Software Architectures

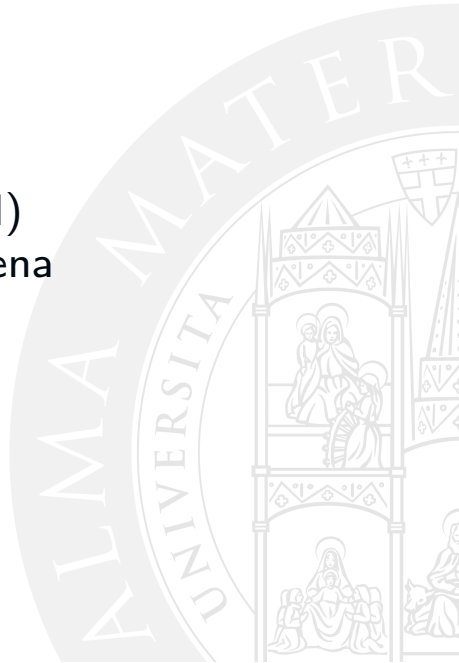
Distributed Systems

Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

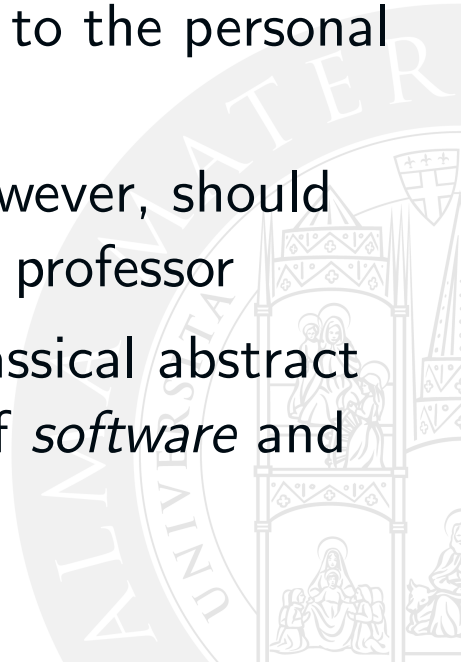
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018



About these Slides

- the following slides borrow a great deal from the slides coming with the book adopted as the basic one for this course [Tanenbaum and van Steen, 2007], as well as from Fielding's PhD Thesis [Fielding, 2000]
- material from those slides (including pictures) has been re-used in the following, and integrated with new material according to the personal view of the professor
- every problem or mistake contained in these slides, however, should be attributed to the sole responsibility of this course's professor
- the goal for students, here, is to be exposed to the classical abstract view on distributed systems provided by the notions of *software* and *system architecture*



Next in Line...

Software Architectures to Handle Complexity

Distributed systems are complex

- in order to manage their intrinsic complexity, distributed systems should be properly *organised*
- organisation of a distributed system is mostly expressed in terms of its *software components*

Software architectures expresses component organisation

- there are many ways to organise components of a distributed system, classified as software architectures
- there are many possible *instantiations* of a software architecture, where components have their actual *place* in the distributed system—often called system architectures

Architectural Styles to Classify Systems

An *architectural style* is formulated in terms of...

- *components*
- the way in which components are *connected* to each other
- the *data* flowing through the components
- the way in which all the above things are *configured* altogether to build the system

The notion of architectural style...

- encompasses a way to cluster and classify groups of similar systems—that is, having the *same* sort of *organisation*
- allow distributed systems to be *compared*
- but also provide general *patterns* for their overall *design*

Components & Connectors I

Components

- a component is a modular unit with well-defined *interfaces*
- which is *replaceable* within its environment
- interfaces are both *required* and *provided*—both ways, then

Connectors

- a connector is an abstraction *mediating* communication, coordination, cooperation among components
- that is, anything providing a *mechanism for interaction* among components

Components & Connectors II

Putting together components and connectors . . .

- produces a huge range of possible organisations and configurations
- that are then classified in terms of architectural styles

What is a Software Architecture?

Software architecture

A software architecture is an abstraction of the run-time elements of a software system during some phase of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture. [Fielding, 2000]

Architectural elements

A software architecture is defined by a configuration of architectural elements – components, connectors, and data – constrained in their relationships in order to achieve a desired set of architectural properties. [Fielding, 2000]

Architectural Elements

Components

A component is an abstract unit of software instructions and internal state that provides a transformation of data via its interface

Connectors

A connector is an abstract mechanism that mediates communication, coordination, or cooperation among components

Data

A datum is an element of information that is transferred from a component, or received by a component, via a connector

Architectural Properties & Constraints

Architectural properties

The set of architectural properties of a software architecture is derived from the *selection* and *arrangement* of components, connectors, and data within a system

- *functional properties*
- *quality attributes* such as ease of evolution, reusability of components, efficiency, and dynamic extensibility

Properties are induced by the *set of constraints* within an architecture

Architectural constraints

Architectural constraints are often motivated by the application of a software engineering principle to an aspect of the architectural elements

Next in Line...

Architectural Styles

Architectural Style

An architectural style is a coordinated set of architectural constraints that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style [Fielding, 2000]

Architectural styles

- are a mechanism for categorising architectures and defining their common characteristics
- provide an abstraction for the interactions of components, capturing the essence of a pattern of interaction by ignoring the accidental details of the rest of the architecture

Main Examples of Architectural Styles

Identification of architectural styles

- architectural styles – like patterns in software engineering – are to be *devised out* rather than invented
- today, four different architectural styles have been identified as the main ones for distributed systems

Main architectural styles for distributed systems

- layered architectures
- object-based architectures
- data-centered architectures
- event-based architectures

\+ shared data-space architectures

Layered Architectures I

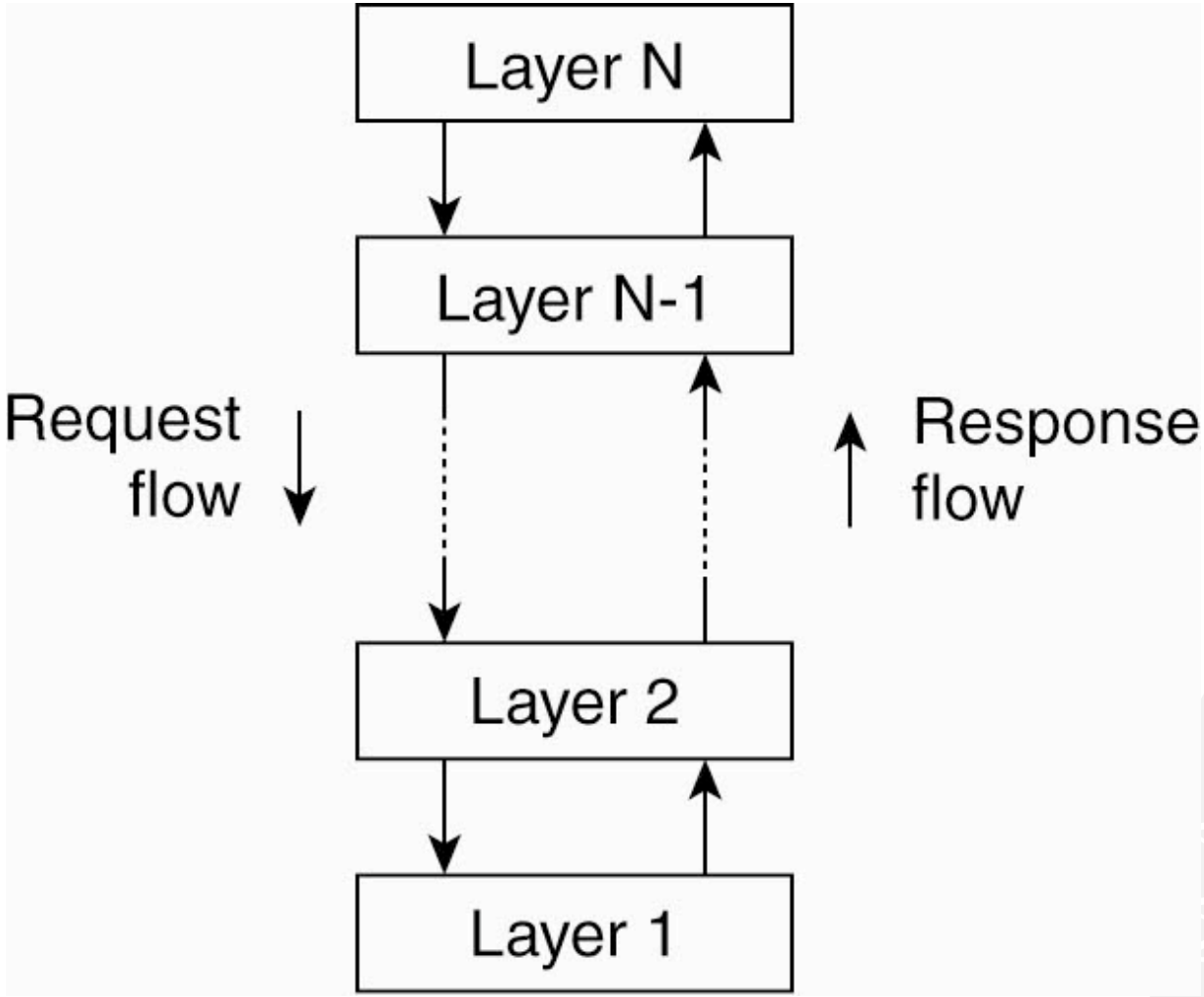
Basic idea

- components are organised in a *layered fashion*
- where components of a layer *only* call components of the layer below, and are *only* called by the components of the layer above

Data flow

- the request-response flow is always top-down / bottom-up
- control flow follows the same pattern along with data

Layered Architectures II



[Tanenbaum and van Steen, 2007]

