Chapter 1

Distributed Software: Introduction

1. Definitions and Terminology
2. Developing distributed applications
3. Architecture
4. Design: middleware and services
5. Classes of distributed systems
6. Important architectures and platforms
7. Scalability and high availability
8. The Java family
9. The .NET family
10. This course

Chapter 1

Distributed Software: Introduction

1. **Definitions and Terminology**
    1. Introduction
    2. Examples
    3. Definitions
    4. Need for distributed systems
    5. Why are distributed systems different ?
2. **Developing distributed applications**
3. **Architecture**
4. **Design: middleware and services**
5. **Classes of distributed systems**

…

Why this course ?

A loose definition

distributed system =

- **collection of interacting processes**
- **appearing as single application to the end user**

"The network is the computer"

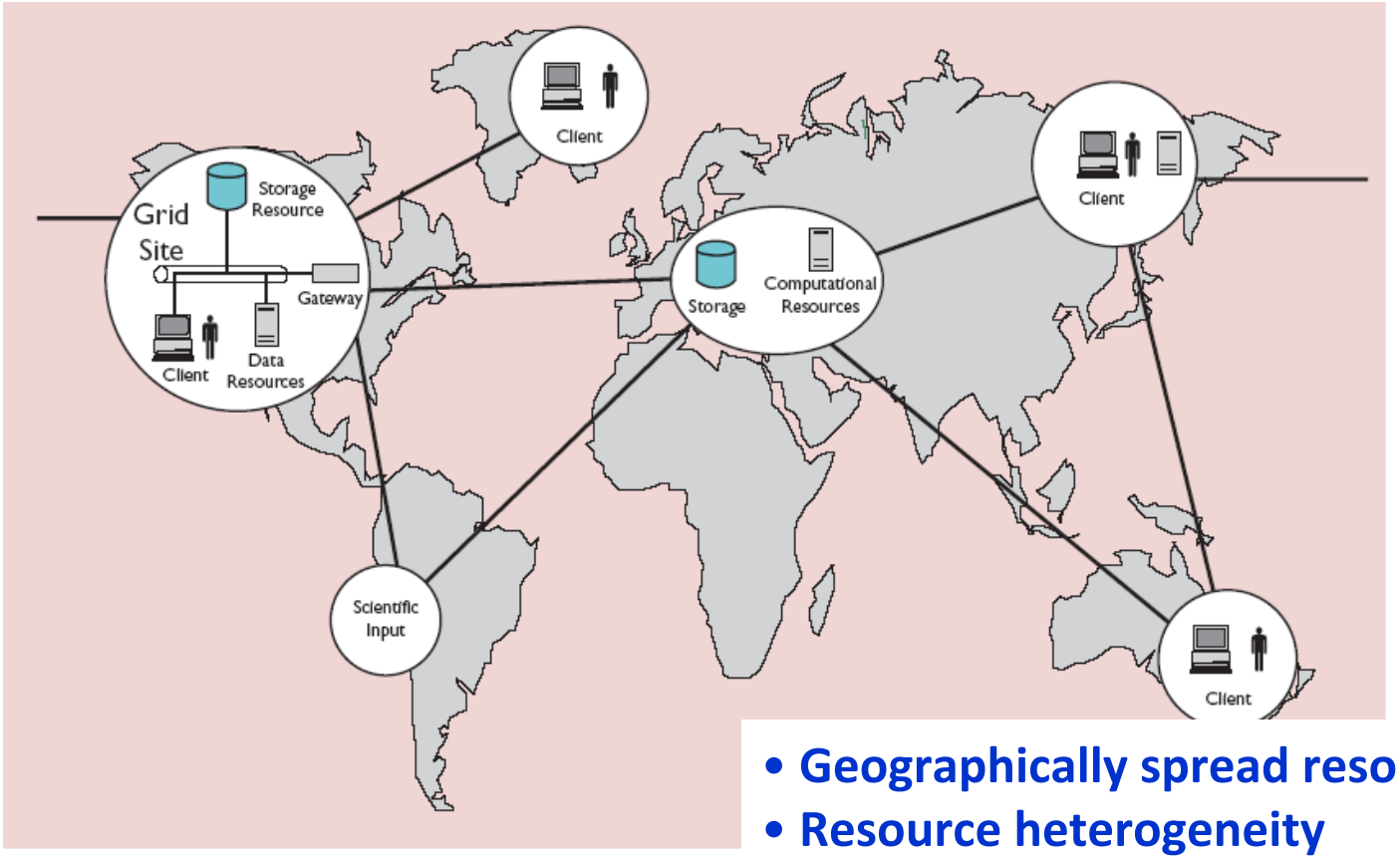
- search portals for huge information stores
- grid computing
- cloud management systems
- P2P applications
- communication tools
- online gaming
- Internet of Things applications

Technology = standard programming tools + sockets ?

- specific, recurring problems due to distributed nature
- solve once, use many times
- middleware solves these problems in a generic way

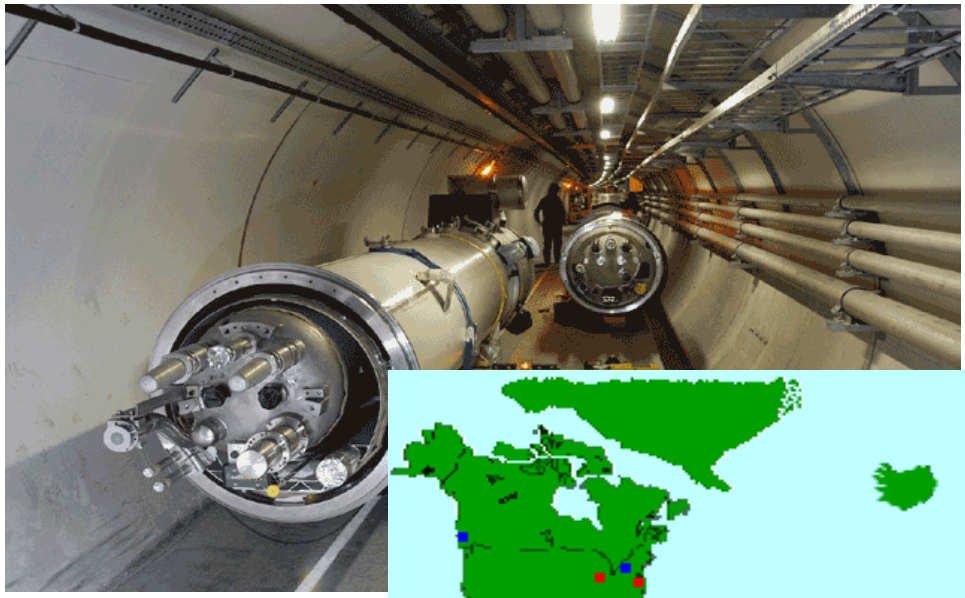
"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities."

[Kesselman & Foster, 1998]



- **Geographically spread resources**
- **Resource heterogeneity**
- **No single entity of control (no big boss)**

Large Hadron Collider (LHC)
Computing Grid Project LCG

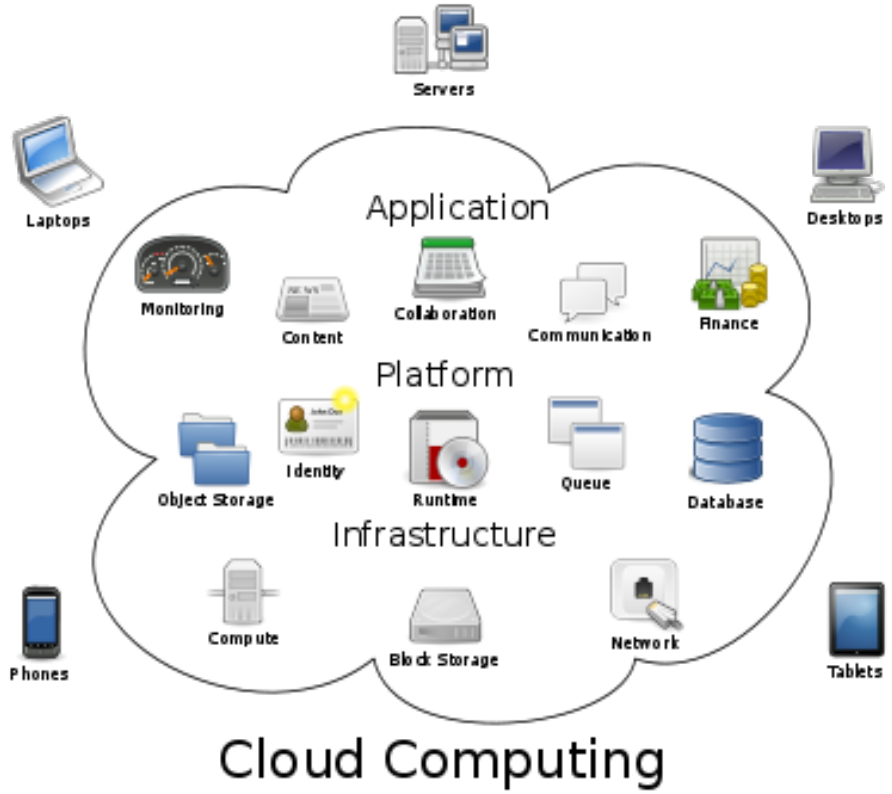


Requirements

- > 6000 users
- 20 Pbyte/year [20 10^6 CDs]
- > 70 000 GFLOP

Roughly defined

Cloud computing = Grid computing
+ virtualization
+ elasticity
+ business model



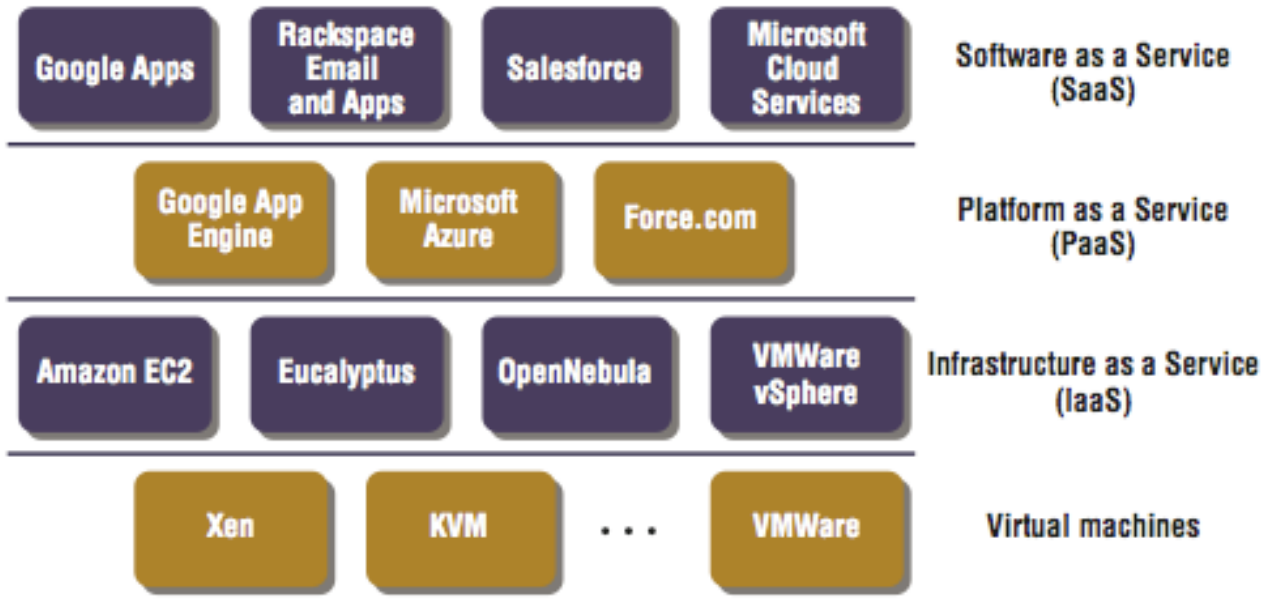
The NIST Definition

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics**, three service models, and four deployment models."**

Cloud computing

Essential Characteristics

-on demand self-service
-broadband access
-resource pooling
-rapid elasticity
-measured service



Service Models

-Software-as-a-Service (SaaS)
-Platform-as-a-Service (PaaS)
-Infrastructure-as-a-Service (IaaS)

Deployment models

-Private Cloud
-Community Cloud
-Public Cloud
-Hybrid Cloud

File sharing

Some facts

Origin : sharing (often illegal) content (KazaA, BitTorrent, eMule, ...)
Huge user base
P2P internet traffic estimated at 70% of total traffic

New applications

- Voice over IP (Skype)
- sharing computational power "desktop grids" (Folding@home)

Benefits

- resources grow with number of users
- better scaling behaviour
- better robustness (no central servers)

Nomadic scenarios

access guaranteed on other locations
(e.g. traveller accessing remote content, using local peripherals)

