Introduction to Distributed System

Javier Espinosa, PhD
javier.espinosa@imag.fr

Outline

- **Distributed Systems**
  - Definition and examples
  - Challenges
  - Case study: the World Wide Web
- **System Architectures**
  - Elements
  - Layers model
  - N-Tier model
  - Case study: the Web Browser
- **Client-Server Model**
  - Characteristics
  - Implementation
  - Example: RMI

Distributed System

- Collection of (heterogeneous) networked computers which communicate and coordinate their actions by passing messages
- Characteristics
  - Appear to the user as a single computer
  - Concurrency (*with or without share memory*)
  - No global clock
  - Independent failures

    A component may fail (crash) independently, leaving the others still running

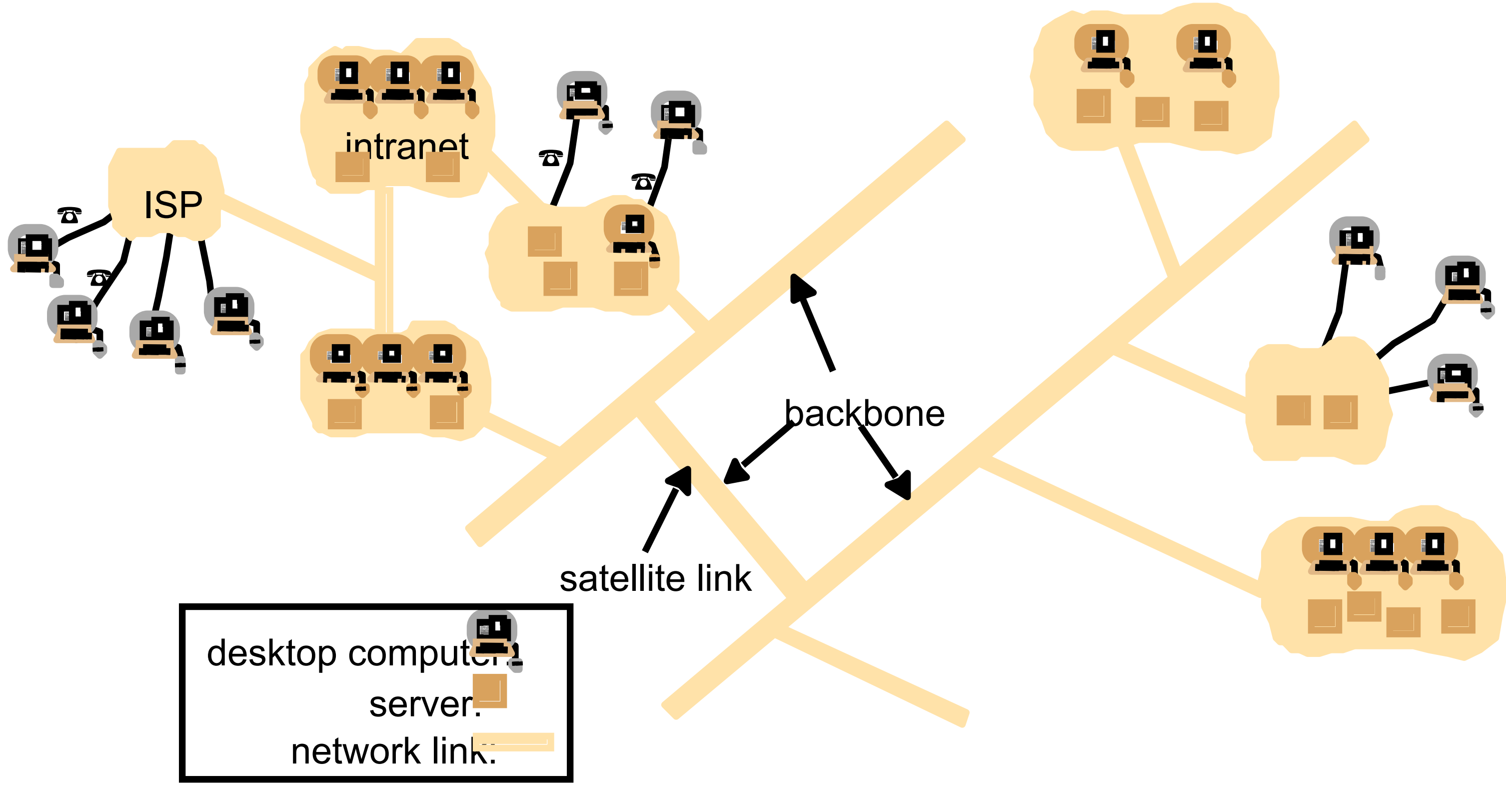
Why Distributed?

