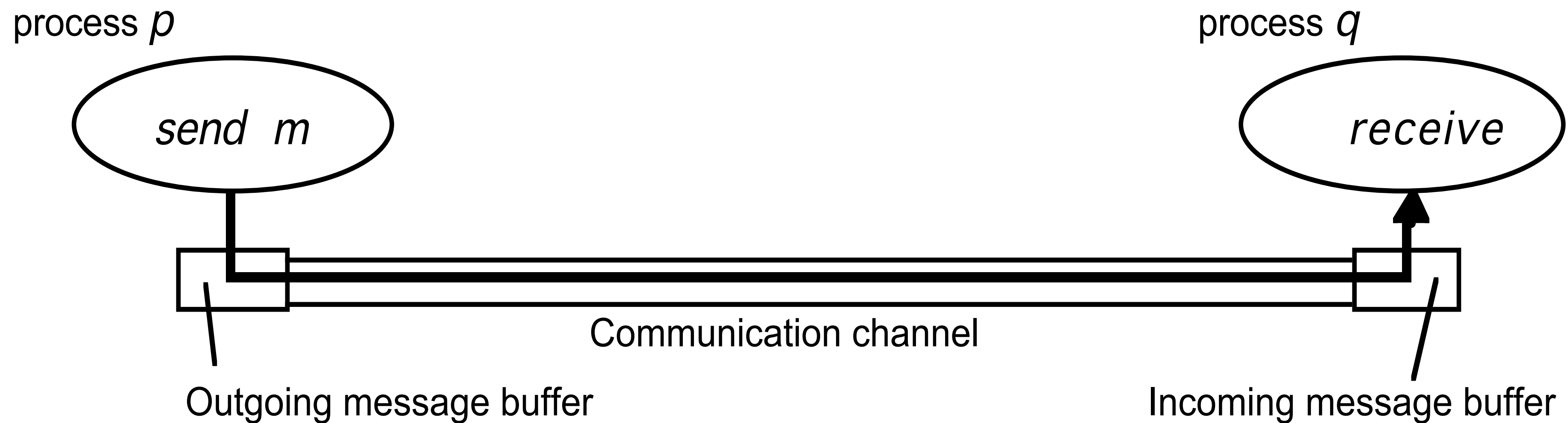
- Suppose that A and B can exchange messengers, which can captured by X
- Is there any protocol allowing A and B to reach an agreement on when attacking X?
- Answer is NO.
- Proof:
  - let us assume there is a protocol of length n, that is, it exchanges n messages.
  - Then X can capture the last messenger. So the last messenger is not useful and can be safely omitted.
  - But then by repeating the argument, all steps are cancelled => the protocol is empty. Absurd.

- Deterministically, the problem is solvable for, and only for, $n \geq 3m + 1$, where m is the number of faulty processors and n is the total number.
  - Pease, Shostak, Lamport, “Reaching Agreement in the Presence of Faults”, J.ACM 1980.
- If messages got lost with probability *p*, and processes fail with probability *q*, we can aim for a *statistic* agreement: given any $r < 1$, there is a protocol which allows to reach agreement (of the non-faulty processes) with probability *r*.

Figure 2.11
Timing failures

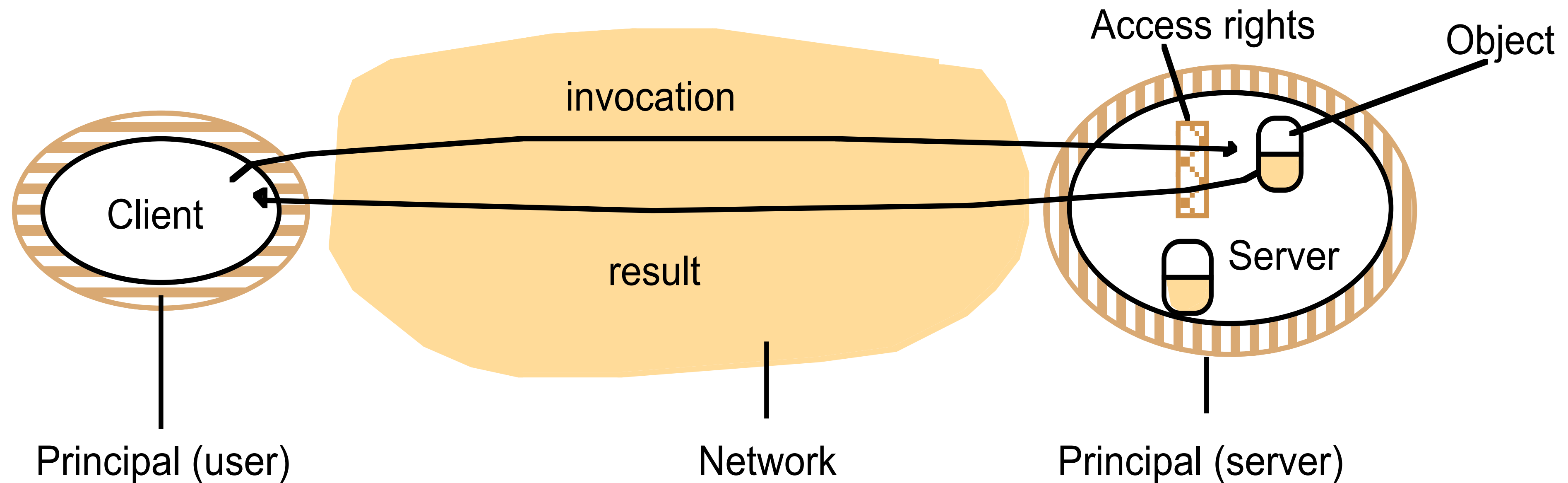
Class of Failure	*Affects*	*Description*
Clock	Process	Process's local clock exceeds the bounds on its rate of drift from real time.
Performance	Process	Process exceeds the bounds on the interval between two steps.
Performance	Channel	A message's transmission takes longer than the stated bound.