Mobile scenarios

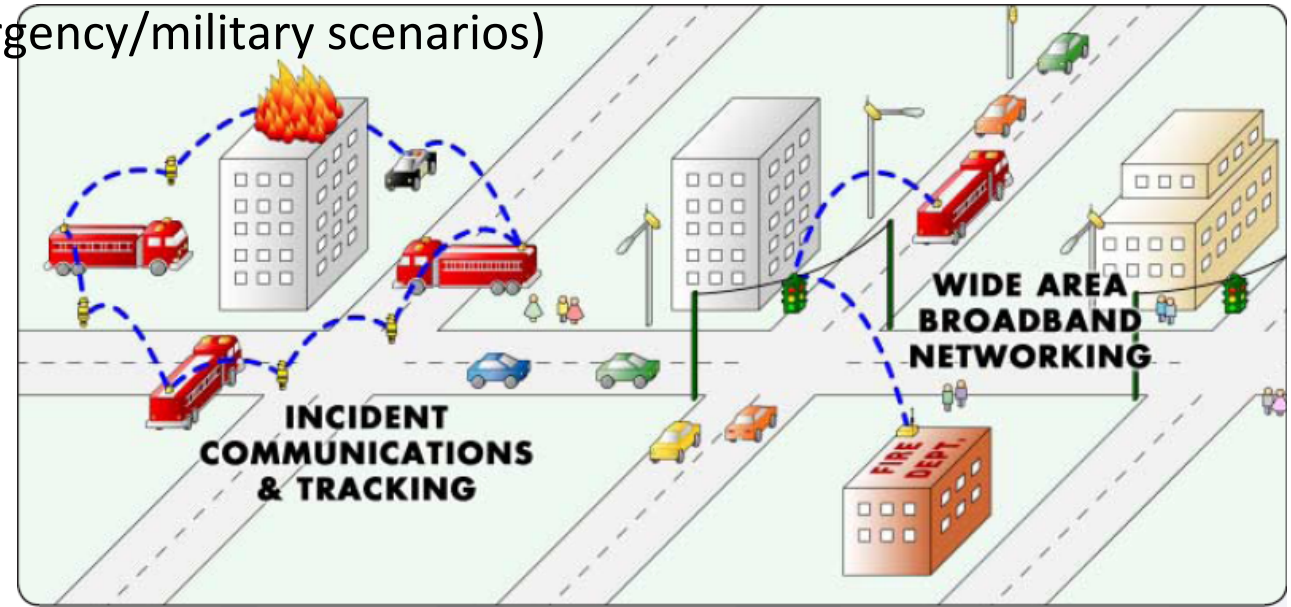
mobility (=moving user) support for applications
(e.g. taking location into account)

Ubiquitous computing scenarios

application fed with sensor info, steers actuators
(e.g. cleaning robot)

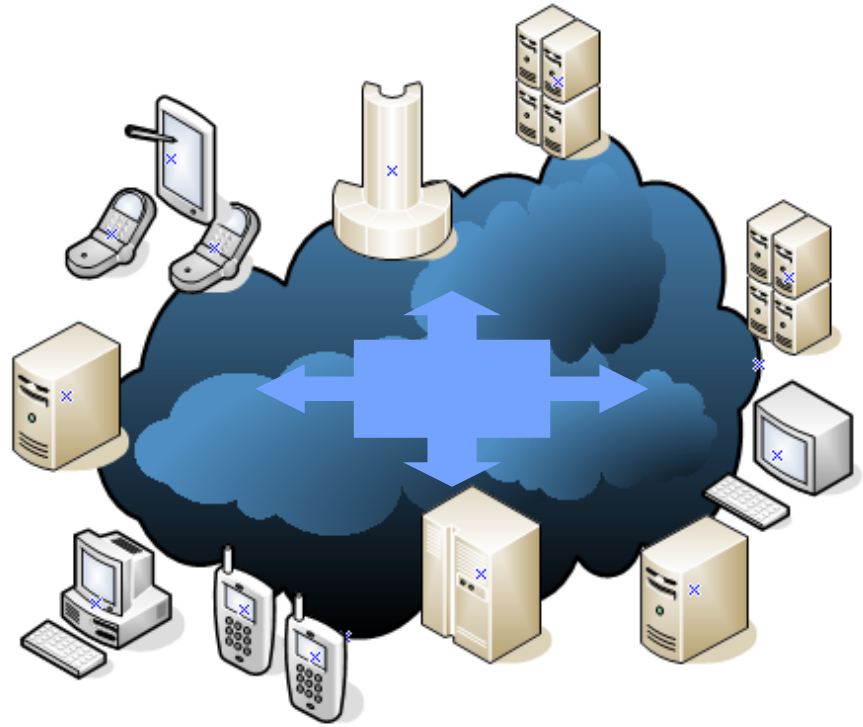
Ad-hoc scenarios

rapid deployment of infrastructure
(often emergency/military scenarios)



Distributed system =

A distributed system = a system where
- hardware and software components are located at networked computers
- components communicate and coordinate their actions ONLY by passing messages



Consequences
- no limit on spatial extent
- no global time notion
- almost always concurrent execution
- (partial) failures likely to happen

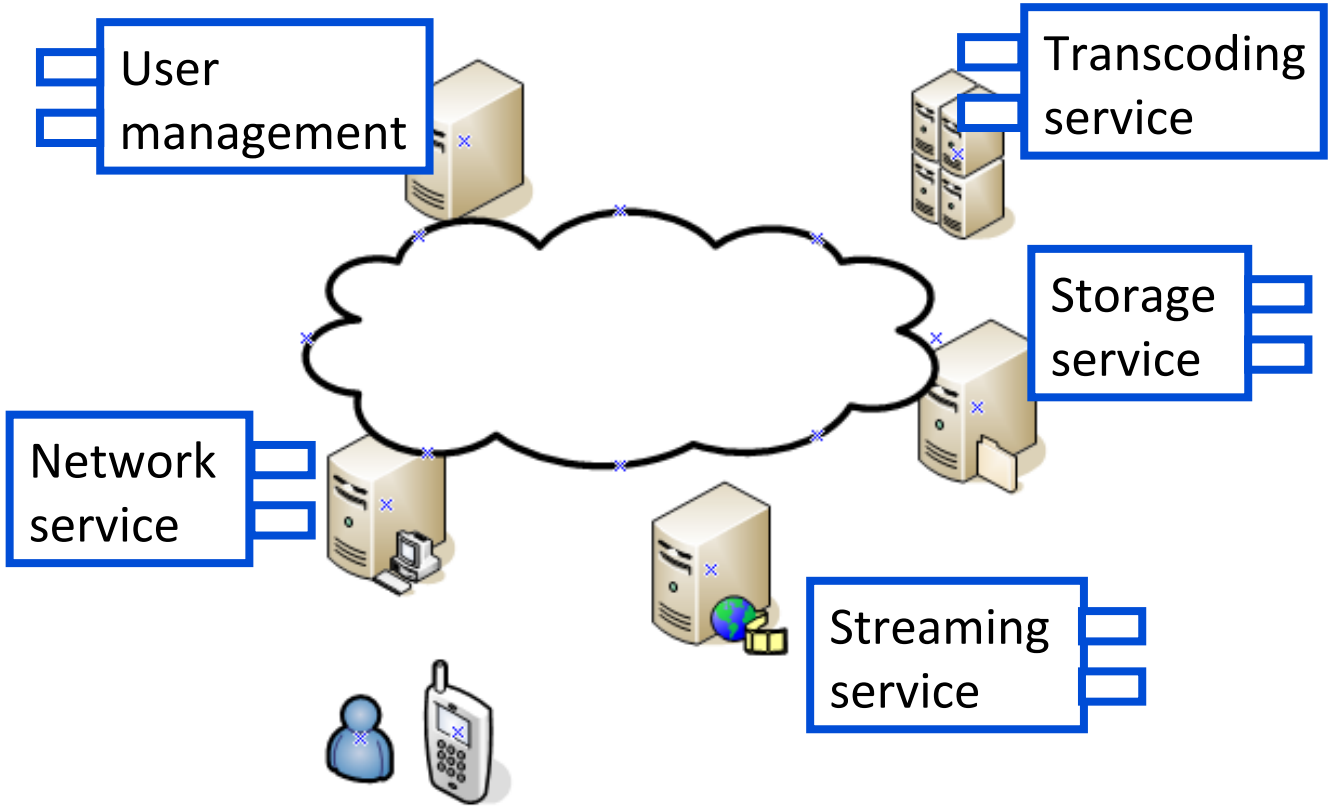
Service =

A service = part of a computer system managing a collection of related resources, presenting their functionality to users and applications

Examples

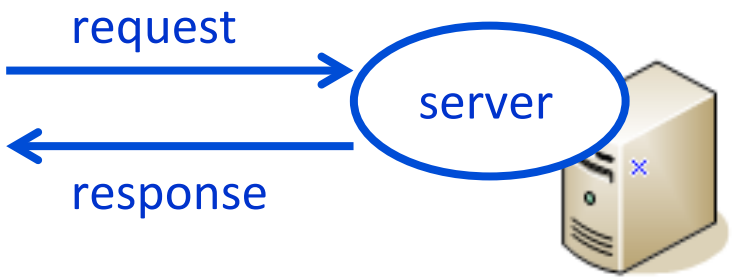
- storage service
- print service
- streaming service
- connectivity service
- transcoding service

Streaming content to mobile clients

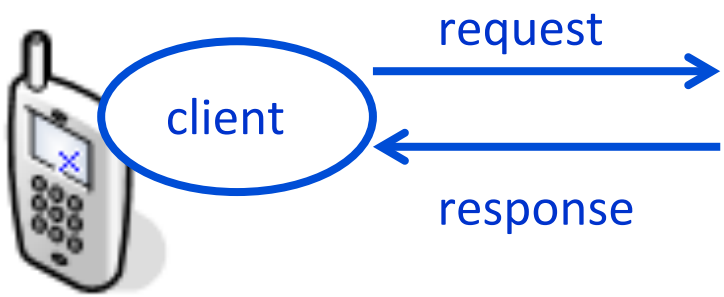


Client - Server

A server = running process on networked computer
- accepting requests to perform a service
- responding appropriately

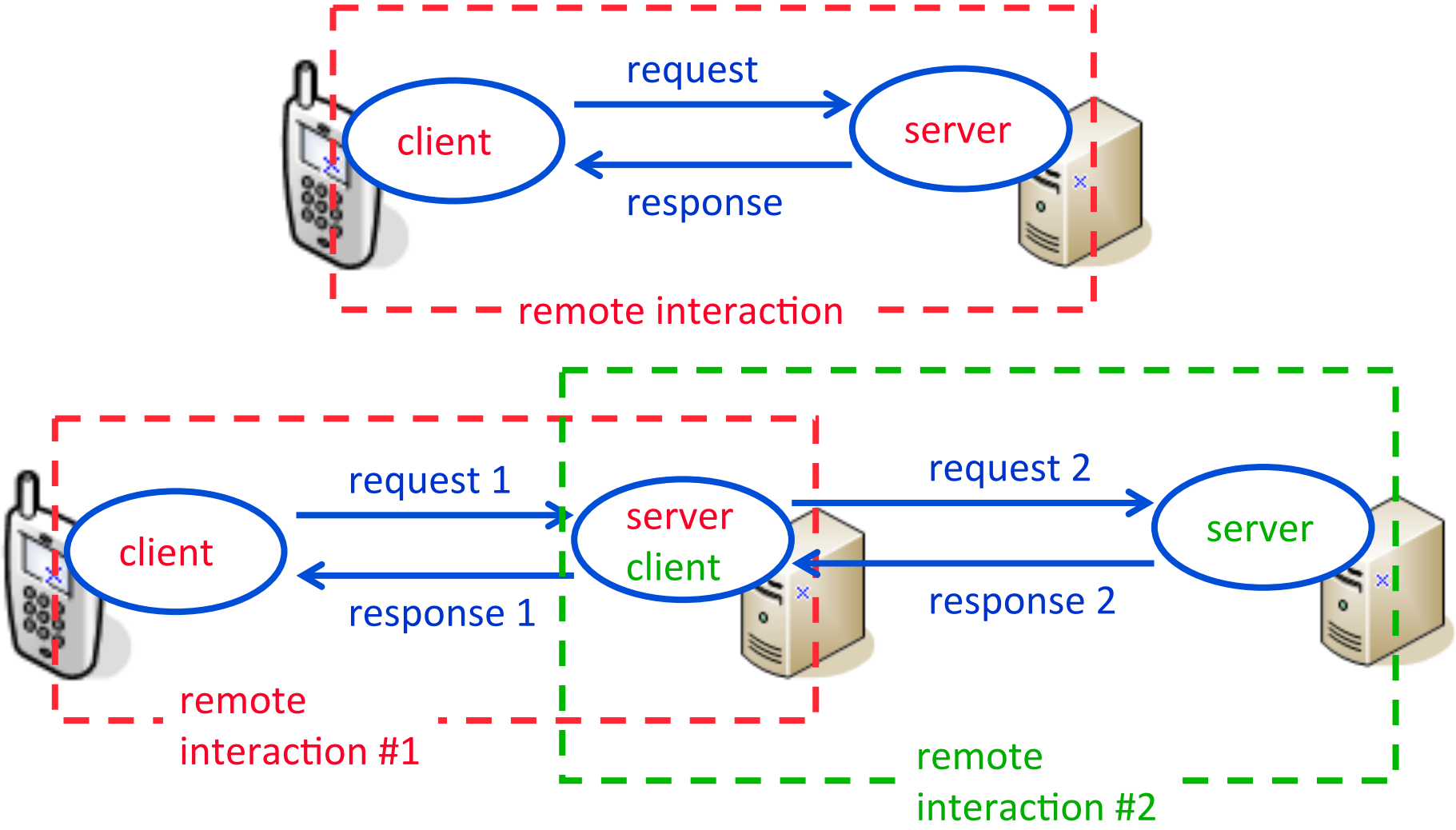


A client = running process on networked computer sending service requests to servers



Remote invocation =

A remote invocation = complete interaction between client and server to process a single request



Why distributed systems ?