Object-based Architectures I

Basic idea

- components are *objects*
- components are connected through a *RPC mechanism*

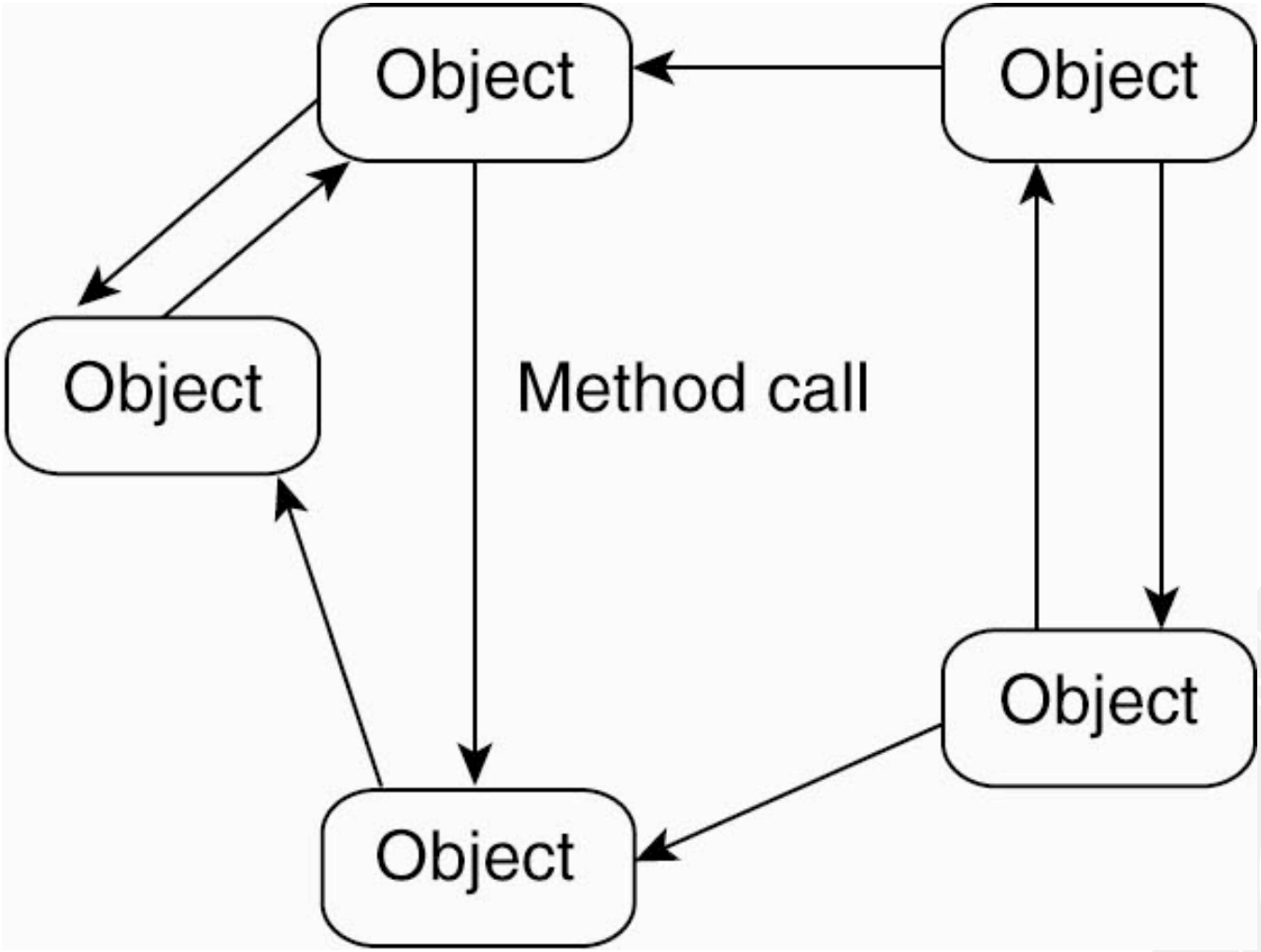
Client-server architectures

- . . . are built out of this style

Layered and object-based architectures

- . . . are the most important styles for distributed systems today
- however, a lot of things are going to happen in the future, which may change such an overall picture

Object-based Architectures II



[Tanenbaum and van Steen, 2007]

Data-centred Architectures I

Basic idea

- communication among processes occurs through a shared repository
- the repository might be either passive (reactive) or (pro)active

Main features

- ... depends on the choice made for the shared repository
  - how *information* is represented
  - how *events* are handled
  - how the *shared repository* behave in response to interaction
  - how *processes* interact with / through the shared repository

Data-centred Architectures II

Examples are everywhere

- web-based systems, for instance, are largely data-centric
- also, many distributed applications still work by sharing files around the network

Event-based Architectures I

Basic idea

- processes communicate through an event bus
- through which *events* are propagated
- possibly carrying *data* along

Main example: Publish/subscribe systems

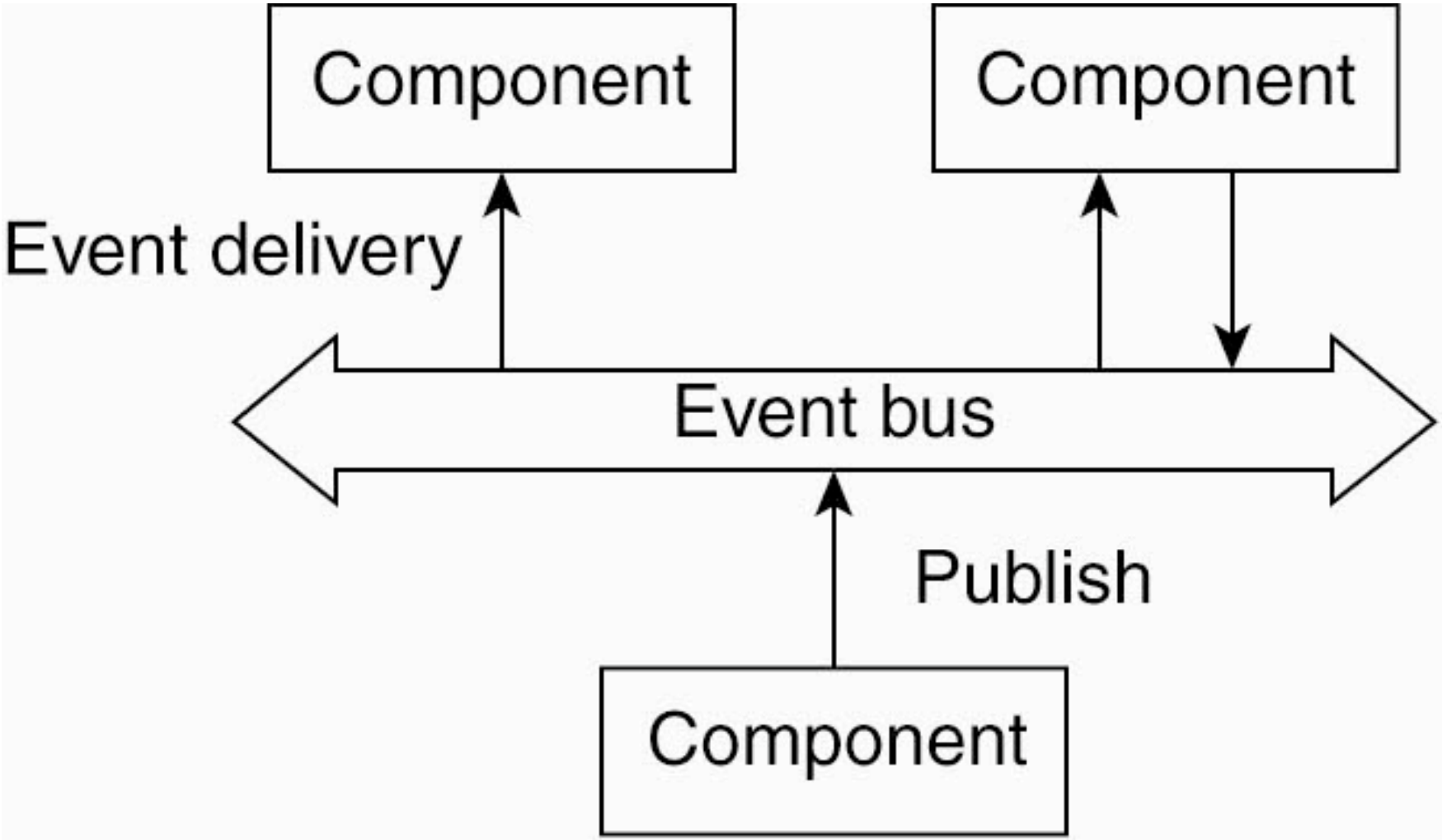
- *publishers* publish events through the middleware
- *subscribers* receive events to which they have subscribed

Event-based Architectures II

Main feature

- processes can communicate with no need to *reference* each other / to know each other—they are *referentially uncoupled*
- processes can communicate with no need to *share* the same space—they are *uncoupled in space*

Event-based Architectures III



[Tanenbaum and van Steen, 2007]

Shared Data-space Architectures I

Basic idea

- putting together Data-centric and Event-based architectures
- the shared repository is a shared persistent data-space, and also an event bus
- where data is stored and accessed
- along with related events