Economics	Microprocessors offer a better price/ performance than mainframes
Speed	A distributed system may have more total computing power than a mainframe
Inherent distribution	Some applications involve spatially separated machines
Reliability	If one machine crashes, the system as a whole can still survive
Incremental growth	Computing power can be added in small increments

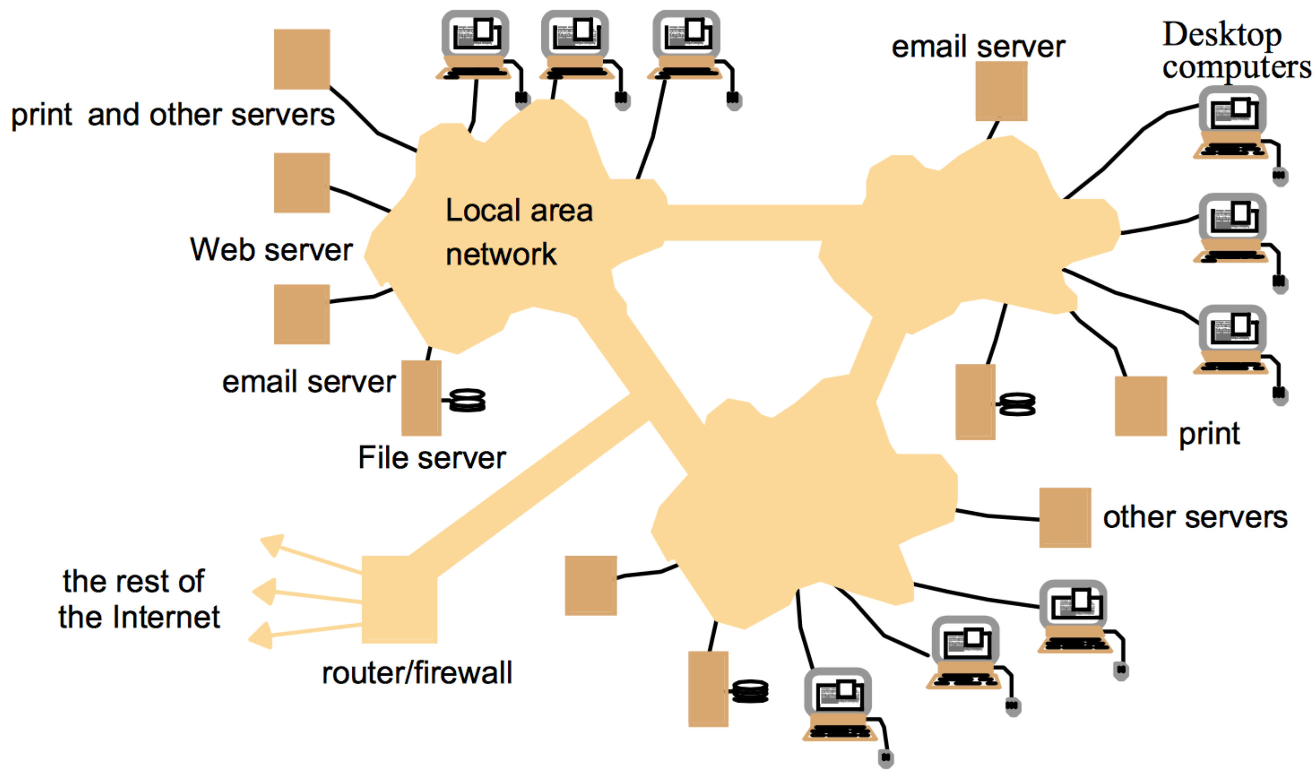
Why Learning Distributed Systems?