Many problems

- no limit on spatial extent, difficult to manage
- no global time notion
- almost always concurrent execution
- (partial) failures likely to happen

Many advantages

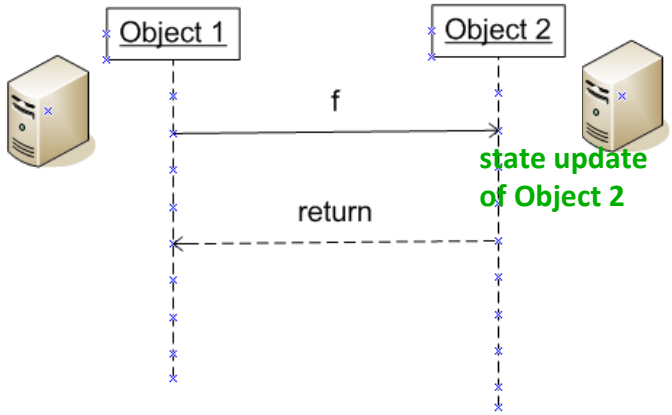
- resource sharing
  - information resources
  - hardware resources
- scalability
  - "easy" to adapt to larger user base
- fault tolerance
  - "easy" to cope with failures

Distributed system vs. standalone system

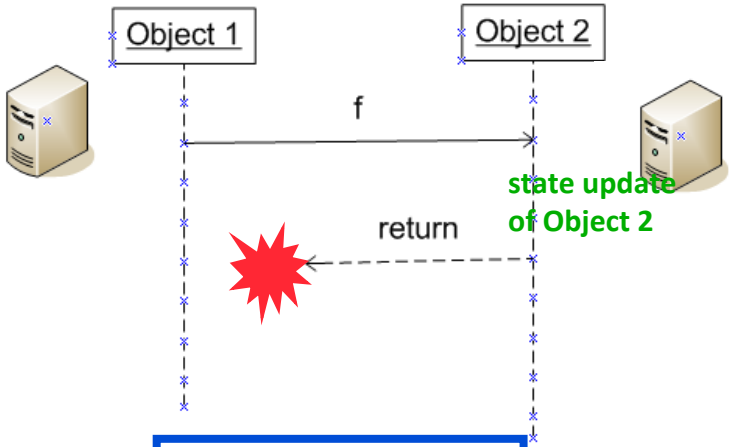
Standalone system	Distributed system
Communication between processes possible through shared memory.	Communication between processes only through the network.
Global state possible through shared memory.	No global state.
Local operating system can be used to get a shared time value.	No global notion of time.
Local operating system offers primitives to safely share resources.	Synchonisation between processes through network communication only.
Failing processes easy to detect (communication is reliable).	(Partial) failures difficult to detect (because of unreliable communication).
Infrastructure known beforehand.	Infrastructure can vary dynamically (new servers/nodes can become available).
Components are located on the standalone machine.	Location of components can vary dynamically.
Local operating system enforces security.	Security threat due to distributed nature (possibly vulnerable communication link).

Faulty communication or faulty process ?

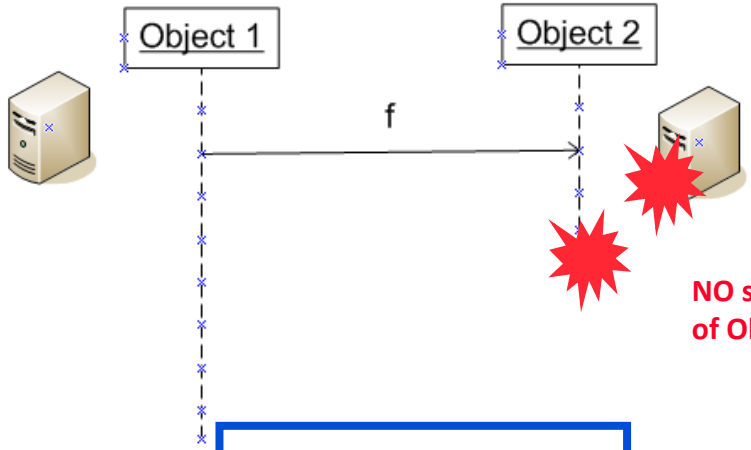
Normal scenario



Failure scenario's



communcation failure [1]



process failure [2]

How can Object 1 distinguish between [1] and [2] ?

Chapter 1

Introduction

1. Definitions and Terminology
2. **Developing distributed applications**
    1. The development process
    2. System models
    3. Challenges for developing distributed app's
3. Architecture
4. Design: middleware and services
5. Classes of distributed systems
6. Important architectures and platforms

...

Developing standalone applications

Requirements analysis phase

- required functions
- non-functional constraints

Architectural phase

Formalised requirements
UML-diagrams capturing requirements

WHAT ? [OOA]

Design phase

High level architecture
subsystems and interfaces
Detailed design
(in parallel for each subsystem)

HOW ?

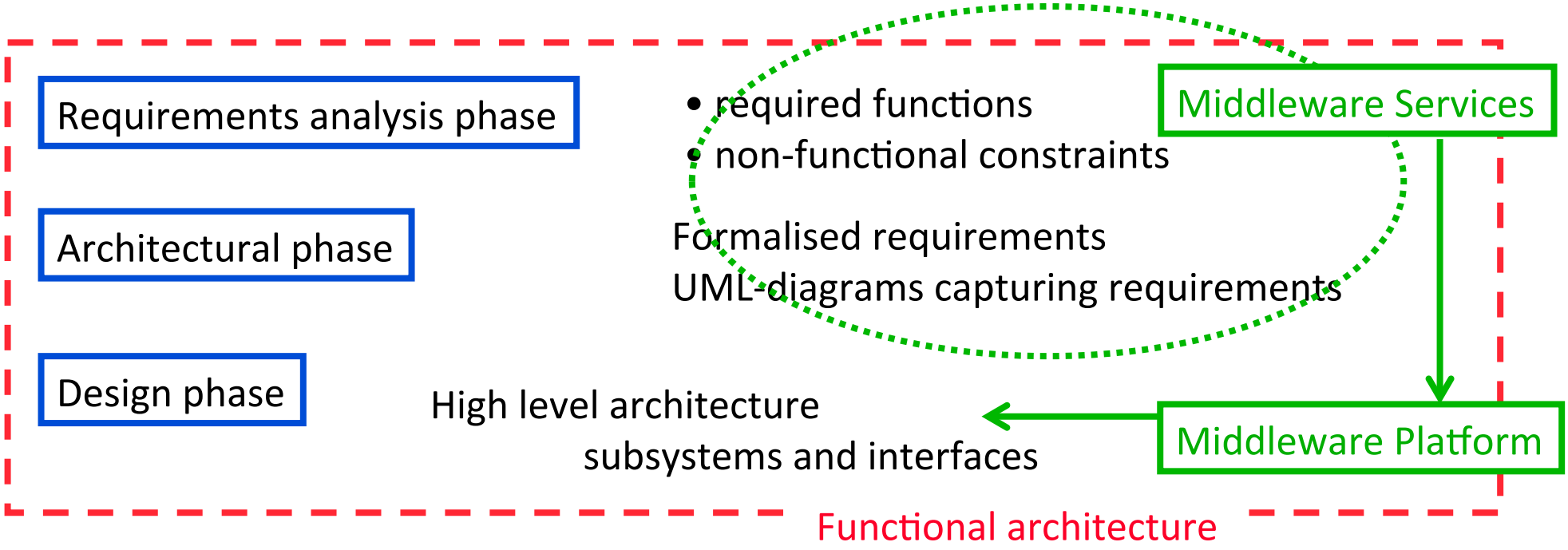
Implementation phase

Code + documentation (in parallel)

Integration phase

Combine realised subsystem code

Developing distributed applications



+

Logical architecture

how will the subsystems interact
(e.g. eventing, through remote method call, data base, ...)

System architecture

client-server or P2P ?

System models

non-functional constraints relating to distributed nature

System models

System models

-> capture non-functionals of the requirements related to distributed nature

-> Interaction model

can we give time guarantees on network communication/process execution ?

-> Failure model

which failures should the application cope with ?
how will we detect failures ?
what will we do in case of failures ?

-> Security model

who can do what in the system ?

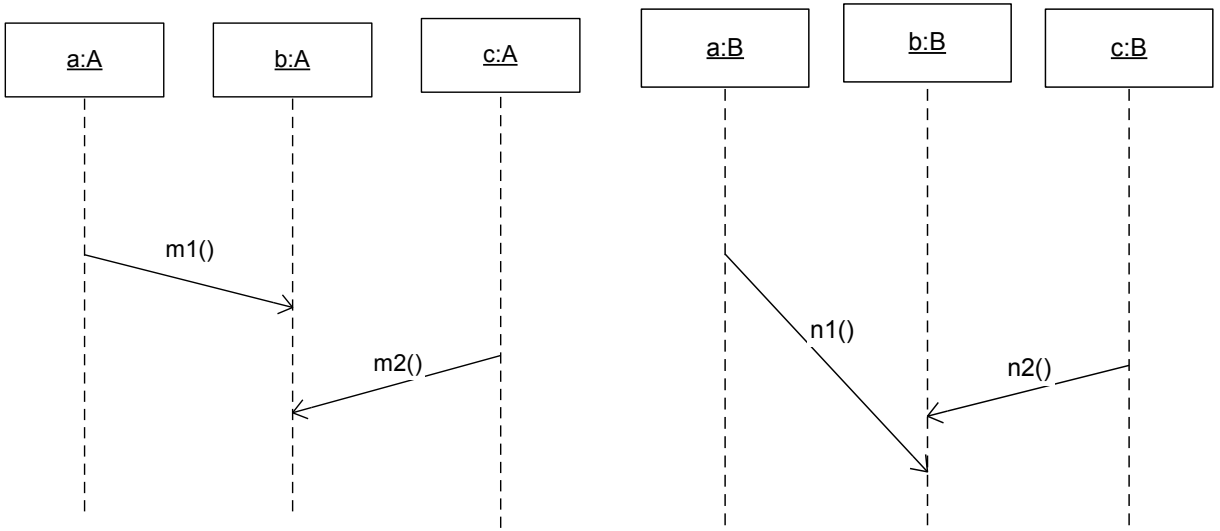
Interaction model

A **synchronous system** is a system where

1. The time to execute each process step has a known lower and upper bound
2. Each message is received within a known bounded time
3. Each process controlled by local clock with known bound for drift rate w.r.t. real time

An **asynchronous system** is a system where one of the above mentioned conditions does not hold.

Asynchronous system needs special algorithms to detect event order



Failure model

Omission failures

a component fails to perform an action

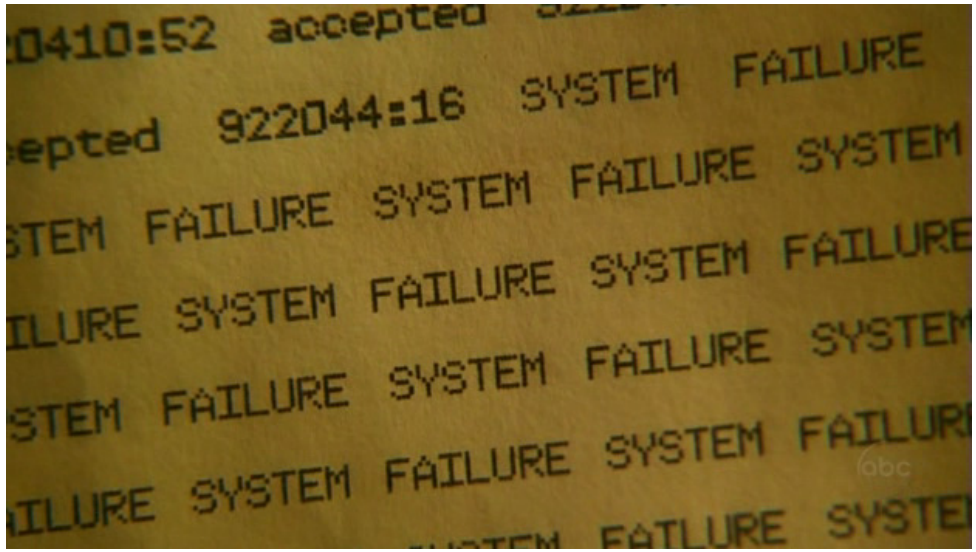
- Process omission : a process does not fulfill a request
- Channel omission: a communication link fails to deliver a message

Byzanthine failures

arbitrary behavior (e.g. a process giving the wrong result)

Timing failures

a synchronous system is violating one (or more) of its timing constraints



Detection mechanisms

heartbeat

timout

Security model

Security threats

Process

server : enemy can use intercepted credentials

client : enemy can pretend to be server

Channels

copy,change,replay, create messages

Privacy issues

Solution

Authentication/Authorization infrastructure

+ Crytographic solutions

(shared secret, asymmetric keys, ...)