Main example: Blackboard systems

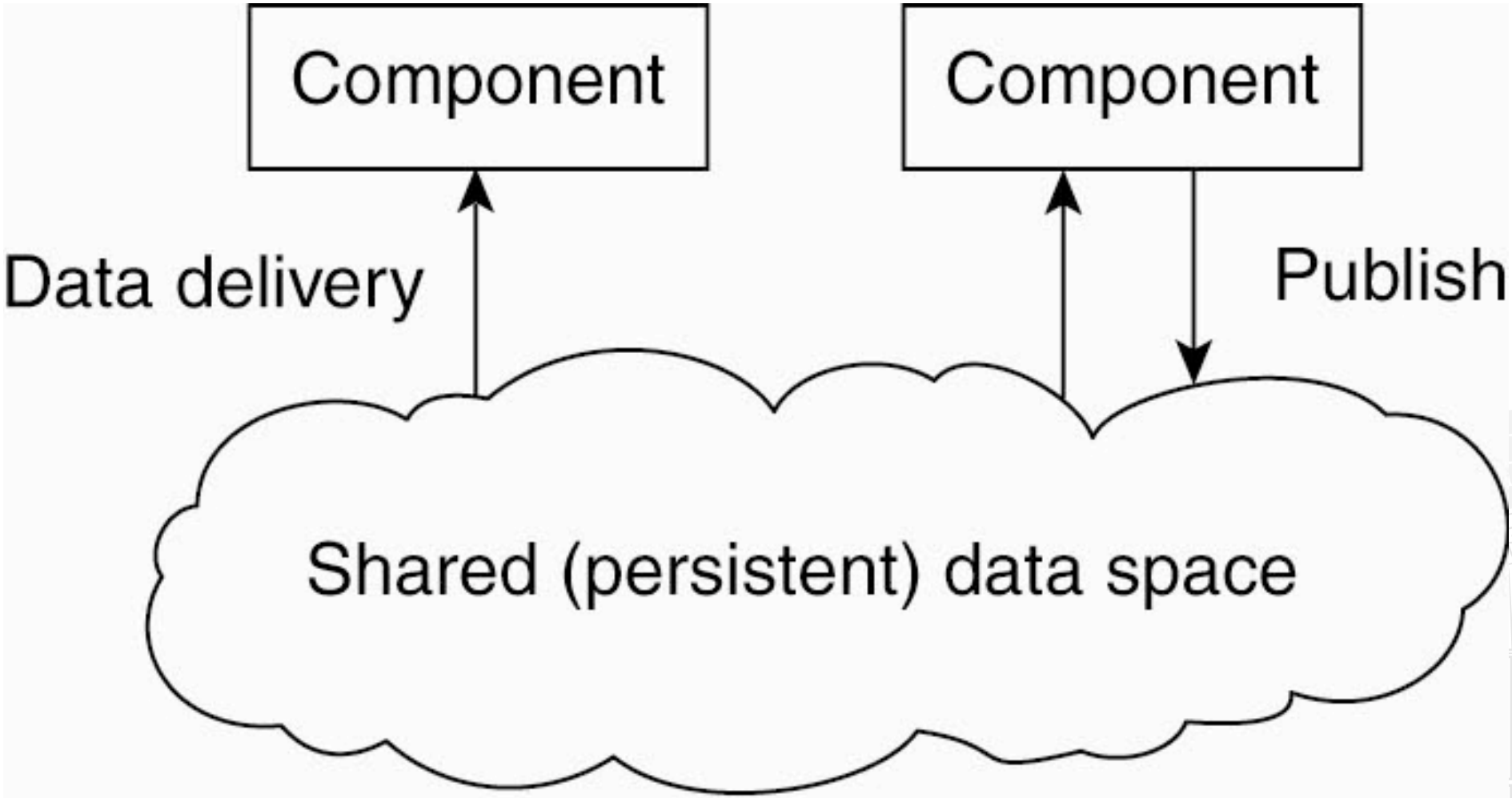
- processes put data in the blackboard
- the blackboard aggregates *knowledge*, implements *policies*, and drive the *coordination* of processes

Shared Data-space Architectures II

Main feature

- processes can communicate with no need of compresence
- processes are also *uncoupled in time*

Shared Data-space Architectures III



[Tanenbaum and van Steen, 2007]

Next in Line...

Where are Software Components?

In a distributed system, distribution of components matters

- when a software architecture is actually instantiated, components are *placed somewhere* in a distributed system
- this is typically taken as an instantiation of a software architecture as a system architecture

Sorts of system architectures

- centralised architectures
- decentralised architectures
- \+ hybrid architectures

Clients & Servers

Main feature

- in a centralised architecture, *clients* request *services* from *servers*—and that is all, more or less
- in the basic *client-server* model, processes are classified in two groups—obviously, *clients* and *servers*
- possibly, the two groups may overlap

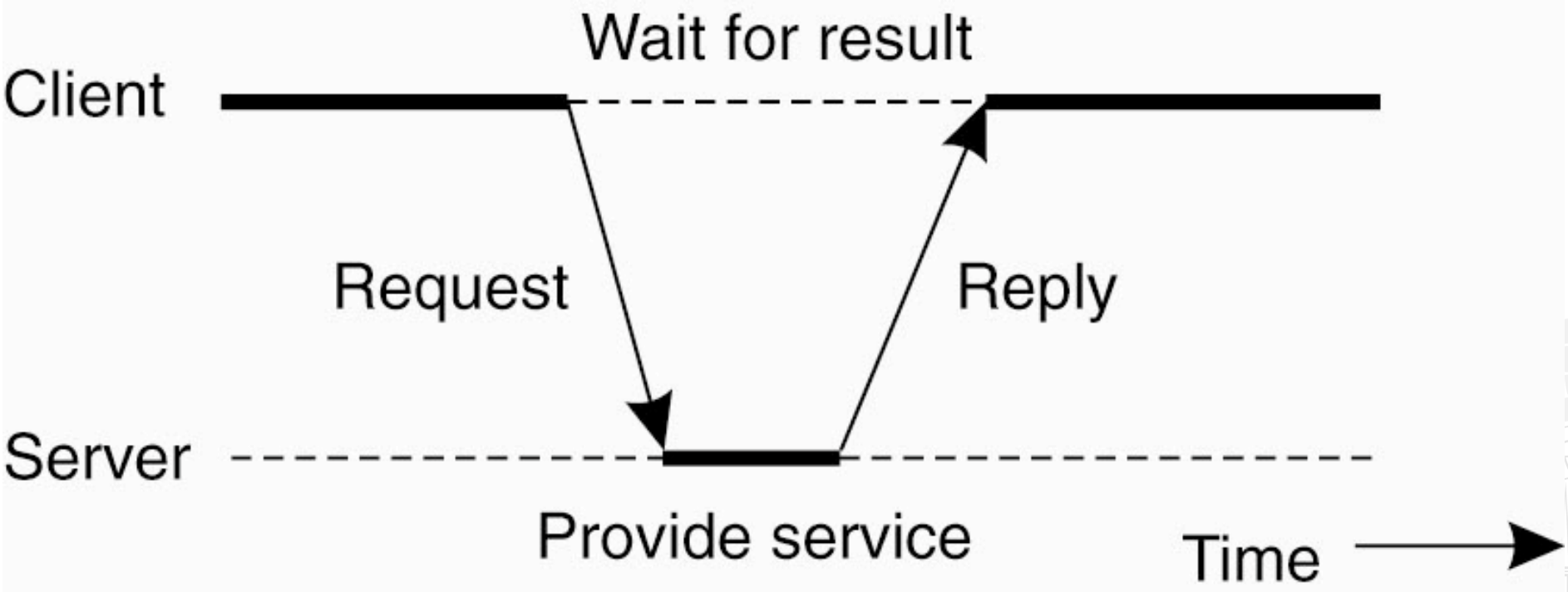
Servers

A server is a process implementing a specific service—like, say, a database service

Clients

A client is a process requiring a specific service from a server

Client-server Interaction



Basic scheme for client-server interaction: *request-reply behaviour*

[Tanenbaum and van Steen, 2007]

Client-Server Communication I

Stateless communication

- each client-server interaction occur independently of each other, with the request containing all information required to the server to reply properly
- no need for the communication to set any pre-defined *channel* for communication
  - also called *connectionless* communication—e.g., IP, UDP, HTTP are *connectionless protocols*
- no need for the server to keep track of previous interactions—no *state* for communication

  → server are simpler in structure and behaviour, and more efficient
- no way for the client to exploit previous interactions for communication—every request contains all information

  → requests from clients take more bandwidth

Client-Server Communication II

Stateful communication

- each client-server interaction occur through some sort of pre-set *communication channel*, with the request containing only information required to each single interaction
  - also called *connection-oriented* communication—e.g., telnet, rlogin, FTP, TCP are *connection-oriented protocols*
- the server keeps track of previous interactions through the channel—communication has a *state*

  → server are complex in structure and behaviour, and less efficient
- clients can exploit previous interactions for communication—every request contains just the minimal information required

  → requests from clients take less bandwidth

Client-Server Communication III

Efficiency vs. reliability

- stateless communication is ok for *idempotent* operations
  - that is, operations that could be repeated more than once without harm
- stateful communication is less efficient, but ensure reliability
  - for instance, Internet protocols are typically based on TCP—reliable but relatively costly for small-grain communication

Application Layering

Logical layering in client-server architectures

user-interface level contains the *interface with the user*

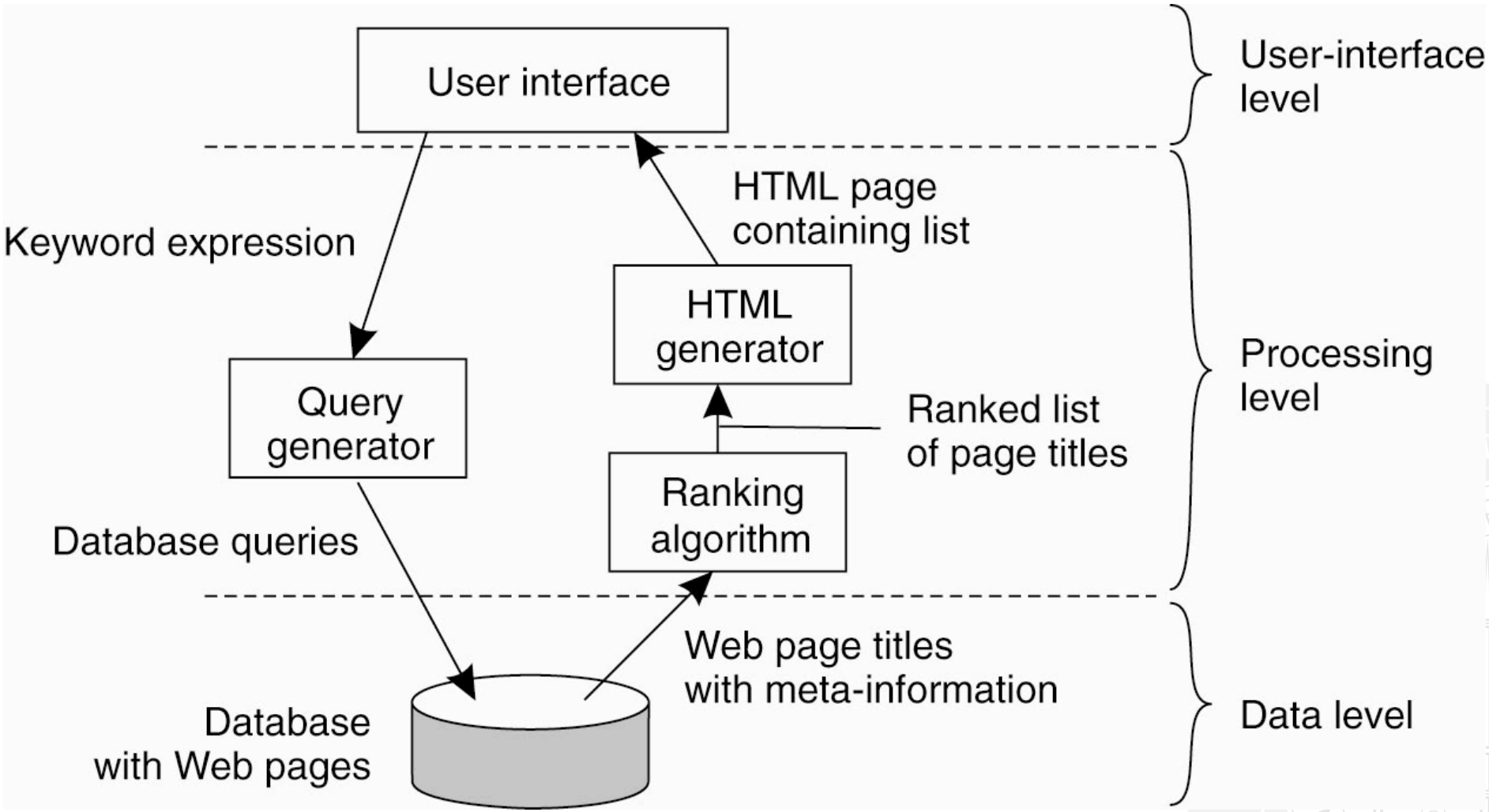
processing level contains the *logic of the control*, in short, the core of the applications