	Principle	Know-how
Everyone	Funcionamiento general (sistemas, aplicaciones distribuidas)	Utilización de sistemas, construcción elemental de aplicaciones distribuidas
	Principios y técnicas de base de los sistemas. Programación concurrente Principios de programación distribuida	Práctica de la programación concurrente, Utilización elemental de herramientas (RMI, CORBA)
Technicians	(Además) Profundizar sobre los sistemas (arquitectura interna) Introducción a la Algorítmica distribuida. Modelado, Evaluación de desempeño, seguridad	Programación avanzada de sistemas (comunicación, etc.) Administración de sistemas
Engeenierees	Arquitectura interna de herramientas de construcción desde objetos hasta componentes. Tolerancia a fallas, QoS, Seguridad	Práctica avanzada de las herramientas (J2EE, .NET, MOM, WEB Services) Construcción, configuración, adaptación de herramientas y de sistemas para dominios Especializados (móvil, embarcados)
Scientists	Algorítmica paralela y distribuida, aspectos fundamentales de la tolerancia a fallas y la seguridad. Arquitectura de sistemas. Servidores de alto desempeño	Según el proyecto de investigación

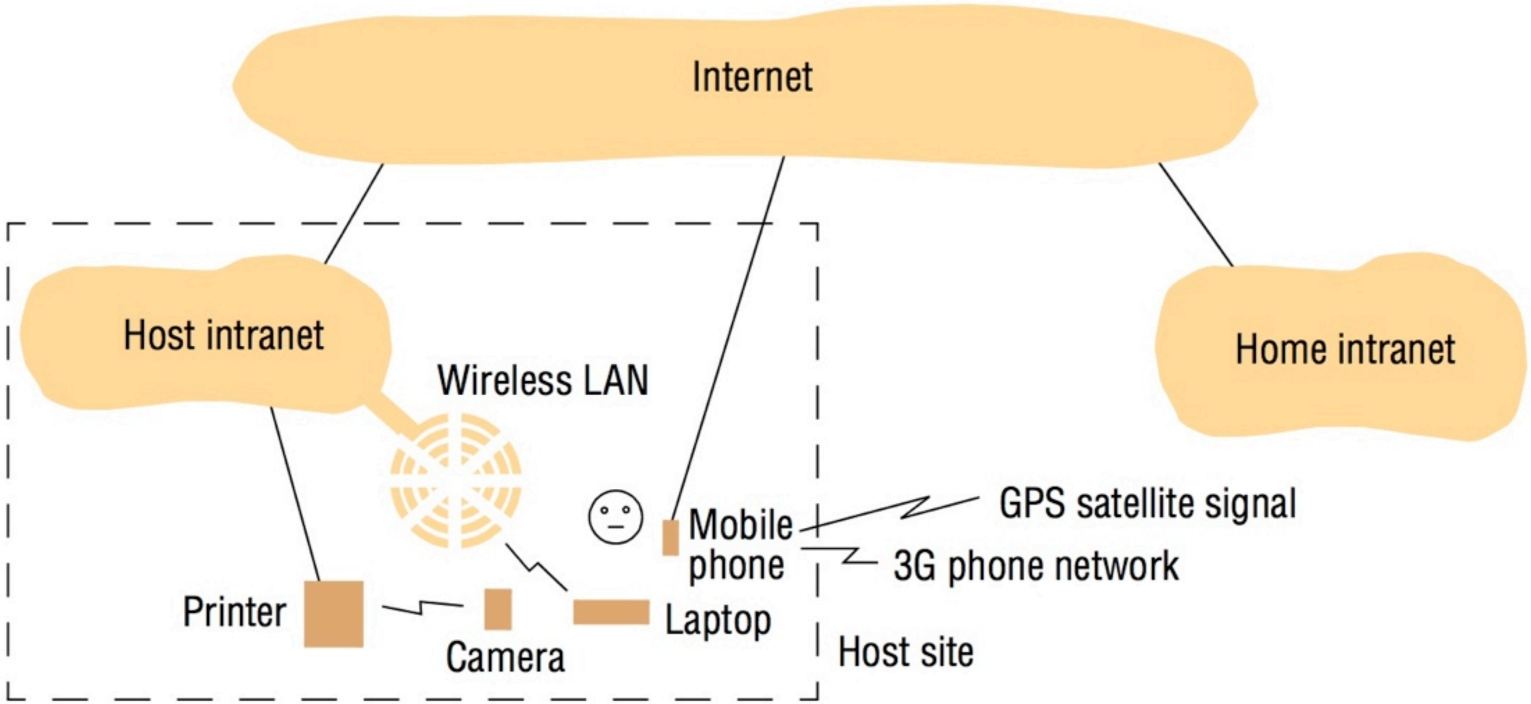
Example: Internet (portion of it)