Challenges for developing distributed app's

Challenge	Solution
Heterogeneity	
physical/L2 network	Common (standard) protocols
hardware	Virtualization
operating system	Virtual machins
programming languages	Middleware
Security	
privacy (e.g. financial/medical data)	cryptographic infrastructure
secure accounting	authorization/authentication
prove identity	virus scanning
viruses	firewall, VPN
(distributed) denial of service attacks	sandboxing
mobile code	
Scalability	
scaling of infrastructure cost physical cost = O(#users)	clever design automatic replication of components
performance scaling	
scaling of logical resources	
performance bottlenecks	

Challenges for developing distributed app's

Challenge	Solution
Failures	
failure detection	polling, heart beat, checksum,…
failure handling	masking : retransmit messages, fail over tolerate : inform user of (partial) failure recover : roll back to ensure consistency
high availability	redundancy and replication
Concurrency	
resource access conflicts	lock mechanisms synchronization primitives

Challenges for developing distributed app's

Challenge	Solution
Transparency	Middleware

Access transparency
identical operations to access local and remote resources

Location transparency
access of resources without location knowledge

Concurrency transparency
provide safe resource sharing amongst concurrent processes

Replication transparency
replicated resources are used implicitly

Failure transparency
hide hardware/software failure

Mobility transparency
resources/clients can move without affecting system operation

Performance transparency
system can be reconfigured according to load conditions

Scaling transparency
system can expand without changing structure and algorithms

Chapter 1

Distributed Software: Introduction

1. Definitions and Terminology
2. Developing distributed applications
3. **Architecture**
    1. Logical architecture
    2. System architecture
4. Design: middleware and services
5. Classes of distributed systems
6. Important architectures and platforms

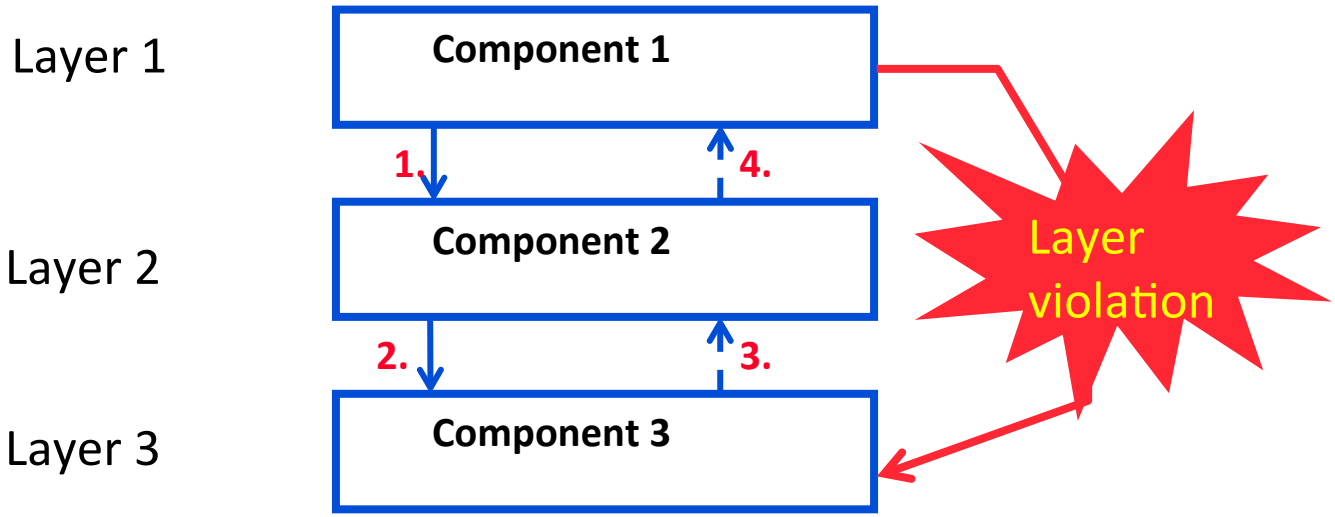
…

Logical architecture

= architecture capturing how subsystems will interact
also referred to as "architectural style"

Layered architecture

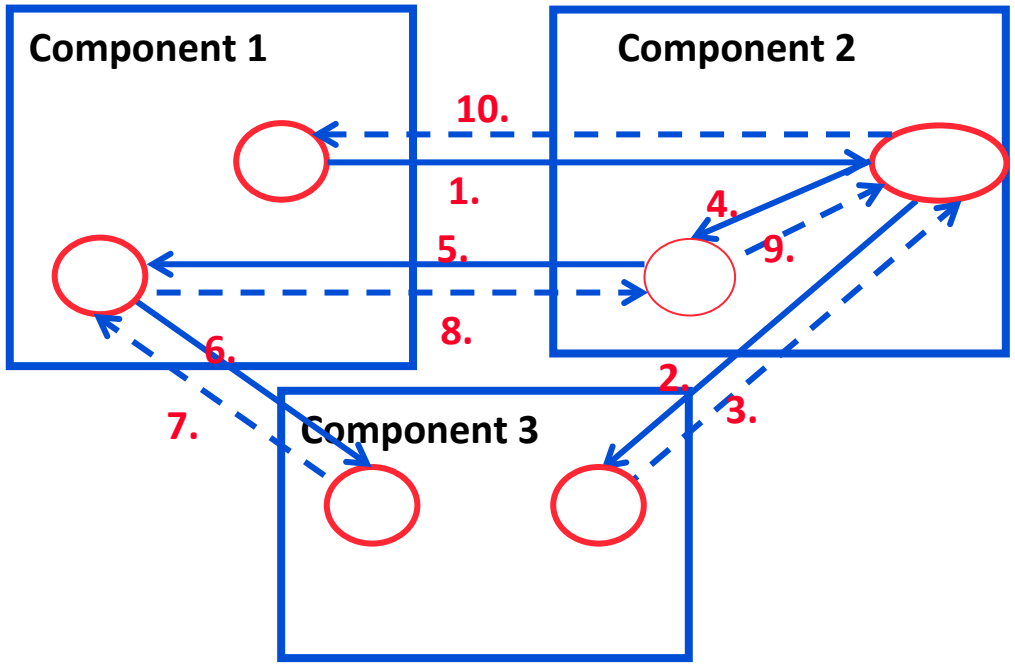
layer only interacts with neighbour
+ reduced number of interfaces, dependencies
+ easy replacement of a layer
- possible duplication of functionality



Logical architecture

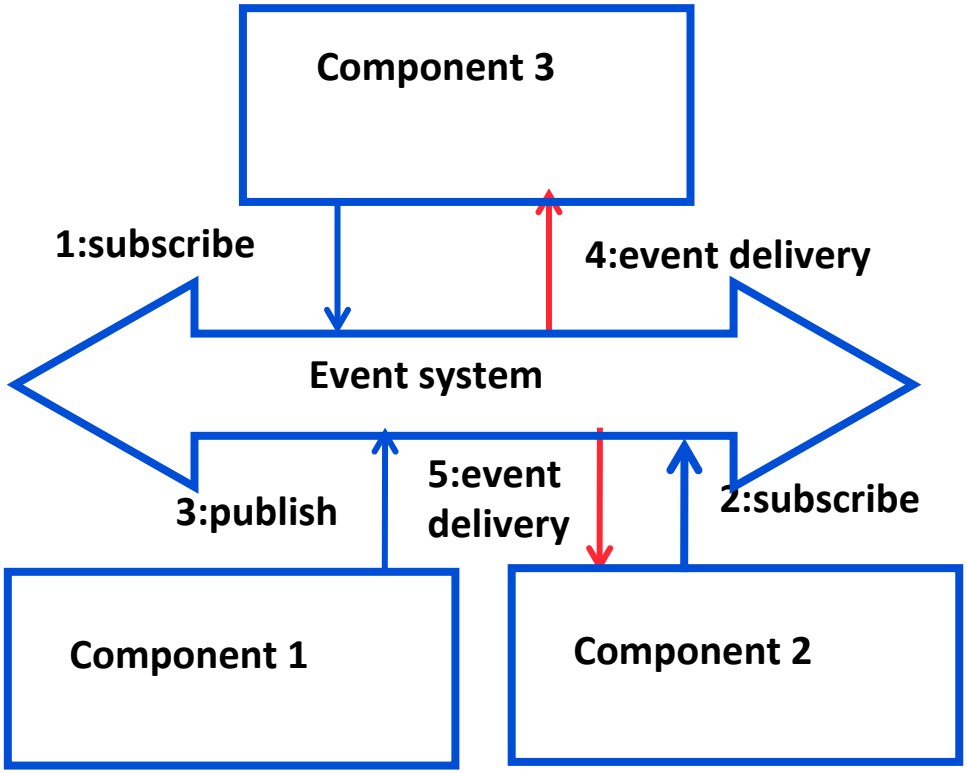
Interacting objects

no predefined interaction pattern
+ highly flexible
- complex to manage and maintain



Event based interaction

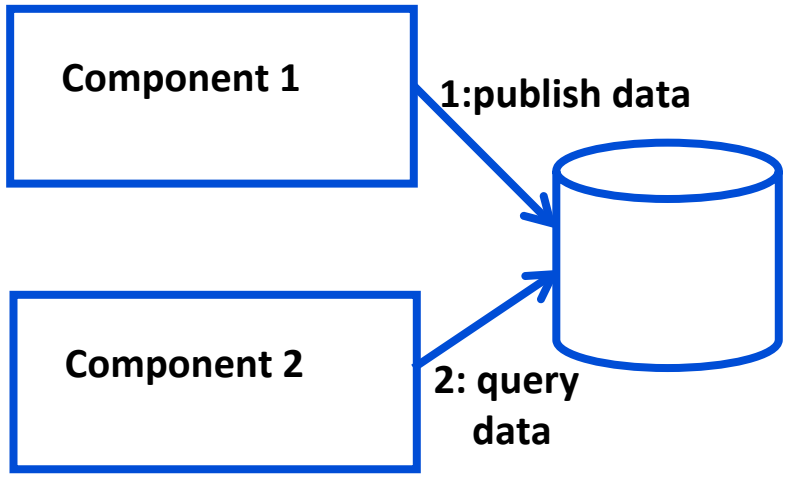
"publish-subscribe" style
+ loose coupling of components
related: message based interaction (also decoupling in time)
often used to integrate legacy systems



Logical architecture

Data centric architecture

only interaction through shared data base
+ loose coupling of components
- possibly slow (central bottleneck, locking, ...)

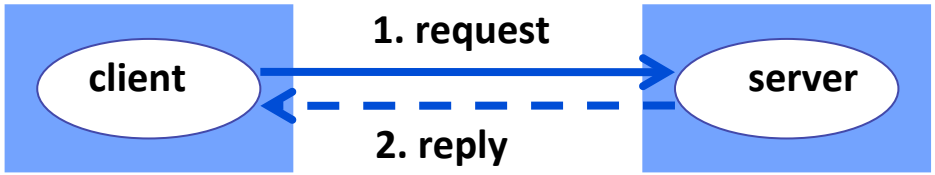


System architecture

How to map logical components to actually deployed components (replicated ?, P2P, pure client server, ...)

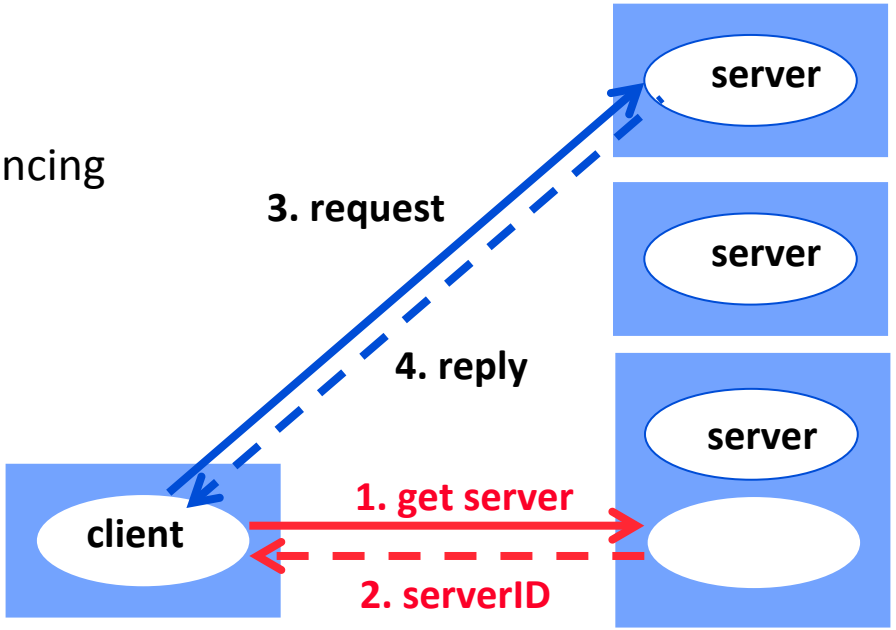
Client-server architectures

"simple" client-server



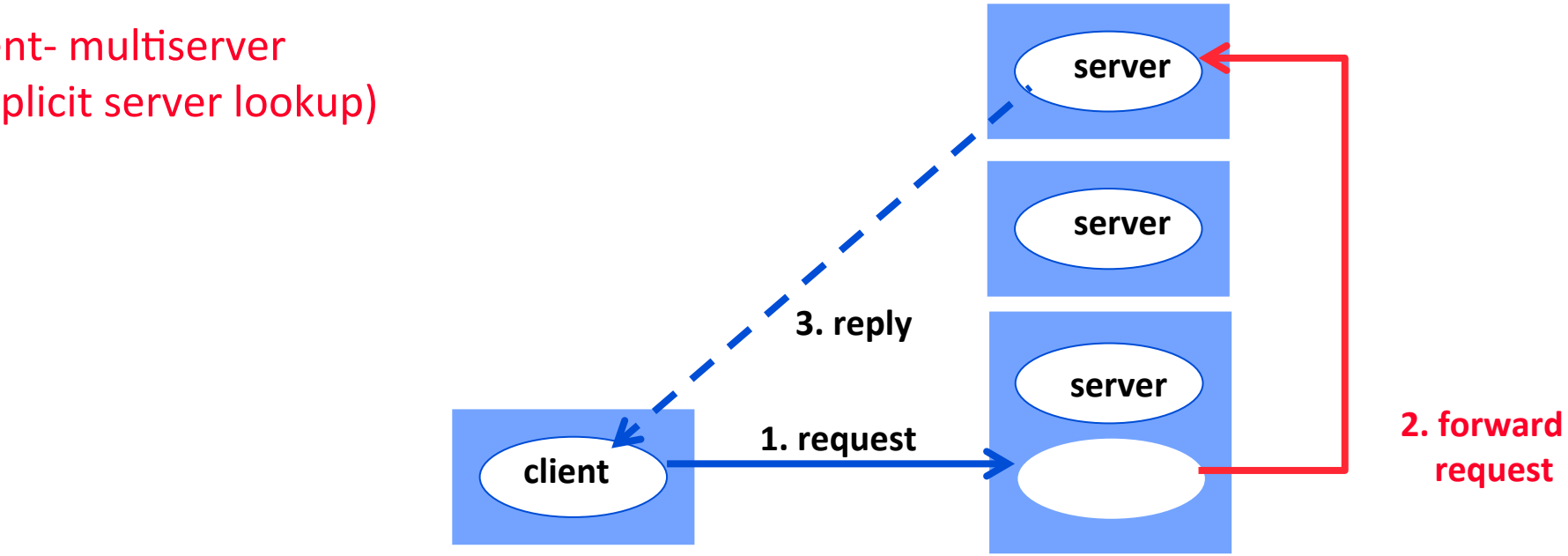
client- multiserver
(explicit server lookup)