data level manages the actual *data that are relevant* to the applications

Typical organisation for client-server applications

- with a part handling user interaction
- a part dealing with data and files
- and a part containing the core functionality of an application

Example: Internet Search Engine



Simple layering of an Internet search engine [Tanenbaum and van Steen, 2007]

Multi-tiered Architectures

How to physically distribute logical layers?

- *logical* organisation is not *physical* organisation
- clients and servers could be placed on the same node, or be distributed according to several different *topologies*

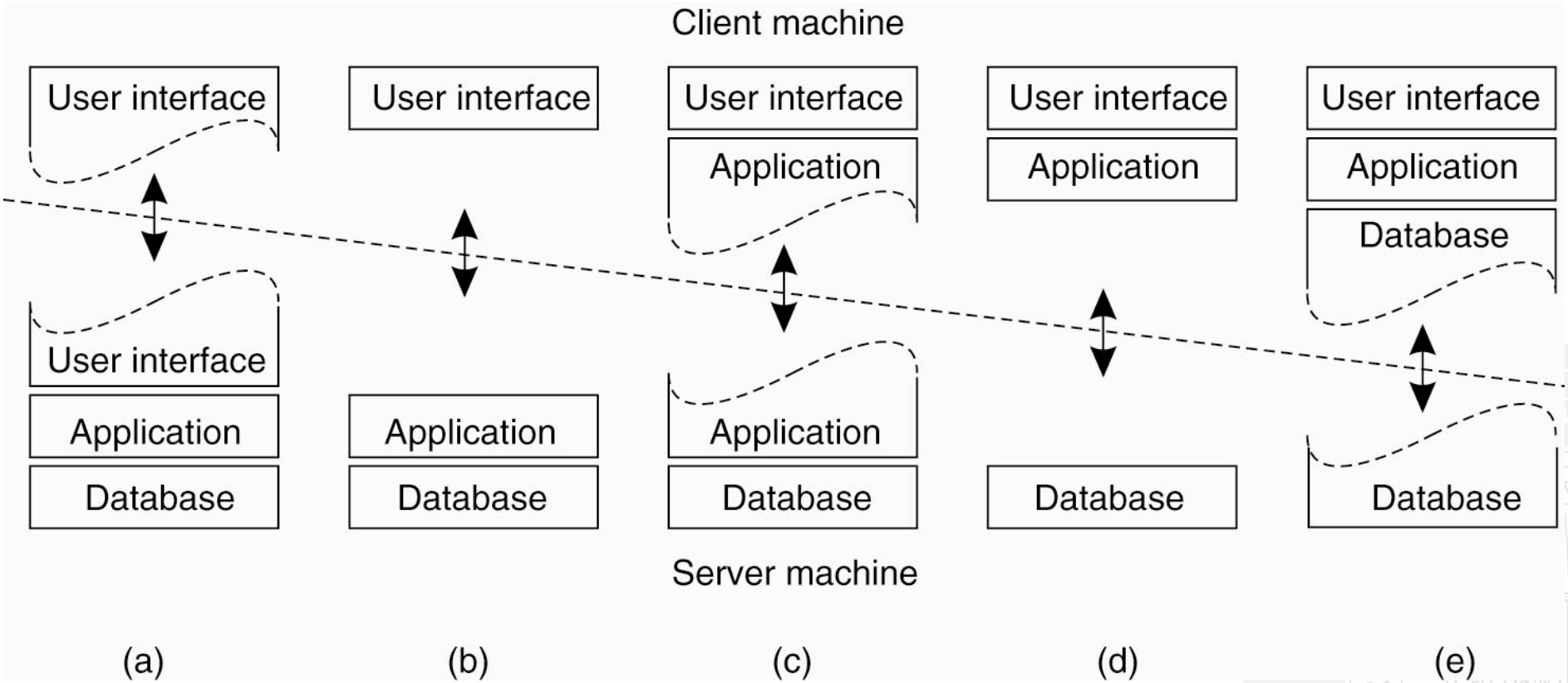
Two-tiered architecture

- the simplest choice is to have just two sort of machines
- working as the places hosting either servers or clients
- resulting in the (physically) *two-tiered architecture*

Possible choices for two-tiered architecture

- where are the three application-layers placed?
- on the client machines, or on the server machines?
- a range of possible solutions, accordingly

Possible Two-tiered Organisations



Alternative client-server organisations [Tanenbaum and van Steen, 2007]

Current Trends in Two-tiered Architectures

Moving toward the clients

- scalability pushes charge far from servers
- along with more efficient network connections, more powerful client machines, and above all more expressive technologies for distributing applications

Thin vs. fat clients

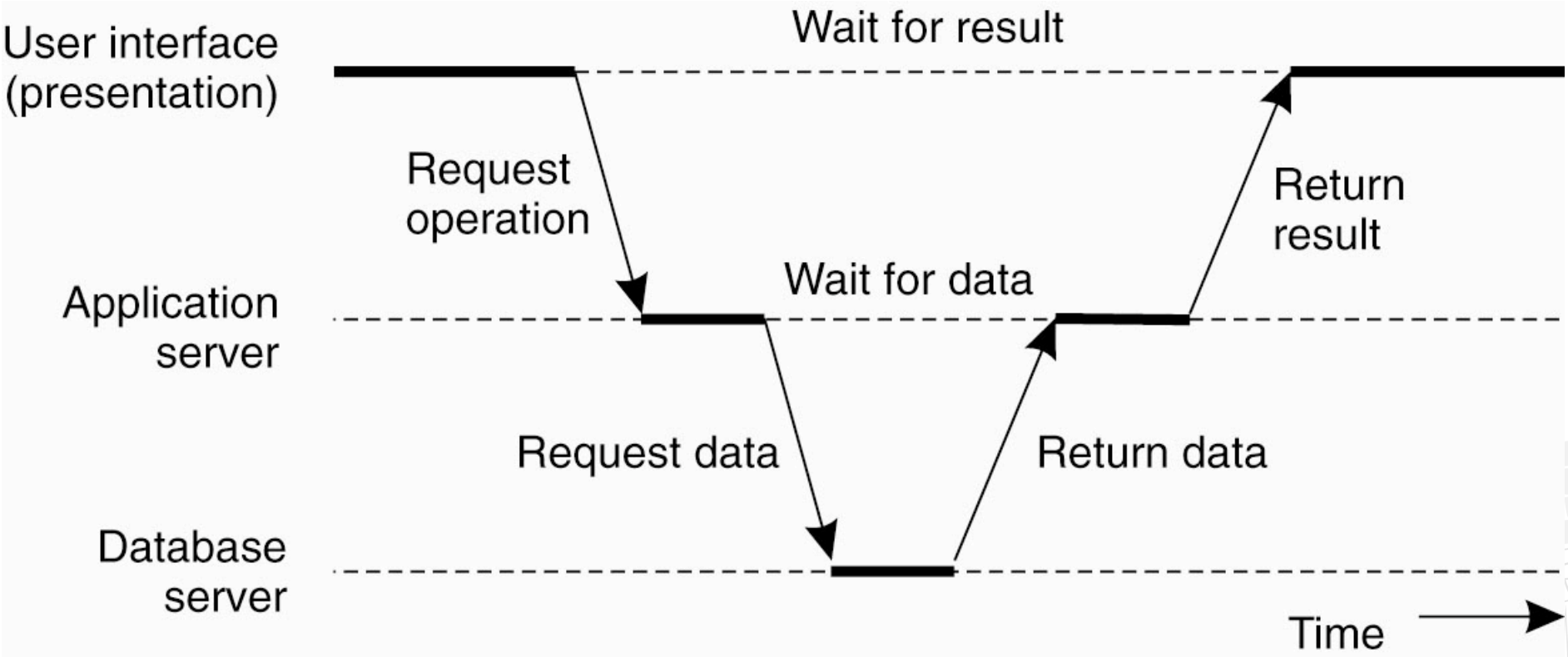
- *thin clients* are simpler
- *fat clients* are more complex, but are typically
  - more efficient from the user's viewpoint
  - more scalable from the engineer's viewpoint

Three-tiered Architectures I

Servers may sometimes act as clients

- servers might be layered, in turn
- we may (physically) distinguish between application servers and database servers
- e.g., the Transaction Processing Monitor discussed in the previous lessons

Three-tiered Architectures II



An example of a server acting as client [Tanenbaum and van Steen, 2007]

Vertical vs. Horizontal Distribution I

Vertical distribution

- multi-tiered client-server architectures directly derive from the three levels of applications
- logical organisation is directly mapped onto the tiers
- more generally, logical distribution has some straightforward relation with the physical distribution
- often, distributed processing amounts at building a client-server application according to a multi-tiered architecture
- this is typically called vertical distribution