Example: Intranet



Example: Mobile Environment

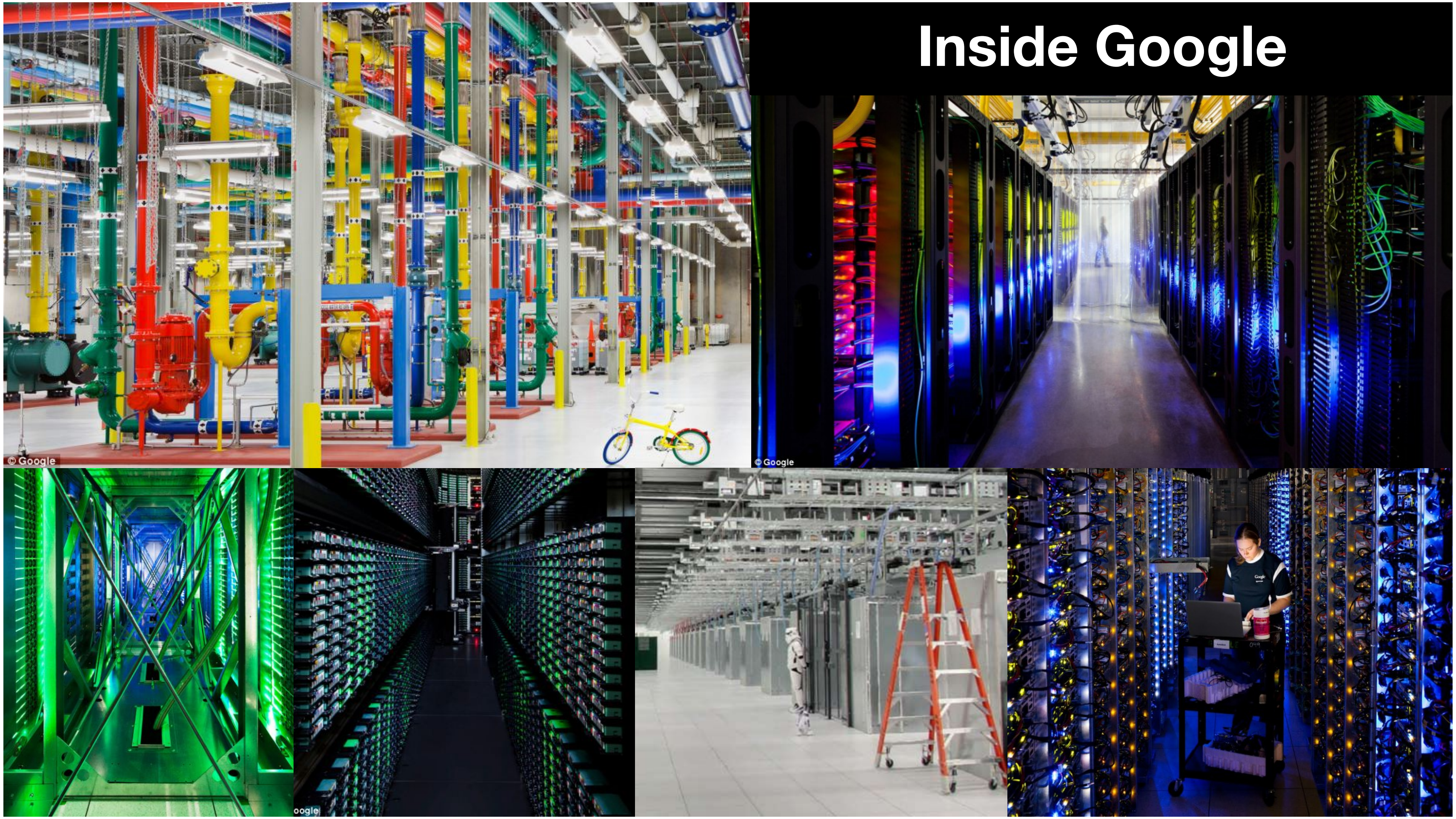


Distributed System Examples (i)

Google (web search engine)

- Index the entire content of the Web
  - *ex. web pages, multimedia documents, books*
- Largest and most complex distributed system in history:
  - Physical infrastructure consisting of very large numbers of networked computers located at data centers all around the world
  - Distributed storage system designed to support very large files and heavily optimized for the style of usage required by search (*fast readings*)
  - Computation infrastructure for managing very large parallel and distributed computations across the underlying physical infrastructure

Inside Google



Distributed System Examples (ii)

Massively Multiplayer Online Games

- Support large numbers of players simultaneously
- Enable players to *cooperate*, *compete* and *interact* in a persistent virtual world
- Major challenge (!!) in the the development of distributed systems:
  - Requires fast response times for preserving *user experience* of the game
  - Requires real-time propagation of events for maintaining a consistent view of the shared world

Distributed System Examples (ii)

Massively Multiplayer Online Games

- **Centralized** state management
  - A single copy of the state of the world is maintained on a centralized server
  - Clients access the state via players' consoles or other devices
  - The server consists of a *cluster* of computer nodes

    "Highly loaded 'star systems' have their own node while the others shared a node". Events are directed to the right node while keeping track of movement of players among start systems."