e.g. DNS based load balancing

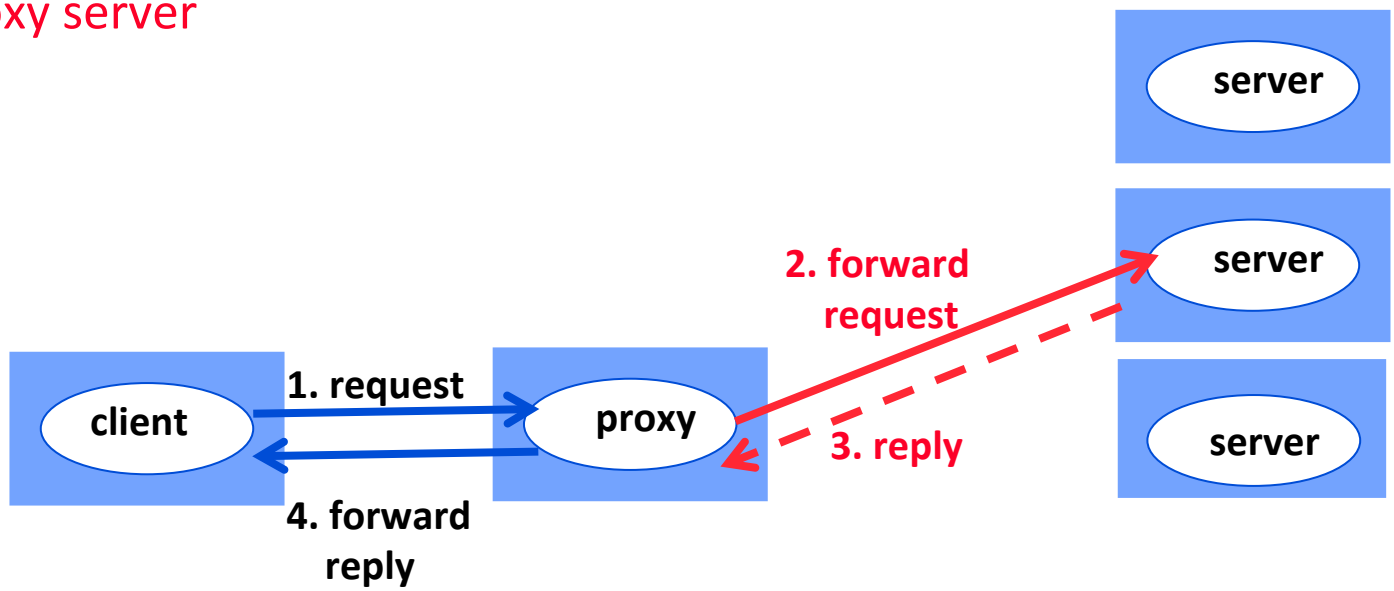


System architecture

client- multiserver
(implicit server lookup)



proxy server



P2P architectures

servent = process issueing requests + fulfilling requests
peering components use logical network (overlay) to interact

Unstructured P2P :

- links have no other meaning than communication
- unstructured search (e.g. flooding) : resource consuming

Structured :

- logical link has relation to service offered
- structured search much more efficient

important data structure : Distributed Hash Table (DHT)

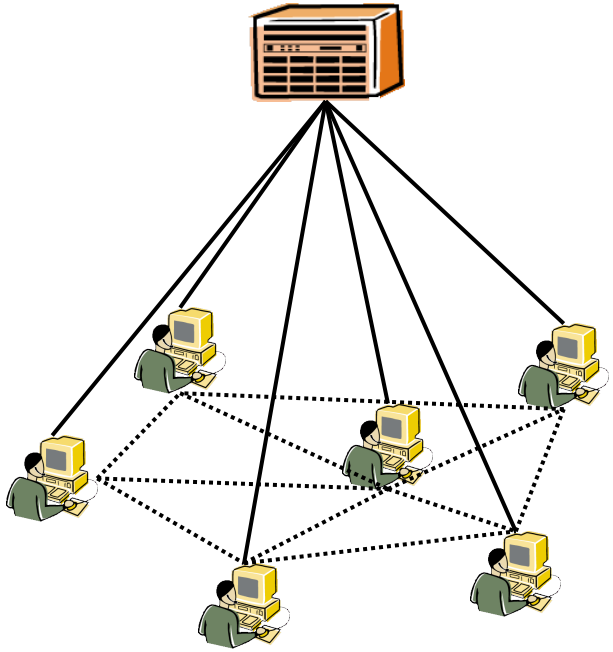
Why P2P ?

- scalability (10000 – 100000 users not exceptional)
- gradual scaling ("organic growth")
- robustness

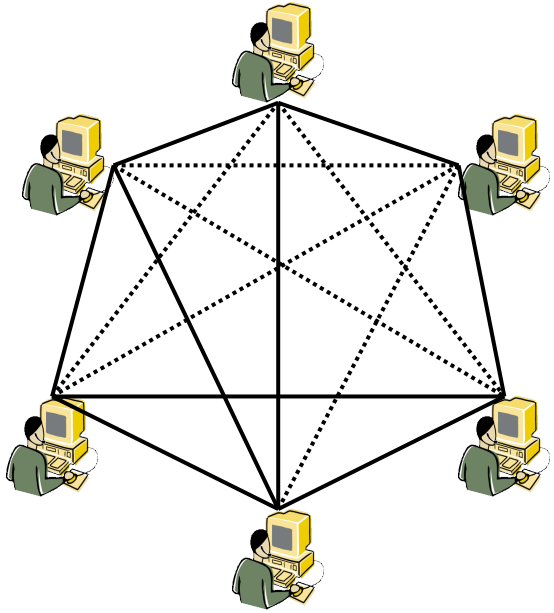
P2P : file sharing

Unstructured

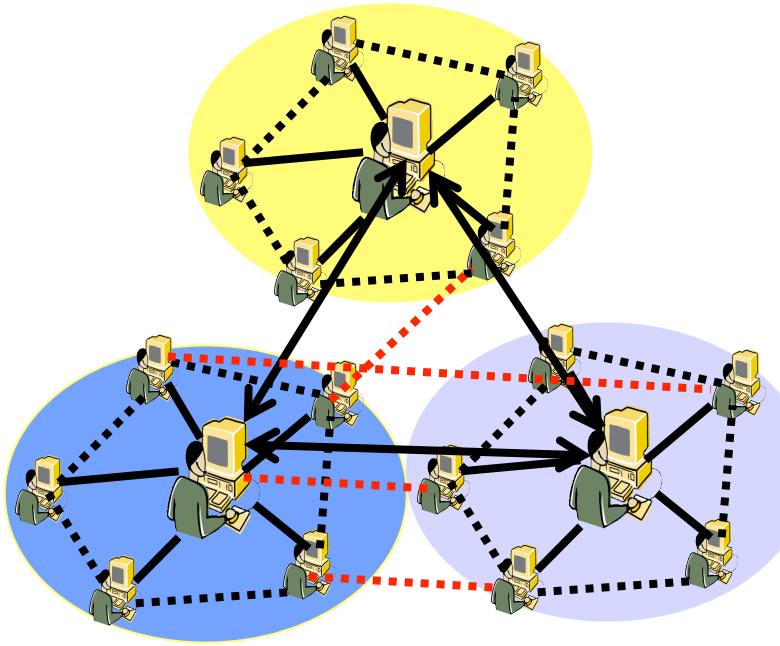
Mediated P2P	Pure P2P	Hybrid P2P



Napster
Audiogalaxy



Early Gnutella
FreeNet

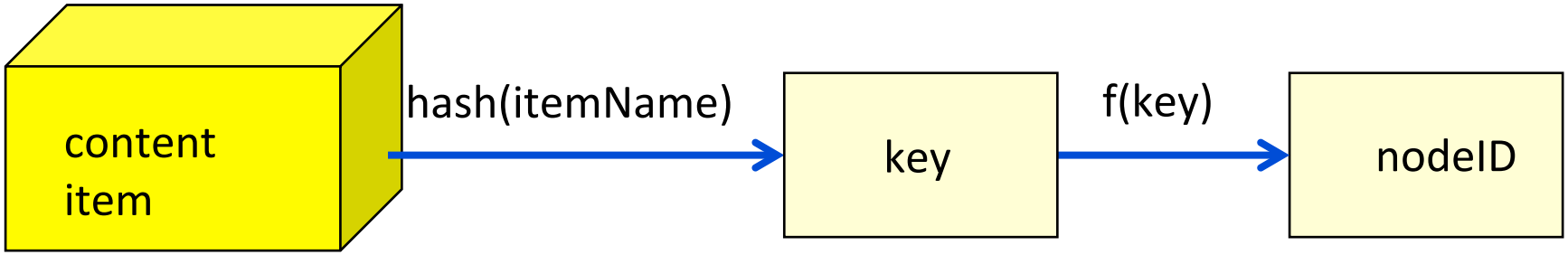


KazaA

control traffic (index, lookup)
data traffic (download, upload)

P2P : file sharing

Structured

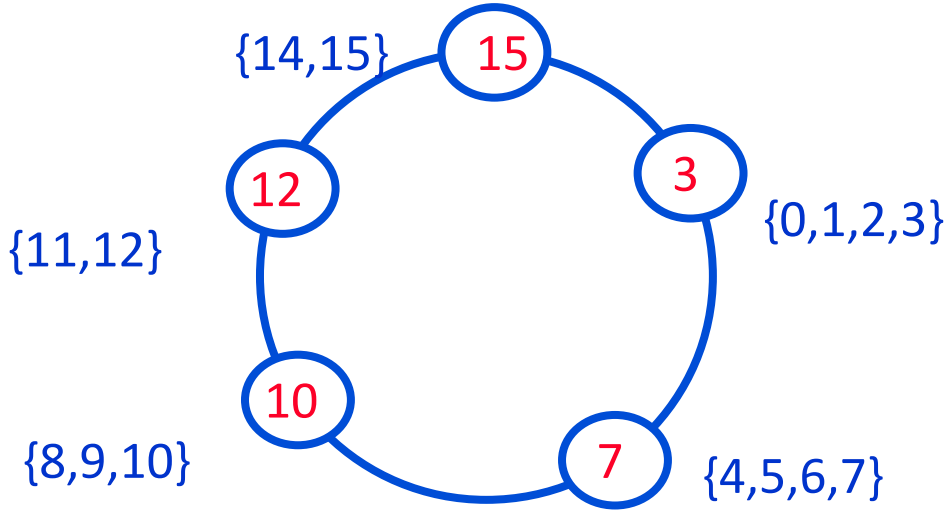


Chord

each node has integer nodeID

e.g. consider 5 node network {nodeID}={3,7,10,12,15}

{key}={0 .. 15}



see chapter "P2P" for more details

Chapter 1

Distributed Software: Introduction

1. Definitions and Terminology
2. Developing distributed applications
3. Architecture
4. **Design: middleware and services**
5. Classes of distributed systems
6. Important architectures and platforms