Vertical vs. Horizontal Distribution II

Horizontal distribution

- sometimes, the physical distribution of the clients and the servers is what actually counts
- clients and servers may be physically split into logically-equivalent parts, each one working on its own portion of the whole data set
- this is typically called horizontal distribution
- this is an obviously *decentralised* class of systems

Horizontal Distribution: Main Example I

Peer-to-peer systems

- all the processes in a peer-to-peer system are equal
- so, every process works to the system main function, whatever it is
- each process works then at the same time as a client and as a server
- so, it is typically called servent

Overlay network

- peer-to-peer architectures are (basically) *symmetric*
- so, the main problem of peer-to-peer architectures is how to organise the network whose nodes are the servents and the links are the communications among them
- such a network organisation is typically called an overlay network

Horizontal Distribution: Main Example II

Types of overlay networks

- processes communicate through available communication channels
- overlay networks may be either structured or unstructured
- accordingly, the two main sorts of peer-to-peer architectures are
  - *structured* peer-to-peer architectures
  - *unstructured* peer-to-peer architectures

Combining the Benefits

Hybrid architectures

- many distributed systems require properties from both client-server and peer-to-peer architectures
- so, they put together features from both centralised and decentralised architectures
- these are typically called hybrid architectures

Collaborative Distributed Systems I

Main idea

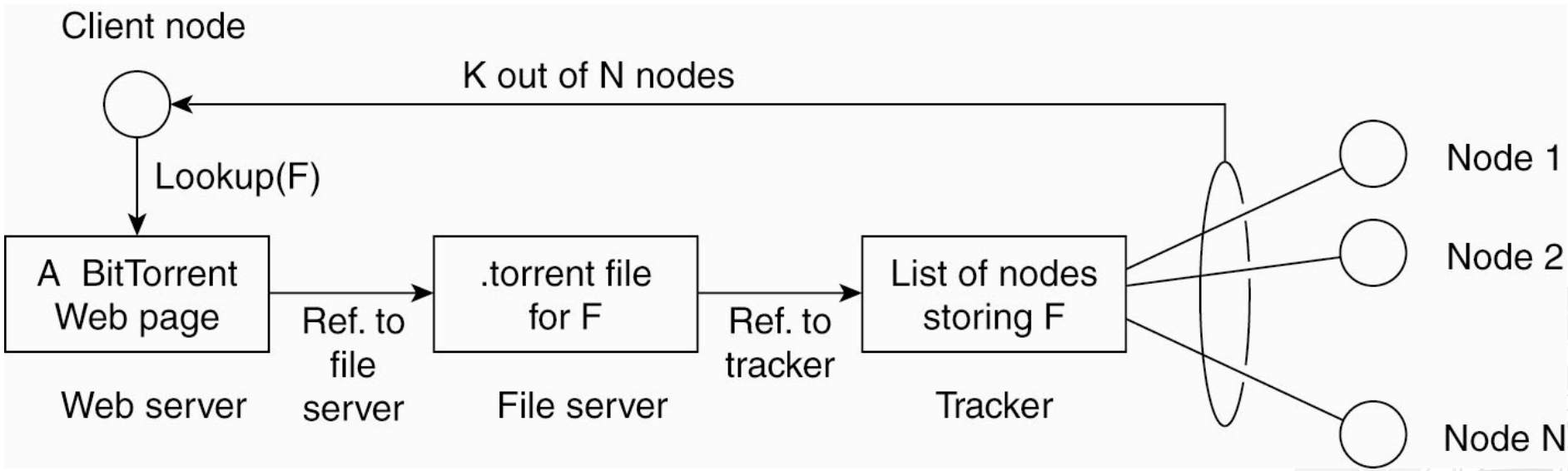
- the main problems of these systems is to get started: a traditional client-server scheme is then used here
- once a node has joined the system, collaboration proceeds using a fully decentralised scheme

Collaborative Distributed Systems II

Main example: BitTorrent

- BitTorrent is a peer-to-peer file downloading system
- when a user needs a file in BitTorrent, he/she gets chunks of the file from other users around until he/she gets it all
- a file can be downloaded by a client only when the client is providing files to other clients
- a global directory provides *.torrent* files that points to the *trackers*
- trackers are servers knowing *active*, collaborating nodes that can provide the requested chunks
- collaboration of nodes is promoted by suitable reward / punishment policies

BitTorrent as a Collaborative Distributed System



The principal working of BitTorrent [Tanenbaum and van Steen, 2007]

Next in Line...

WWW as a Network-based Application

The World Wide Web is a *network-based application* because

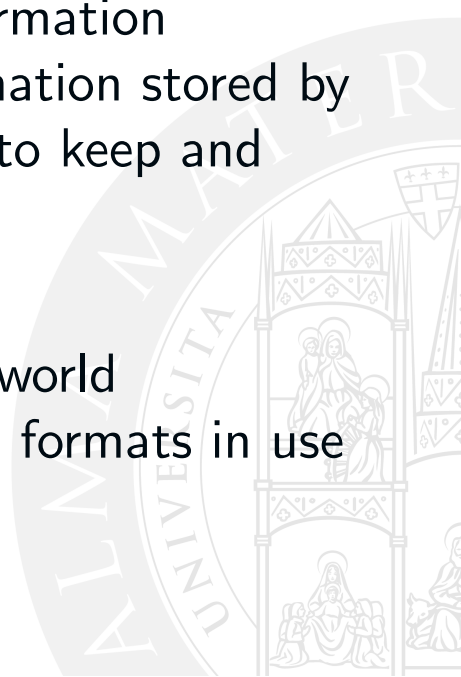
- communication between components is restricted to *message passing*, unlike more general applications
- operations across the network are performed in a fashion that is *not necessarily transparent* to the user, unlike classic distributed systems that look to their users like ordinary centralised systems
- applications represent "business aware" functionalities—unlike operating systems, networking software, and support systems
- in application architectures, the goals of a user action are representable as functional architectural properties, such as locating information, performing requests, and rendering data streams
  - this is in contrast with e.g. a networking abstraction, where the goal is to move bits from one location to the other without regard to why those bits are being moved

Architectural Properties for Network-based Applications

- performance
  - network performance
  - user-perceived performance
- scalability
- simplicity
- modifiability
  - evolvability
  - extensibility
  - customisability
  - configurability
  - reusability
- visibility
- portability
- reliability

The Application Domain of the World Wide Web

- the major goal of the World Wide Web was to be a "shared information space through which people and machines could communicate."
- such a goal brought about two basic needs
  - a way for people to store and structure their own information
  - a way to be able to reference and structure the information stored by others so that it would not be necessary for everyone to keep and maintain local copies
- more requirements came from
  - *distribution* of intended end-users located around the world
  - *heterogeneity* of machines, operating systems, and file formats in use



The Web as a Distributed Hypermedia System

The World Wide Web was intended as a *distributed hypermedia system*

Hypermedia

Hypermedia is defined by the presence of application control information embedded within, or as a layer above, the presentation of information

Distributed hypermedia

Distributed hypermedia allows the presentation and control information to be stored at remote locations

Simplicity

A low entry-barrier was necessary to enable sufficient adoption by readers, authors, and application developers

Readers

Hypermedia was chosen as the user interface because of

- simplicity and generality
- flexibility of relationships (links) allowing for unlimited structuring

Authors

Hypertext allowed partial availability of content and references without preventing their creation

Developers

Text-based protocols were the basis for simplifying application development

Extensibility

While simplicity makes it possible to deploy an initial implementation of a system, *extensibility* allows the system to evolve beyond the limitations of what was initially *deployed*

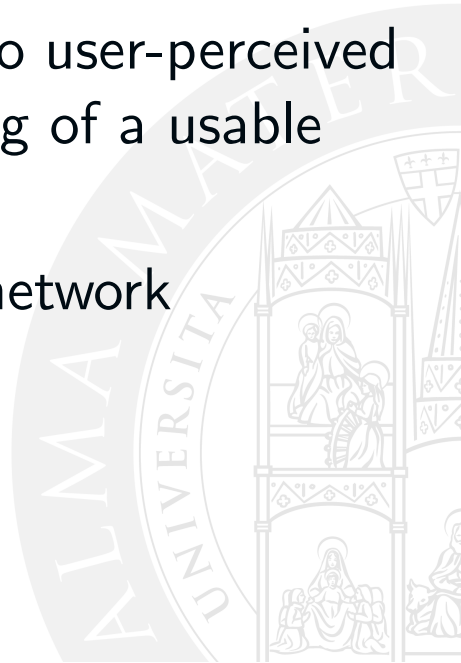
Latency

User actions within a distributed hypermedia system require the transfer of large amounts of data from where the data is stored to where it is used

- the World Wide Web architecture must be designed for large-grain data transfer

The usability of hypermedia interaction is highly sensitive to user-perceived latency: the time between selecting a link and the rendering of a usable result

- the World Wide Web architecture needs to minimise network interactions



Scalability

The Web is intended to be an Internet-scale distributed hypermedia system

- the entire system is not under the control of a single entity
- the system is about interconnecting information networks across multiple organisational boundaries
- all entities participating in the system may be acting towards different or crossing purposes

Scalability

Architectural elements need to continue operating when they are subjected to unanticipated load, or when given malformed or maliciously constructed data, since they may be communicating with elements outside their organisational control

- the architecture must feature mechanisms enhancing visibility and scalability

Security

Multiple organisational boundaries implies that multiple trust boundaries could be present in any communication

Security

This requires that the architecture be capable of communicating authentication data and authorisation controls. However

- authentication degrades scalability
- the architecture's default operation should be limited to actions that do not need trusted data

Independent Deployment

Multiple organisational boundaries also means that the system must be prepared for gradual and fragmented change

- old and new implementations co-exist
- old implementations must not prevent the new implementations from making use of their extended capabilities

Deployment

The architecture as a whole must be designed to ease the deployment of architectural elements in a partial, iterative fashion

- Existing architectural elements need to be designed with the expectation that architectural features will be added later
- Older implementations need to be easily identified so that legacy behaviour can be encapsulated without adversely impacting newer architectural elements

Deriving the Web Architectural Style

From the above requirements, an *architectural style* can be *derived* and used to define the principles behind the World Wide Web architecture. The formalisation process for the Web architectural style works under two hypothesis:

Hypothesis I

The design rationale behind the WWW architecture can be described by an architectural style consisting of the set of constraints applied to the elements within the Web architecture

Hypothesis II

Constraints can be added to the WWW architectural style to derive a new hybrid style that better reflects the desired properties of a modern Web architecture

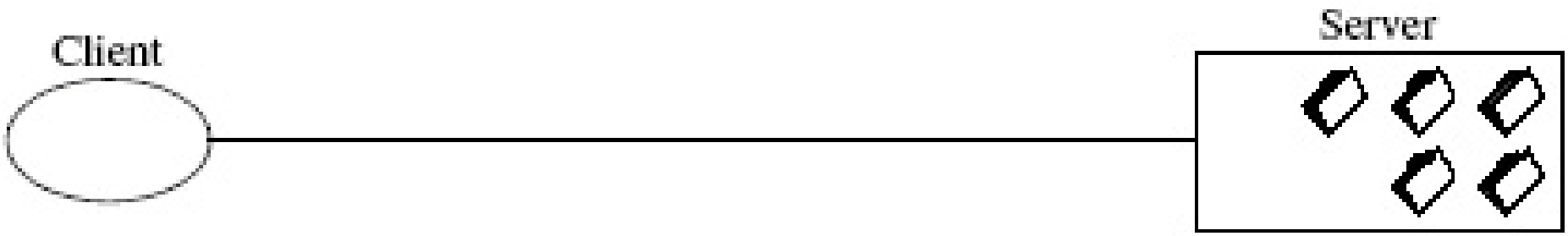
Deriving ReST as the Web Architectural Style

- the design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture
- by examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web constraints

Starting From the Null Style

The Null style starts with the system needs as a whole, without constraints, and then constraints are incrementally identified and applied to elements of the system in order to differentiate the design space and allow the forces that influence system behaviour to flow naturally, in harmony with the system

Client-Server I



Principle

Separation of concerns—between user interface and data storage

Client-Server II

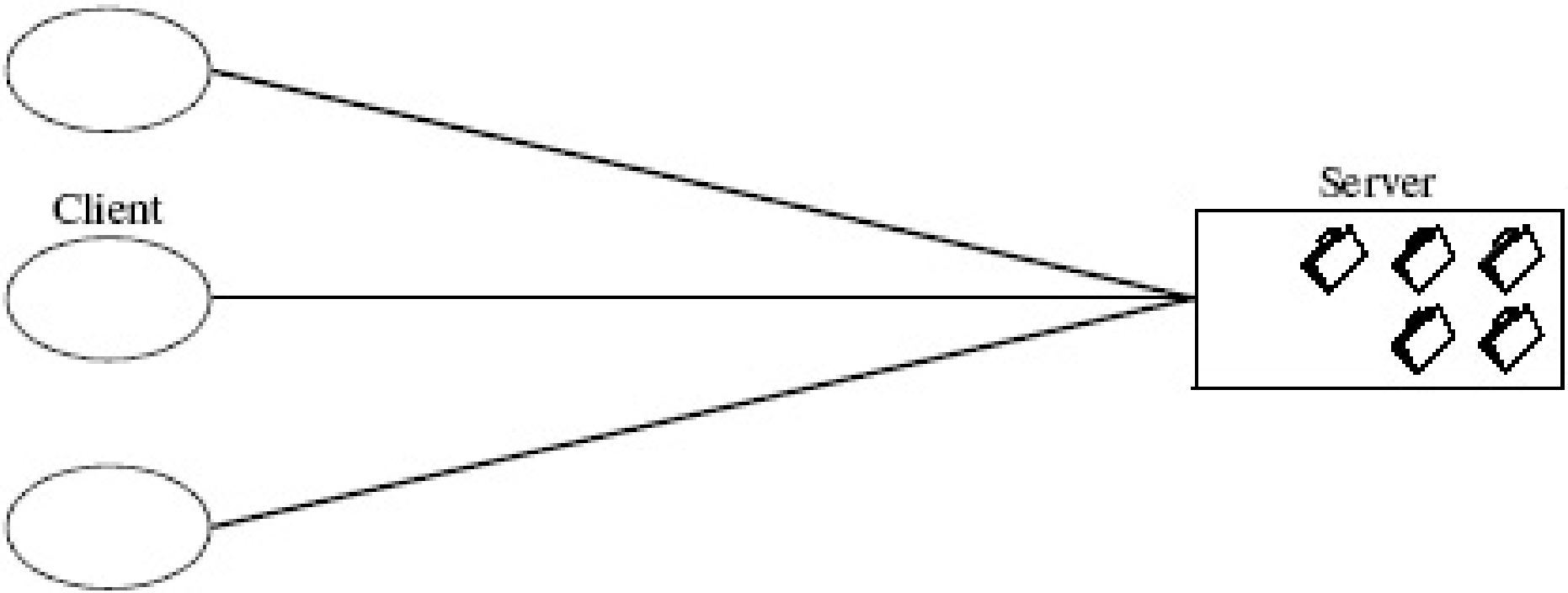
Constraints

- a server component, offering a set of services, listens for requests upon those services
- a client component, desiring that a service be performed, sends a request to the server via a connector
- the server either rejects or performs the request and sends a response back to the client.

Properties

- improves *portability* of the user interface across multiple platforms
- improves *scalability* by simplifying the server components
- allows components to *evolve* independently

Stateless I



Stateless II

Constraint

Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server

Properties

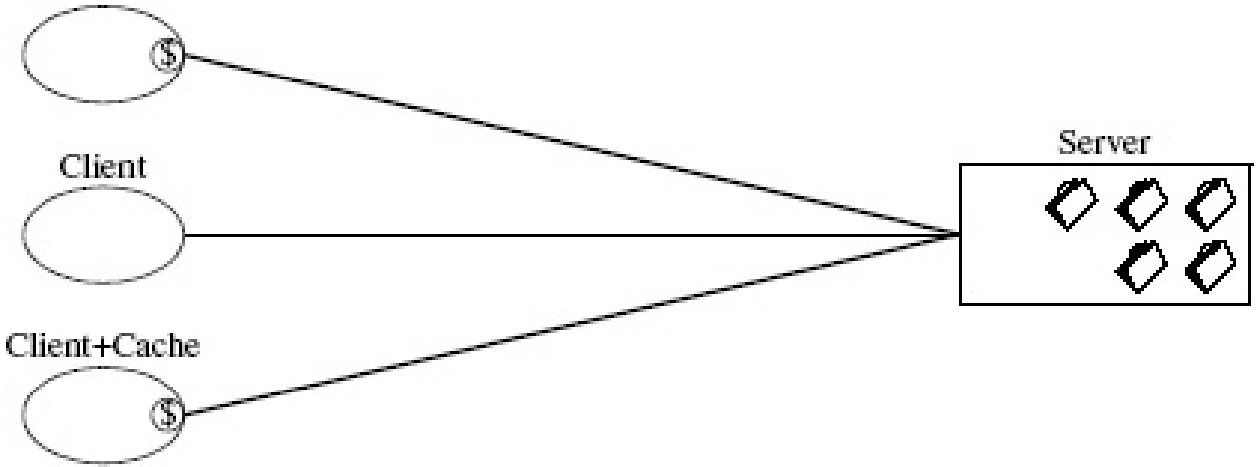
- improves *visibility* by allowing monitoring systems to determine the full nature of a request without looking beyond it
- improves *reliability* by easing the recovering from partial failures
- improves *scalability* by allowing server components to quickly free resources
- improves *simplicity* by simplifying implementation because the server does not have to manage resource usage across requests

Stateless III

Drawbacks

- decreases network performance by increasing the per-interaction overhead repetitive data sent in a series of requests
- reduces the server control over consistent application behaviour

Cache I



Constraint

Data within a response to a request be implicitly or explicitly labeled as cacheable or not. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

Cache II

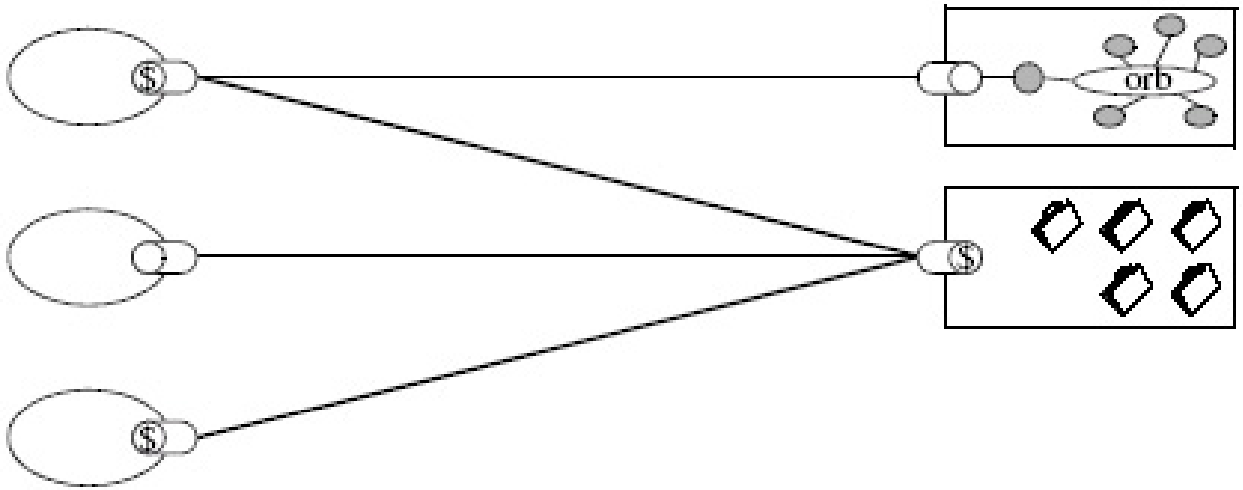
Properties

- improves *efficiency*, *scalability*, and *user-perceived performance* by reducing the average latency of a series of interactions

Drawbacks

- decreases *reliability* if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server

Uniform Interface I



The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a *uniform interface* between components

Principle

Generality (applied to the component interface)

Uniform Interface II

Constraint

Multiple architectural constraints are needed to guide the behaviour of components:

- identification of resources
- manipulation of resources through representations
- self-descriptive messages
- hypermedia as the engine of application state

Properties

- *simplifies* the overall system architecture
- improves the *visibility* of interactions
- encourages independent *evolvability* by uncoupling implementations from the services they provide

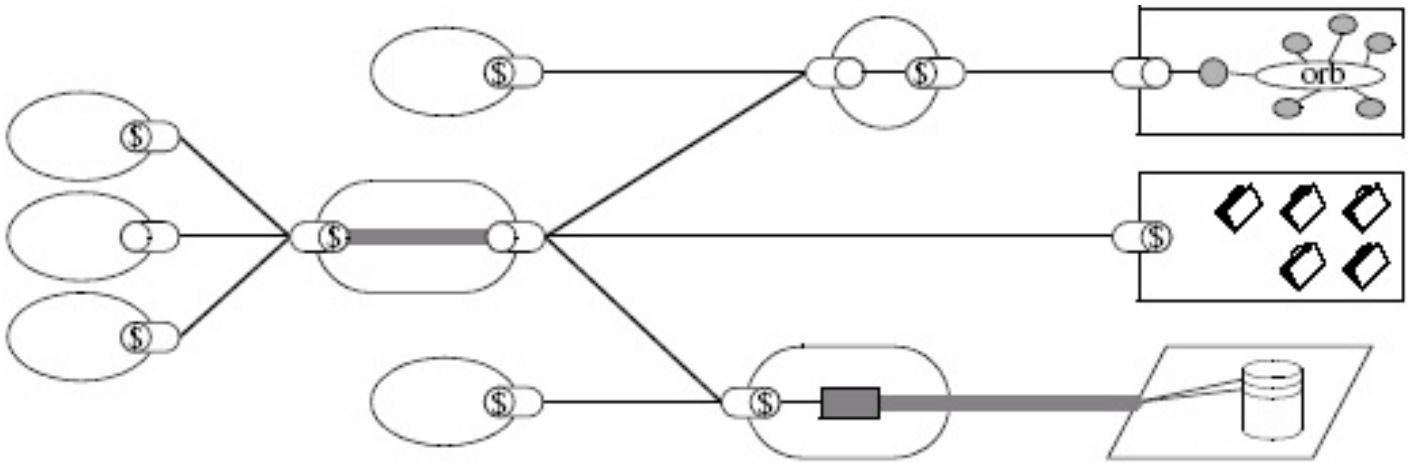
Uniform Interface III

The ReST interface is designed to be efficient for large-grain hypermedia data transfer, optimising for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction

Drawbacks

- degrades *efficiency* by transferring information in a standardised form rather than one which is specific to an application's needs

Layered System I



Constraint

Compose an architecture of hierarchical layers by constraining component behaviour such that each component cannot see beyond the immediate layer with which they are interacting

Layered System II

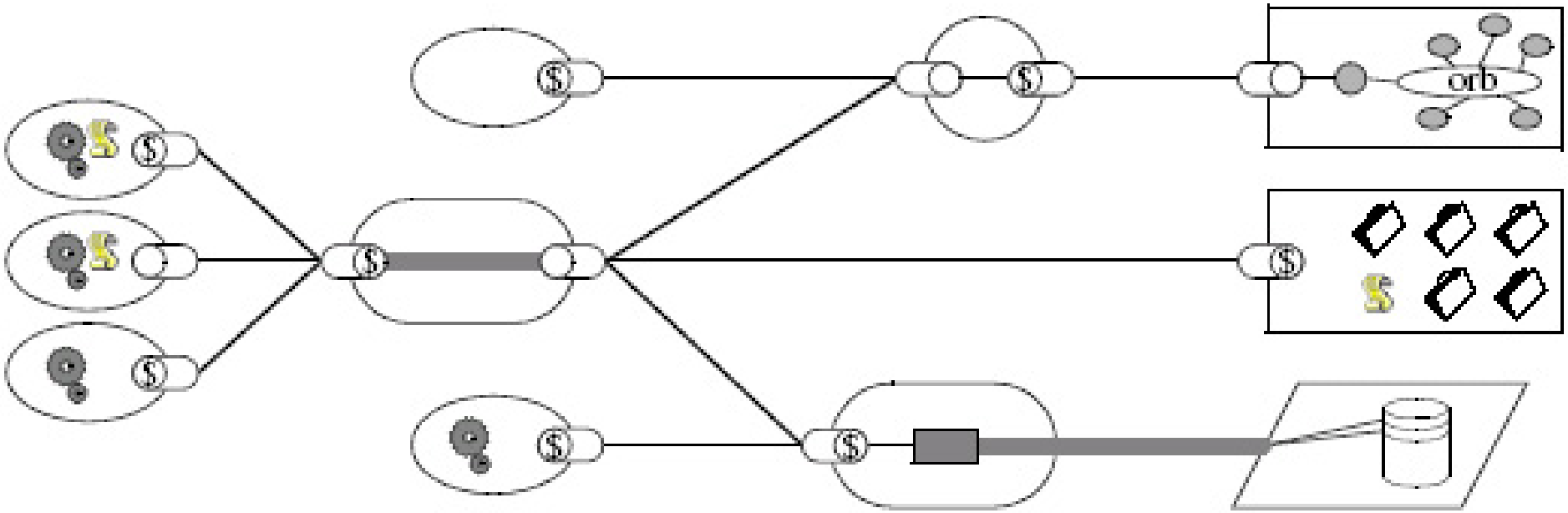
Properties

- improves the overall system *simplicity* and promote *independence* by restricting knowledge of the system to a single layer
- improve system *scalability* by enabling load balancing of services at intermediaries
- improves *security* by allowing policies to be enforced on data crossing organisational boundaries

Drawbacks

- reduces *user-perceived performance* by adding overhead and latency to data processing
  - the effect is countered by shared caching at intermediaries

Code-On-Demand I



Code-On-Demand II

It is an *optional* constraint, so that the architecture only gains the benefit and suffer the drawbacks when they are known to be in effect for some realm of the overall system

Constraint

Client functionalities can be extended by downloading and executing code—typically in the form of applets and scripts

Properties

- *simplifies* clients by reducing the number of features required to be pre-implemented
- improves system *extensibility* by allowing features to be downloaded after deployment

Code-On-Demand III

Drawbacks

- reduces *visibility* and thus is only an optional constraint

ReST General Overview I

The Representational State Transfer (ReST) is an abstraction of the architectural elements within a distributed hypermedia system. ReST ignores the details of component implementation and protocol syntax in order to focus on

- the roles of components
- the constraints upon interaction between components
- the component interpretation of significant data elements

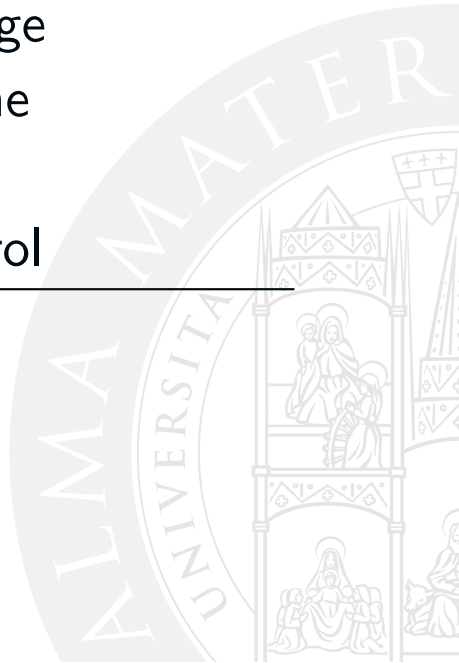
ReST General Overview II

Core of the ReST architecture

ReST components communicate by transferring a representation of a resource in a format matching one of an *evolving set* of *standard* data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the resource. Whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the component uniform interface.

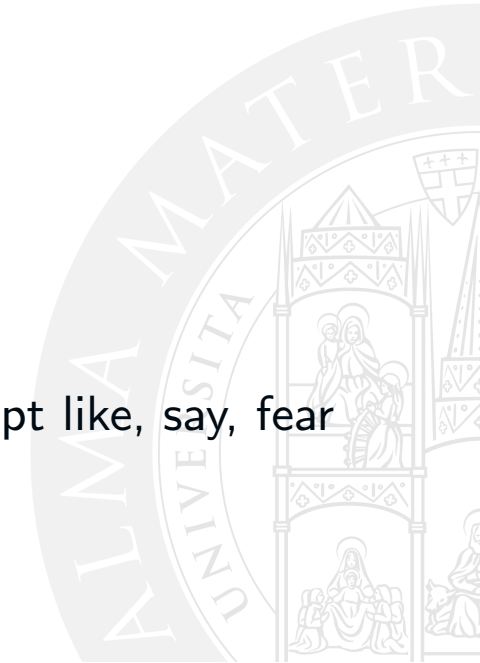
ReST Data Elements

Data Element	Modern Web Examples
resource	the conceptual target of a hypertext link
resource identifier	URI (URL, URN)
representation	HTML document, JPEG image
representation metadata	media type, last-modified time
resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control



Resources

- the key abstraction of information in ReST is a resource
- any information that can be named and is important enough to be referenced as a thing in itself can be a resource
  - a document
  - an image
  - a temporal service
    - e.g. today's weather in any Italian city
  - a collection of other resources
    - e.g. a list of open bugs in a bug database
  - a non-virtual object
    - e.g. a physical object like a lamp, an abstract concept like, say, fear



Resources as Conceptual Mappings

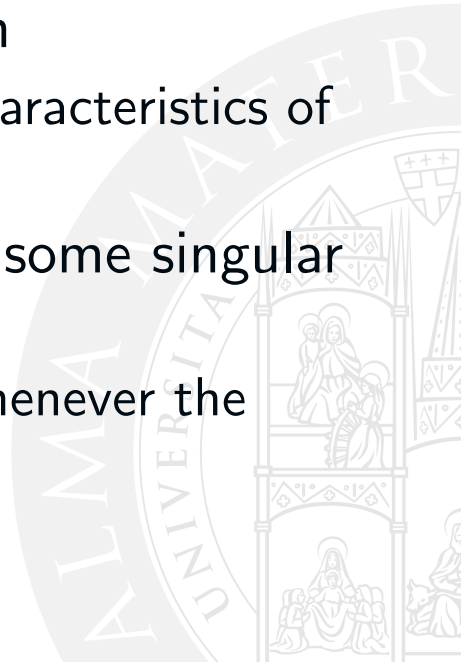
- a resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time
- the entities in the set can be
  - resource *representations*
  - resource *identifiers*
- a resource can map to the empty set
  - references can be made to a concept before any realisation of that concept exists

  resources can be
  - *static*, in the sense that, when examined at any time after their creation, they always correspond to the same entity set
  - *dynamic*, otherwise
- the only thing that is required to be *static* for a resource is the *semantics* of the mapping, since the semantics is what distinguishes one resource from another