- **Distributed** state management
  - World state is partitioned across a number of servers
  - Users are dynamically allocated to a particular server based on *current usage patterns (ex. based on geographical proximity or network delays)*

Distributed System Challenges

Challenge	Description
Heterogeneity	A distributed system must be constructed from a variety of different networks, operating systems, computer hardware and programming languages.
Openness	A distributed systems should be extensible.
Security	Sensitive information is keep secret when transmitted in messages over a network.
Scalability	A distributed system is scalable when the cost of adding a user is constant in terms of the resources that must be added.
Failure handling	A distributed system needs to be aware of the possible ways in which its components may fail and be designed to deal with each of those failures.
Concurrency	The presence of multiple users is a source of concurrent requests to a resource. Each resource must be designed to be safe in a concurrent environment.
Transparency	Certain aspects of distribution are invisible to the application programmer
Quality of Service	Properties provided at runtime (performance, security, reliability).

Distributed System Challenges

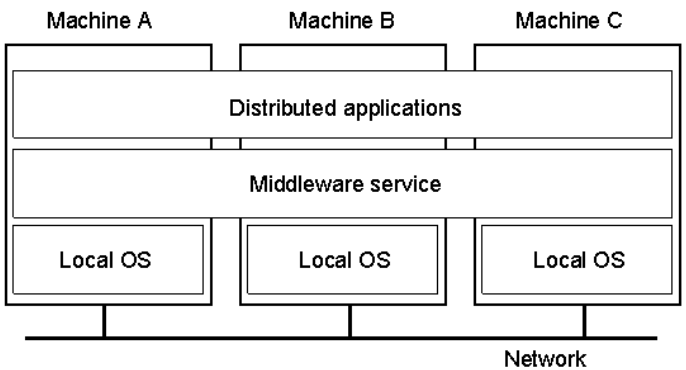
Heterogeneity

- Networks (*ex. ethernet, token ring*)
  - Masked by the fact that all computers in the network know the protocol
- Hardware (ex. little/big endian)
  - Must be dealt with before exchanging messages between programs running on different hardware
- Operative Systems communication APIs (ex. unix vs windows)
- Programming Languages data types (ex. arrays vs hash tables)