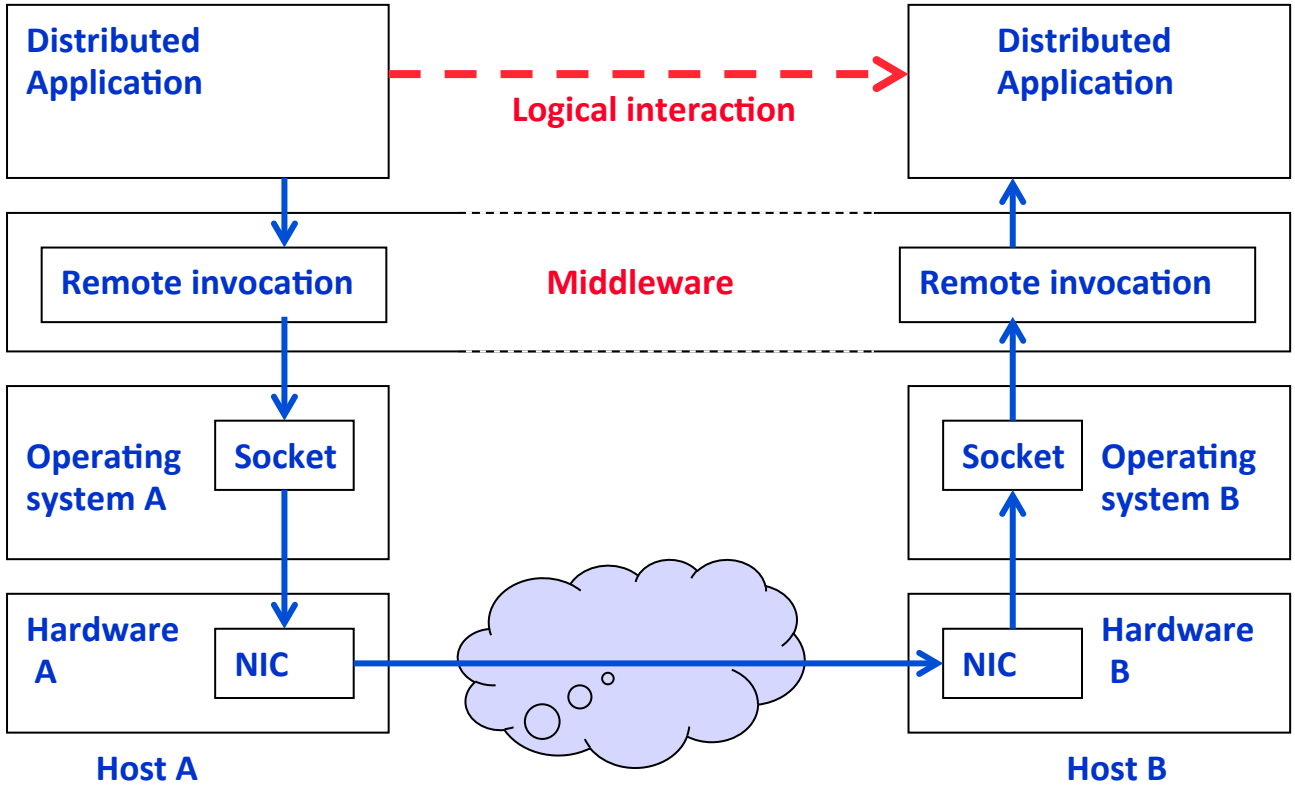
…

Role of middleware

1) Abstract hardware/software platform

- operating system
- implementation language
- ...

2) Realize transparency – focus on location transparency



Role of middleware

3) Provide generic services

Naming service

associate logical names to remote entities

Discovery and registration service

easy service lookup and registration

Transaction service

support distributed (database) transactions

Security service

support to authenticate and authorize users

Event service

distributed notification of events

Data replication

offer the same data at different locations to optimize data access

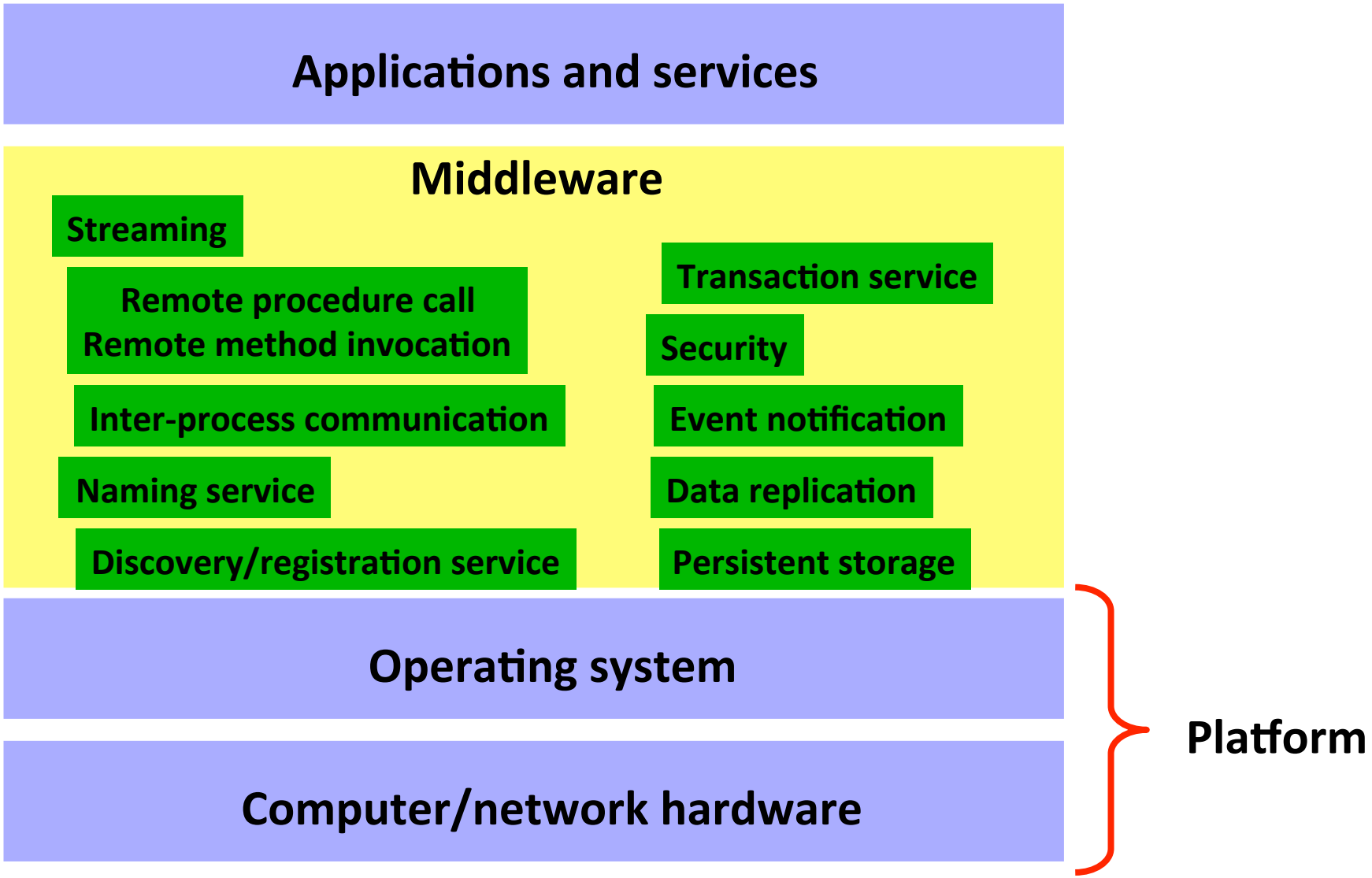
Persistence service

to transparently offer data persistence
(i.e. data is automatically stored in a persistent medium)

Life cycle service

managing service components
(start up of services, shut down, etc.)

Middleware Layer



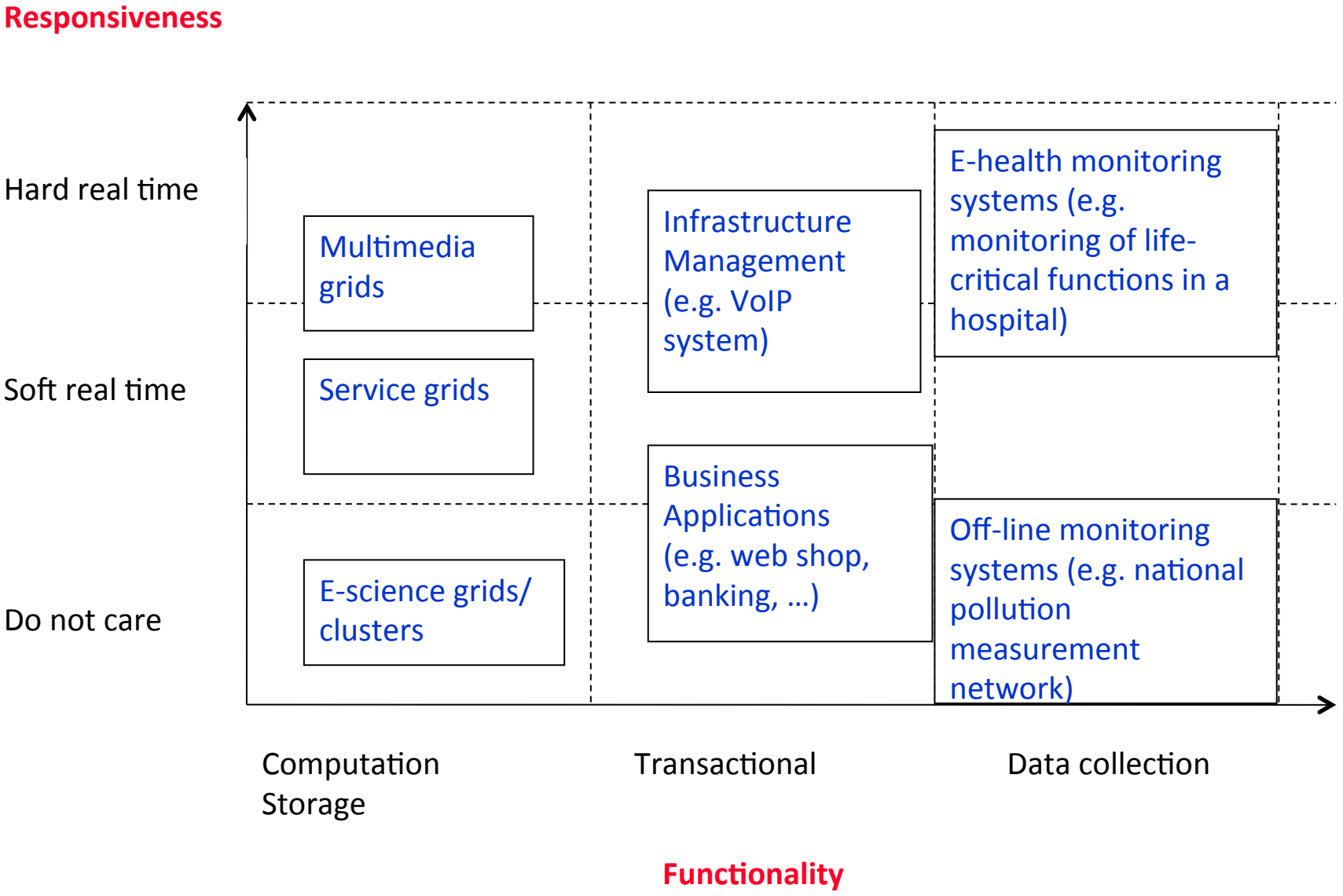
Chapter 1

Introduction

1. Definitions and Terminology
2. Developing distributed applications
3. Architecture
4. Design: middleware and services
5. **Classes of distributed systems**
6. Important architectures and platforms

…

Taxonomy


Responsiveness
Hard real time
Soft real time
Do not care
Multimedia grids
Service grids
E-science grids/ clusters
Infrastructure Management (e.g. VoIP system)
Business Applications (e.g. web shop, banking, …)
E-health monitoring systems (e.g. monitoring of life-critical functions in a hospital)
Off-line monitoring systems (e.g. national pollution measurement network)
Computation Storage
Transactional
Data collection
Functionality

Chapter 1

Distributed Software: Introduction

1. Definitions and Terminology
2. Developing distributed applications
3. Architecture
4. Design: middleware and services
5. Classes of distributed systems
6. **Important architectures and platforms**
    1. Thin versus thick ?
    2. Grid and cluster systems
    3. Transactional systems
    4. Integration
7. Scalability and high availability

…

How thick is the client ?

Thin client

+ cheaper and many devices involved -> overall cheaper
+ cheaper to update (controlled by server), and less updates needed
- some functions more natural to support on client
- permanent network connection needed

Technologies

- **web browser**
  - + ubiquitous and uniform client
  - + no network blockages (firewalls) for HTTP (port 80)
  - + huge user base
  - + browser functionality can be extended
- **remote desktop clients**
  - (cf. Athena) “screen scrapers”

Cluster middleware

- job scheduling
- cluster monitoring
- user administration
- matchmaking
- job submission