Security models



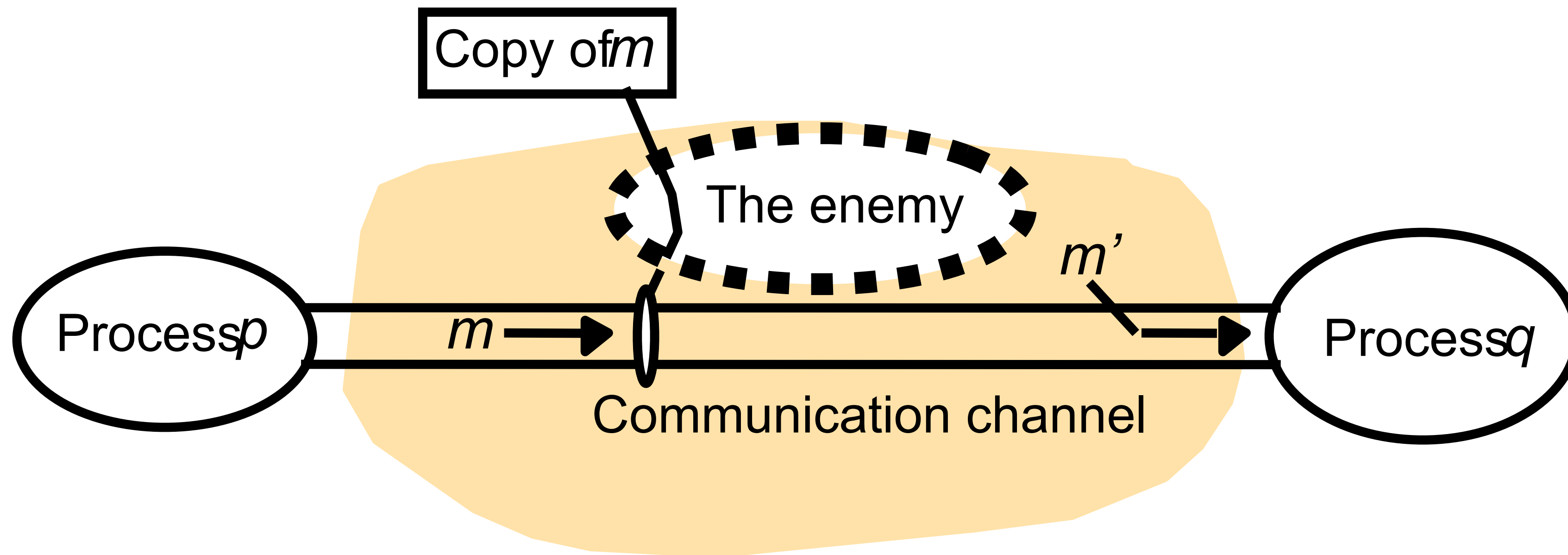
- security of a distributed system can be obtained by securing the processes and the channel used for their interactions and by protecting the objects that they encapsulate, against unauthorized accesses

Security model: Objects and principals



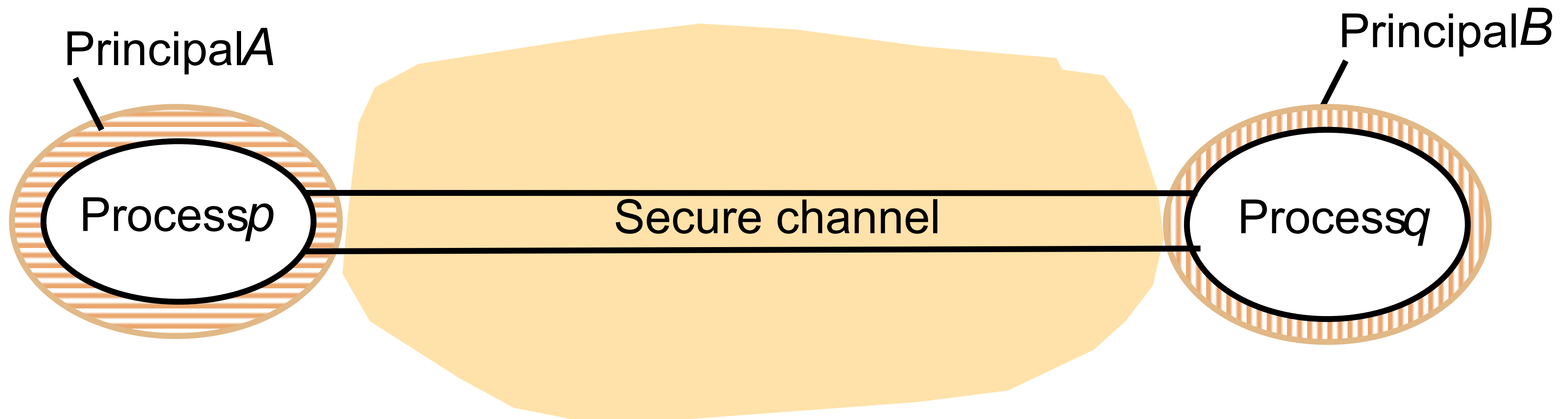
- CIA: Confidentiality, Integrity, Availability
- server is responsible for verifying client's identity and checking its access rights
- client may check the identity of the server

The enemy



- Dolev-Yao model: the enemy can:
  - read messages during transmission (sniffing)
  - retain messages, and possibly reply them
  - observe traffic
  - forge new messages
- slogan: “the channel is the enemy”

Secure channels



- threats can be avoided by implementing "secure channels"
- Based on cryptography and authentication protocols

Movin' on!