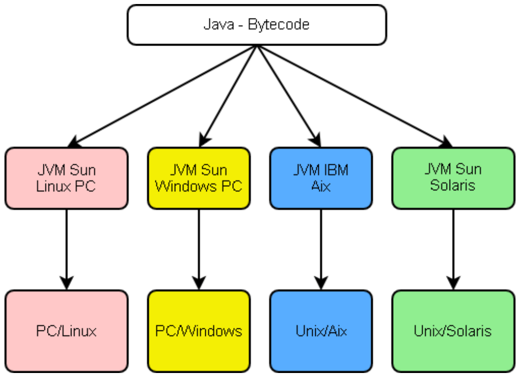
Distributed System Challenges

Heterogeneity

- Multiple Implementations
  - Due to the number of programmers and technologies
  - Requires the use of abstractions (*network, hardware, OS, programming languages*)



Middleware



Virtual Machine

Distributed System Challenges

Failure handling

- Partial failures
  - Can non-failed components continue operation?
  - Can the failed components easily recover?
- Failure detection and failure masking
- Recovery
- An import technique is replication
  - Why does space shuttles has 3 on-board computers?

Distributed System Challenges

Transparency

- For a user a distributed system appears as a single integrated facility

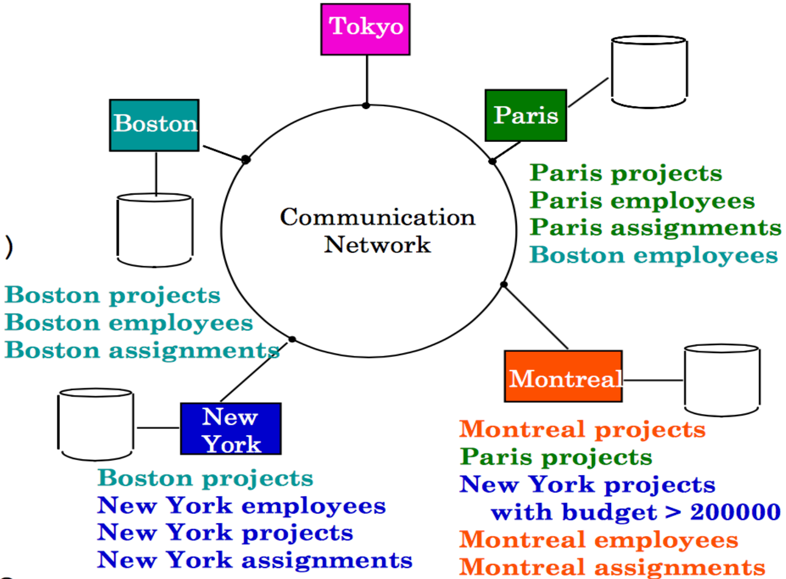
Transparency	**Description**
Access	Hide differences in data representation and how a resource is accessed
Location	Hide where a resource is located
Migration	Hide that a resource may move to another location
Relocation	Hide that a resource may be moved to another location while in use
Replication	Hide that a resource may be shared by several competitive users
Concurrency	Hide that a resource may be shared by several competitive users
Failure	Hide the failure and recovery of a resource
Persistence	Hide whether a (software) resource is in memory or on disk