Important technologies: Oracle Fusion, Mosix

Grid middleware

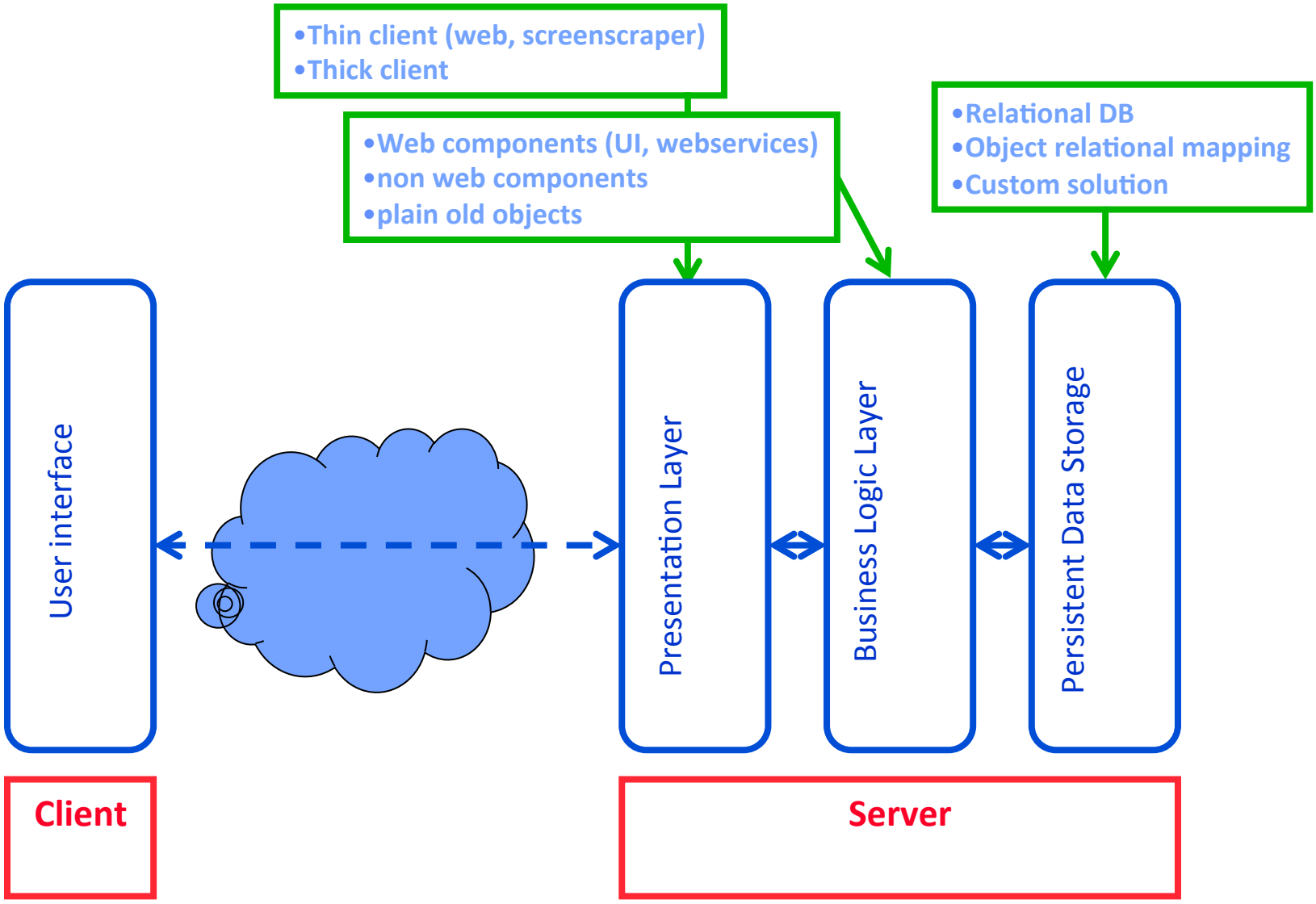
- meta-schedulling
- abstraction of heterogeneity
- information service

Important technologies :

Globus (and descendents), gLite (used in EGEE(+))

interacting jobs : Message Passing Interface (MPI)

3-tier architecture

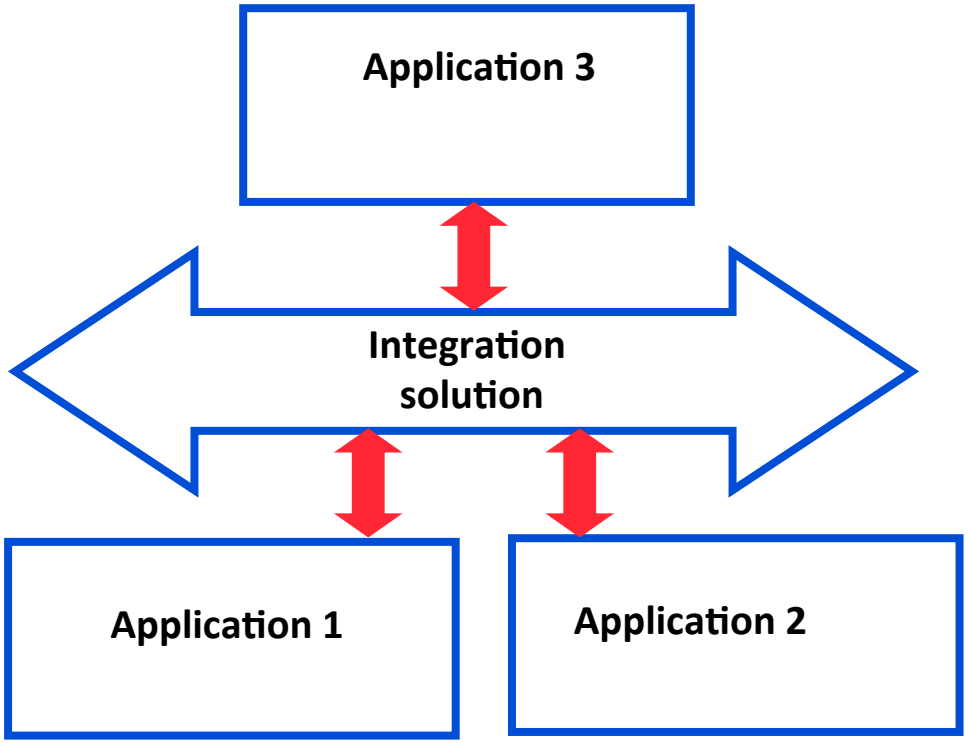


3-tier business architecture

layered architecture

rationale : separiation of concerns

Integration through bus



Integration solutions :

- Enterprise service bus (SOAP based)
- Message Oriented Middleware

Chapter 1

Distributed Software: Introduction

…

3. Architecture
4. Design: middleware and services
5. Classes of distributed systems
6. Important architectures and platforms
    1. Thin versus thick ?
    2. Grid and cluster systems
    3. Transactional systems
    4. Integration
7. **Scalability and high availability**
8. The Java technology family

…

Scalability

= the system ability to handle gracefully a growing amount of requests

two types of scalability:

- horizontal scalability
- vertical scalability

Vertical Scalability

adding more resources to a single computational node
improve existing code to handle more requests

Techniques:

- adding memory or disks, or more powerful CPU
- improving the I/O and concurrency models
  - using non-blocking and/or asynchronous I/O,
  - using the most appropriate thread model (i.e single, pool, per-connection),
  - using synchronous or asynchronous invocations.
- Distribution of processes over available processor cores

adding more computational nodes to distributed system

Techniques:

- Master/slave techniques
- Partitioning of data on the available nodes (e.g. Distributed Hash Tables)
- NoSQL-based systems in particular allow for horizontal scaling

- the property of a distributed system to handle hardware and software failures

Techniques:

- Redundancy: software components are available on several locations,
- Failover : when a software component or resource fails, automatic failover to the redundant components
- Replicas: multiple copies of the same data are available.

Very often the source of performance problems!

CAP theorem

A distributed computer system can NOT simultaneously provide all three of the following guarantees:

- Consistency: all read/write operations must result in a global consistent state,
- Availability: all requests on non-failed components must get a response,
- Partition Tolerance: the system continues to operate when nodes are not able to communicate with each other.

In practice:
- the Availability and Consistency guarantees are the most important
- Partition Tolerance or effects of network failures are often neglected.

Chapter 1

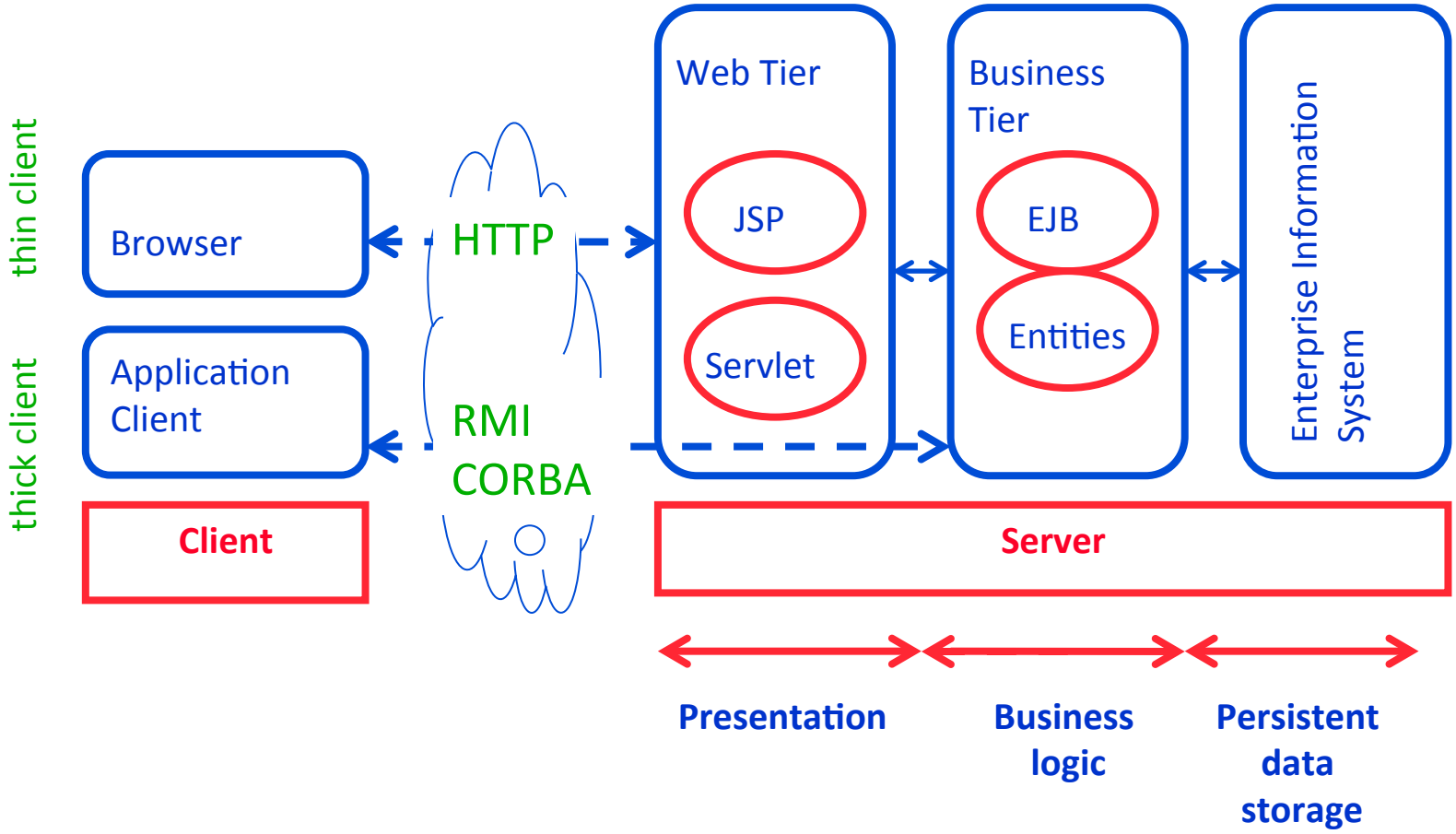
Distributed Software: Introduction

...

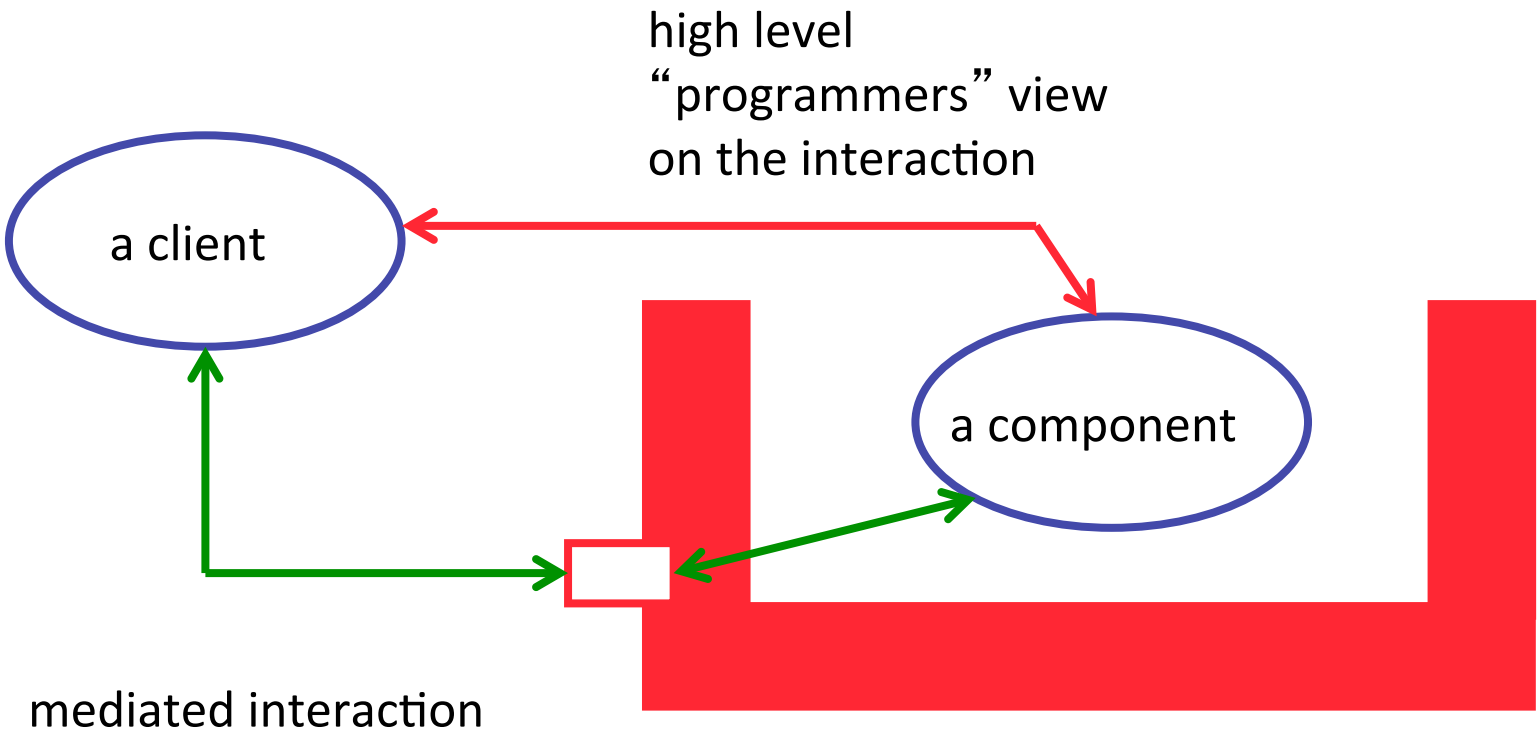
4. Design: middleware and services
5. Classes of distributed systems
6. Important architectures and platforms
7. Scalability and high availability
8. **The Java technology family**
    1. General architecture
    2. Client tier
    3. Web tier
    4. Business tier
    5. Web services

...