Resources and the Web Architecture

The abstract definition of resources enables key features of the Web architecture

- provides *generality* by encompassing many sources of information
  - avoids artificially distinguishing information sources by type or implementation
- allows *late binding* of the reference to a representation
  - enables content negotiation to take place based on characteristics of the request
- allows an author to reference the concept rather than some singular representation of that concept
  - thus removing the need to change all existing links whenever the representation changes



Resource Identifiers

- each resource has to have at least one *identifier* in the form of a URI (RFC 2396, [Berners-Lee et al., 1998])
- the URI is the *name* and *address* of a resource

> *The URI is the fundamental technology of the Web. There were hypertext systems before HTML, and Internet protocols before HTTP, but they didn't talk to each other. The URI interconnected all these Internet protocols into a Web.*
> *The web kills off other protocols because it has something most protocols lack: a simple way of labeling every available item. Every resource on the Web has at least one URI.*

Design guidelines

- URI should have a structure
- their structure should vary in predictable ways

The Relationship Between URIs and Resources

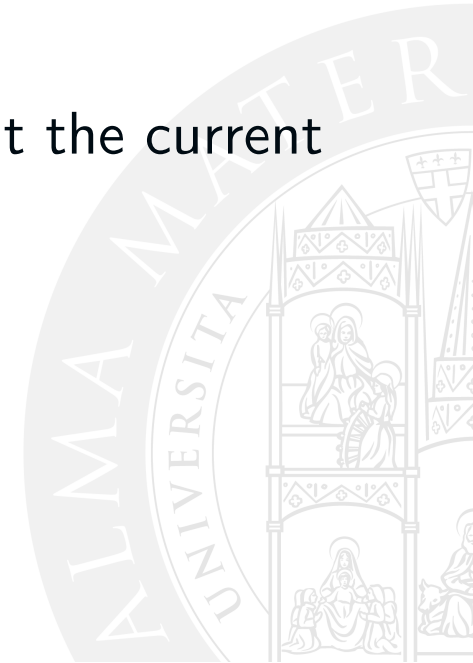
- no two resources can be the same, since each resource maps a different concept
- however, at some moment in time, two different resources may point to the same data, e.g.
  - `http://example.com/software/release/1.0.3/`
  - `http://example.com/software/release/latest/`
- a resource may have one URI or many
- every URI designates exactly one resource

Design guidelines

- a resource and its URI ought to have an intuitive correspondence

Representations

- a representation is a sequence of bytes, plus *representation metadata* to describe those bytes
  - other commonly used but less precise names for a representation include: document, file, HTTP message
- a representation contains *any useful information* about the current state of a resource



Representations Metadata

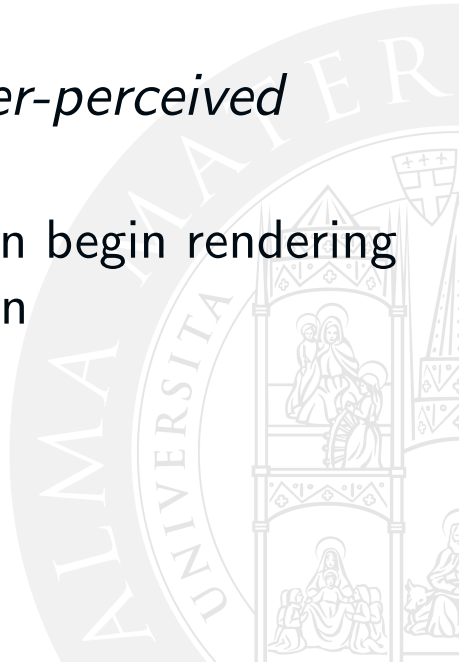
- representation metadata is in the form of name-value pairs, where the name corresponds to a standard that defines the structure and semantics of the value
- response messages may include both
  - representation metadata, and
  - *resource metadata*, i.e. information about the resource that is not specific to the supplied representation

Control data

- *control data* defines the purpose of a message between components
  - e.g. the action being requested or the meaning of a response
- control data is also used to parameterise requests and override the default behaviour of some connecting elements
  - e.g. cache behaviour

Media Types

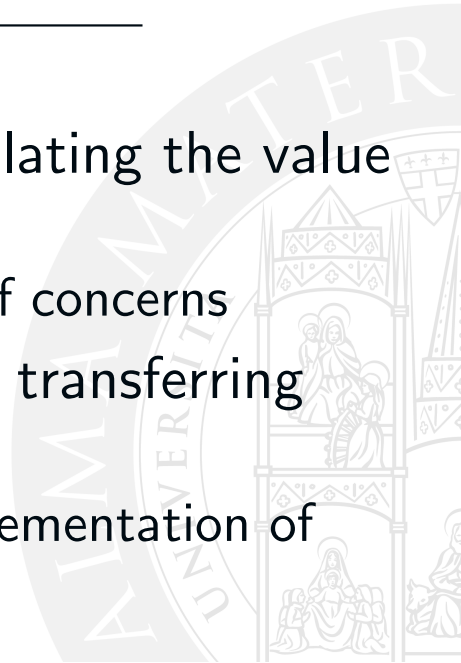
- the data format of a representation is known as a media type
- representation can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type
- the design of a media type can directly impact the *user-perceived performance* of a distributed hypermedia system
  - any data that must be received before the recipient can begin rendering the representation adds to the latency of an interaction



Connectors

Connector	Modern Web Examples
client	libwww, libwww-perl
server	libwww, Apache API, NSAPI
resolver	bind (DNS lookup library)
tunnel	SOCKS, SSL after HTTP CONNECT
cache	browser cache

- providing a generic interface for accessing and manipulating the value set of a resource
  - enhancing *simplicity* by providing a clean separation of concerns
- encapsulating the activities of accessing resources and transferring resource representations
  - enabling *substitutability* by hiding the underlying implementation of resources and communication mechanisms



Connector Types I

Client and Server

The primary connector types are *client* and *server*

- a client initiates communication by making a request
- a server listens for connections and responds to requests in order to supply access to its services

Resolver

A *resolver* translates partial or complete resource identifiers into the network address information needed to establish an inter-component connection

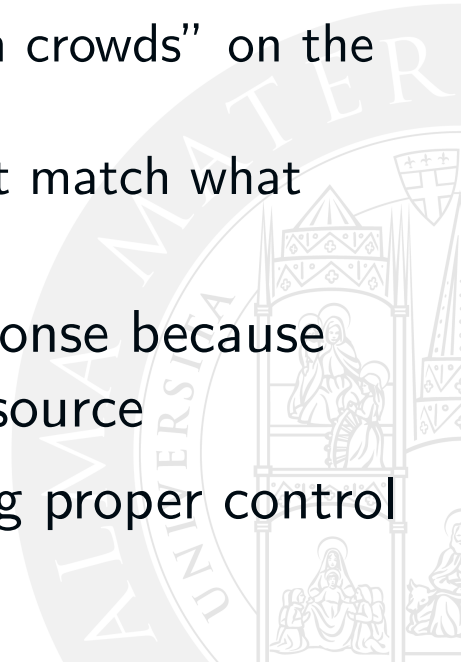
Connector Types II

Tunnel

A *tunnel* simply relays communication across a connection boundary, such as a firewall or lower-level network gateway. Some ReST components (e.g. proxy) may dynamically switch from active component behavior to that of a tunnel.

Cache I

- the cache connector is located on the interface to a client or server connector in order to save cacheable responses to current interactions so that they can be reused for later requested interactions
- some cache connectors are shared, meaning that cached responses may be used in answer to a client other than the one for which the response was originally obtained
  - can be an effective way to reduce the impact of "flash crowds" on the load of a popular server
  - can also lead to errors if the cached response does not match what would have been obtained by a new request
- a cache is able to determine the *cacheability* of a response because the interface is generic rather than specific to each resource
- default cache behaviour can be overridden by including proper control data in the interaction



Components I

Component	Modern Web Examples
origin server	Apache httpd, Microsoft IIS
gateway	Squid
proxy	CERN Proxy, Netscape Proxy
user agent	Mozilla Firefox, Safari

Origin Server

Uses a server connector to govern the namespace for a requested resource

- the server is the definitive source for representations of its resources and must be the ultimate recipient of any request that intends to modify the value of its resources
- provides a generic interface to its services as a resource hierarchy

Components II

User Agent

Uses a client connector to initiate a request and becomes the ultimate recipient of the response

Proxy and Gateway

Intermediary components act as both a client and a server in order to forward, with possible translation, requests and responses

- a client determines when it will use a proxy
- a gateway is imposed by the network or origin server

Summing Up

Organisation of distributed systems

- architectural styles deal with software organisation
- they are approximative and maybe non-scientific ways to model systems
- however they are expressive and abstract enough to help distributed system engineering

Main issues

- software architectures are concerned with logical organisation
- system architectures are concerned with component placement in a distributed setting
- the ReST case study

References

Berners-Lee, T., Fielding, R., and Masinter, L. (1998).
Uniform Resource Identifiers (URI): Generic syntax.
http://www.ietf.org/rfc/rfc2396.txt.
Internet RFC 2396.

Fielding, R. T. (2000).
Architectural Styles and the Design of Network-based Software Architectures.
PhD thesis, University of California, Irvine, CA, USA.

Tanenbaum, A. S. and van Steen, M. (2007).
Distributed Systems. Principles and Paradigms.
Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.

Software Architectures

Distributed Systems

Sistemi Distribuiti

Andrea Omicini
andrea.omicini@unibo.it

Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Alma Mater Studiorum – Università di Bologna a Cesena

Academic Year 2017/2018