Java Enterprise Edition (JEE)



The container concept



mediated interaction
by container to
implement services
(e.g. security checks, load balancing, ...)

The container concept

Thin client

Container = browser

- runs HTTP protocol
- services
  - install plug-ins
  - run security checks
  - sandboxing

Thick client

Container = appclient

- runs RMI/CORBA/... protocol
- middleware services:
  - naming and lookup
  - security checking
  - ...

Servlet

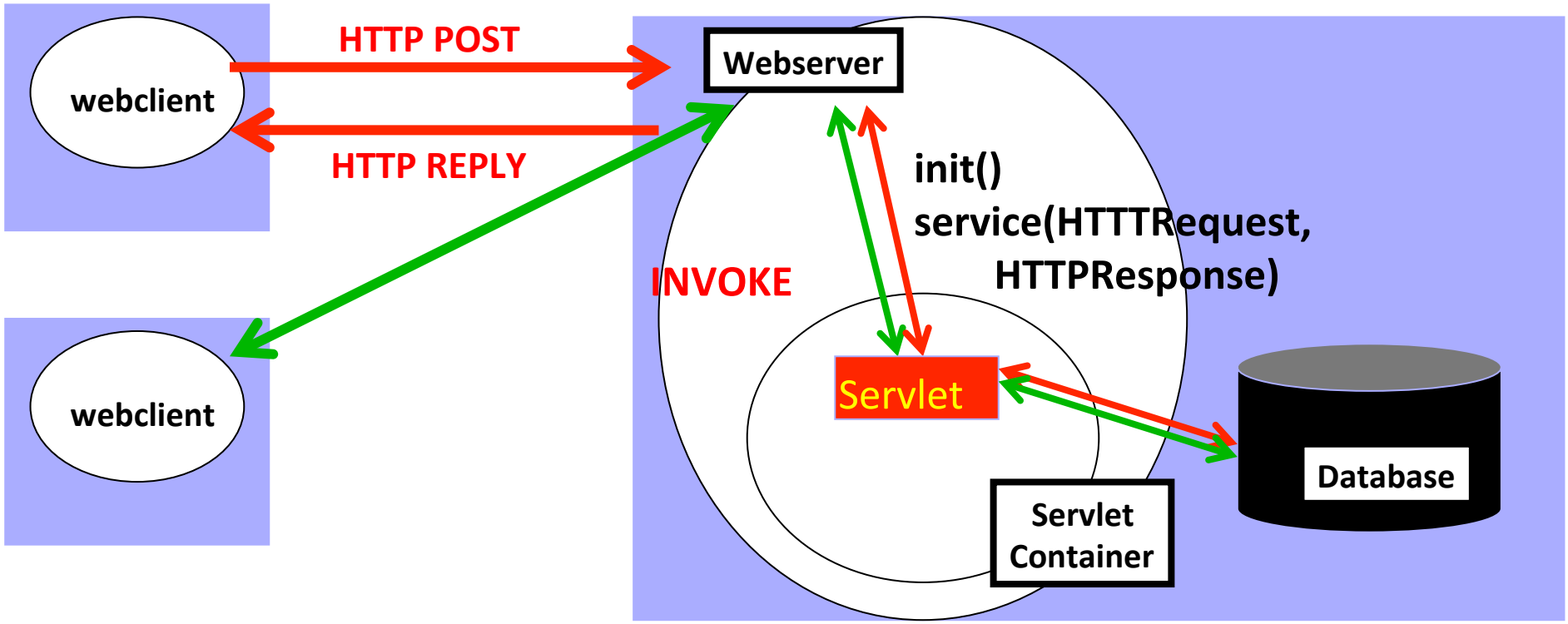
= server side plug-in
= base technology for all Java web tier components

Basic operation

1. browser sends request to URL referring to a Servlet instance
   HTTP request contains parameters, HTTP headers, ...
2. Servlet gets HTTPRequest object from web server
3. Servlet analyzes request
4. Servlet prints HTML output to HTTPResponse object
5. HTTPResponse is sent to webserver
6. Webserver sends HTML to client

Container services

- Servlet life cycle management
- Servlet pooling
- Session tracking
- Providing access to other resources (e.g. a database)
- Providing access to other components (other Servlets, EJBs, …)



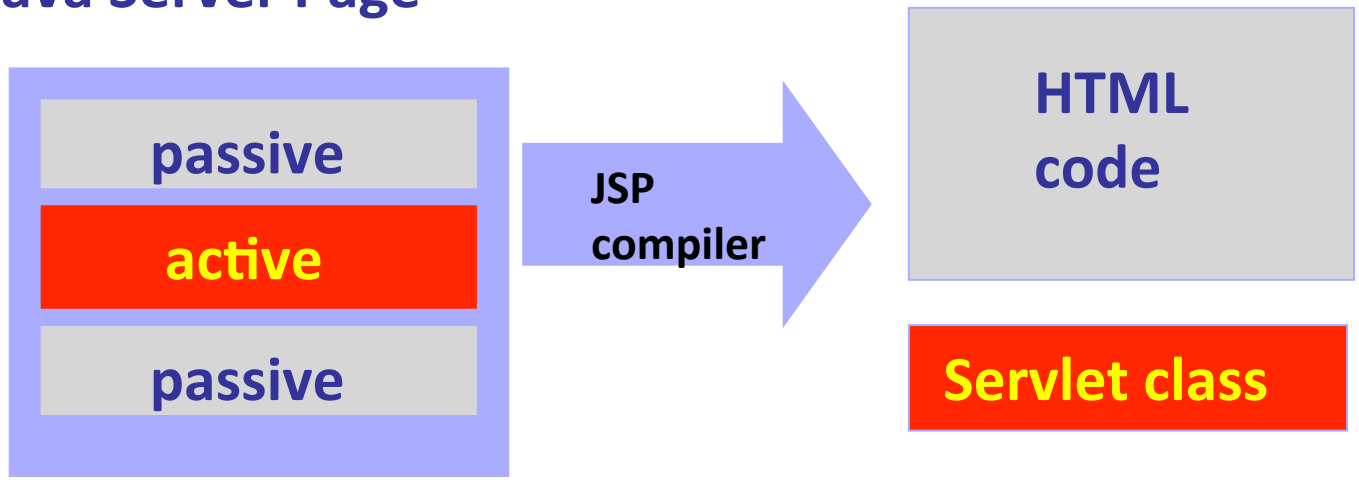
Servlet disadvantage

- pages contain static + dynamic data
- static data created dynamically -> not optimal

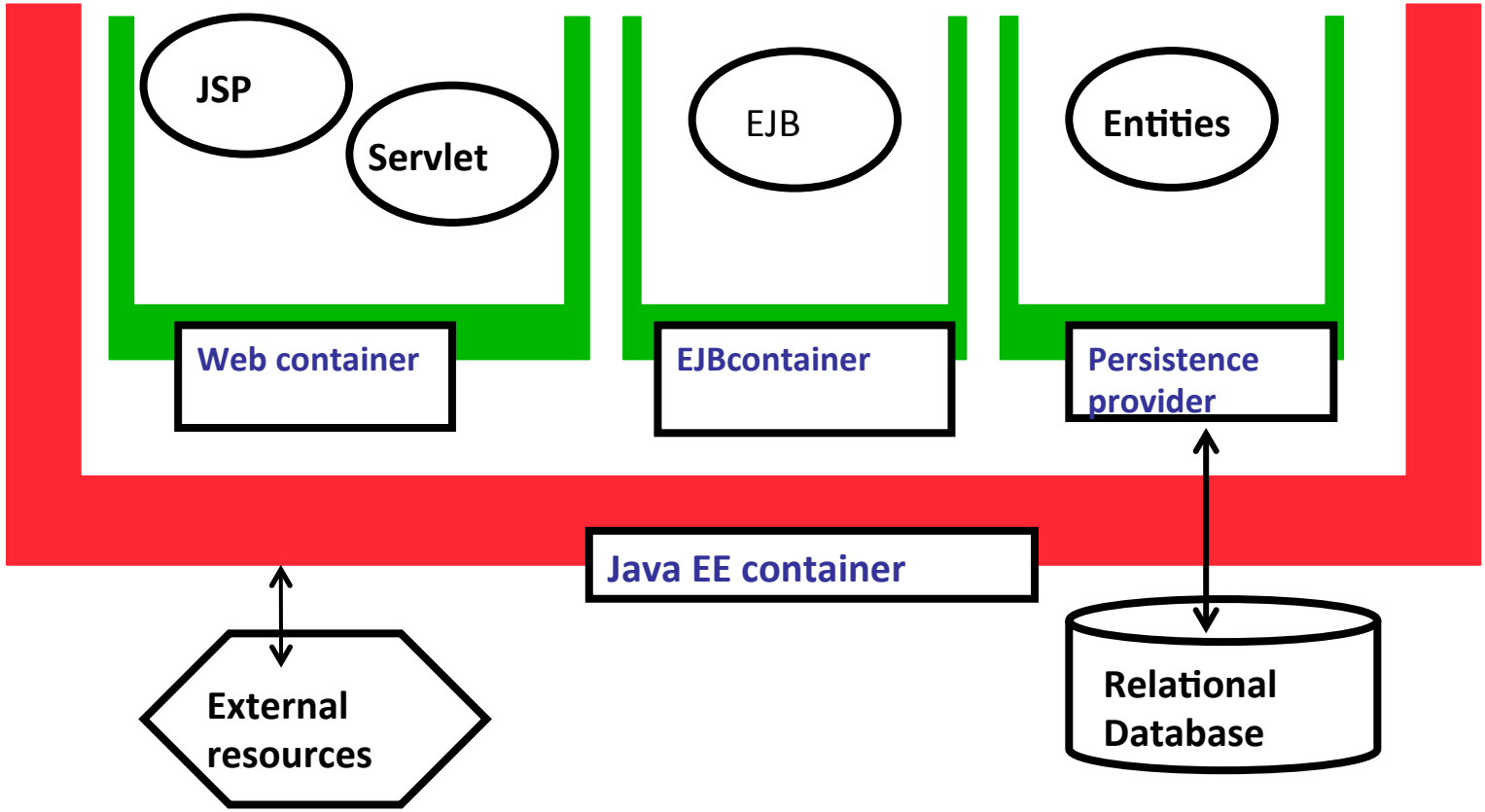
Solution

- mix static and dynamic code in single webpage
- compile webpage to extract servlets and static portions
- configure webserver to
  - automatically invoke servlets to create dynamic content
  - merge dynamic and static content to response to client

JSP : Java Server Page



Beans and entities



Beans and entities

EJB container services

- component life cycling management,
- transaction processing,
- security handling,
- persistence
- remotability
- timer
- state management
- resource pooling
- messaging

Persistence provider services

- store/retrieve from DB
- optimization through caching

Beans and entities

Session Beans

- session related object
- always associated to one single client at most
- types
  - **stateful** : "conversational state"
  - **stateless**
  - **singleton**

synchronous

Message Beans

- asynchronous message handling
- connected to JMS compliant message queue
- new since J2EE 1.3

asynchronous

Entities

- "real life" object
- mostly associated to "row in database"
- persistent
- NEW since EJB3.0 **[replace (very) complex EntityBeans]**
- NOT managed by EJB container

Web services

What ?

= “interface” technology
specifies how to access remote objects through the web
typically : SOAP (Simple Object Access Protocol) or
REST (Representational State Transfer) over HTTP

Pro – con ?

+ language and platform neutral
+ often uses HTTP as underlying protocol (no network discontinuities)
- performance (large messages, complex parsing)
- only stateless invocations

JEE and web services

Pojo’s and session beans can be exposed as webservice

Chapter 1

Distributed Software: Introduction

...

4. Design: middleware and services
5. Classes of distributed systems
6. Important architectures and platforms
7. Scalability and high availability
8. The Java technology family
9. **The .NET technology family**

...

Very similar functions as in JEE

Web tier

- web service (SOAP / REST) heavily used
- ASP is counterpart for JSP
- .NET framework for web tier development : ASP.NET

Business tier

- COM+ component model ported to .NET
- “.NET enterprise services”
  - resource pooling
  - eventing
  - security
  - transaction support
  - synchronization

Chapter 1

Distributed Software: Introduction

...

4. Design: middleware and services
5. Classes of distributed systems
6. Important architectures and platforms
7. Scalability and high availability
8. The Java technology family
9. The .NET technology family
10. This course

Outline

Chapter 1: Introduction
Chapter 2: Middleware
Chapter 3: Enterprise Applications
Chapter 4: Global State and Time
Chapter 5: Coordination
Chapter 6: P2P systems
Chapter 7: Cloud Computing
Chapter 8: Resource Allocation