Distributed System Challenges

Transparency Example

- Transparency
  - Not easy to maintain

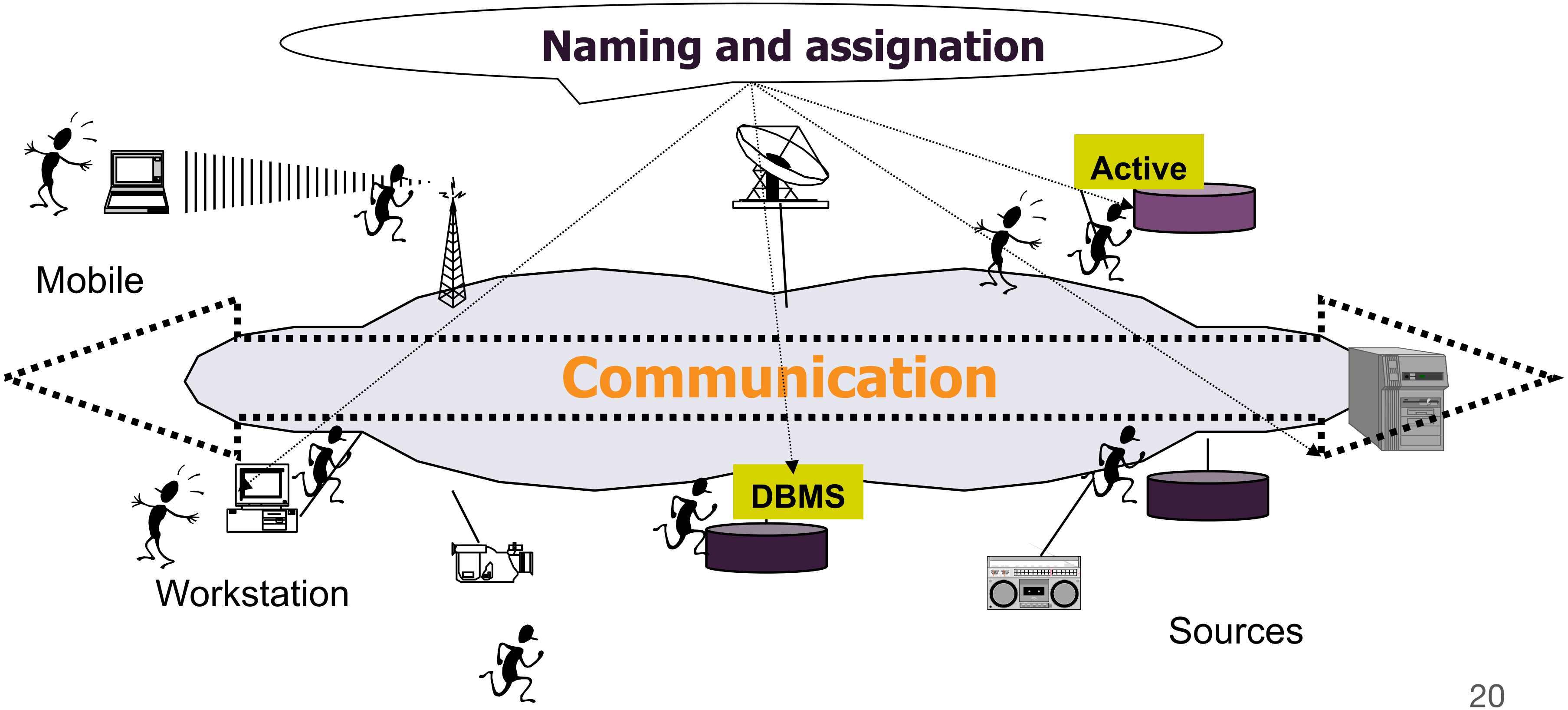
```
EMP(ENO,ENAME,TITLE,LOC)
PROJECT(PNO,PNAME,LOC)
PAY(TITLE,SAL)
ASG(ENO,PNO,DUR)

SELECT ENAME,SAL
FROM   EMP,ASG,PAY
WHERE  DUR > 12
AND    EMP.ENO = ASG.ENO
AND    PAY.TITLE = EMP.TITLE
```



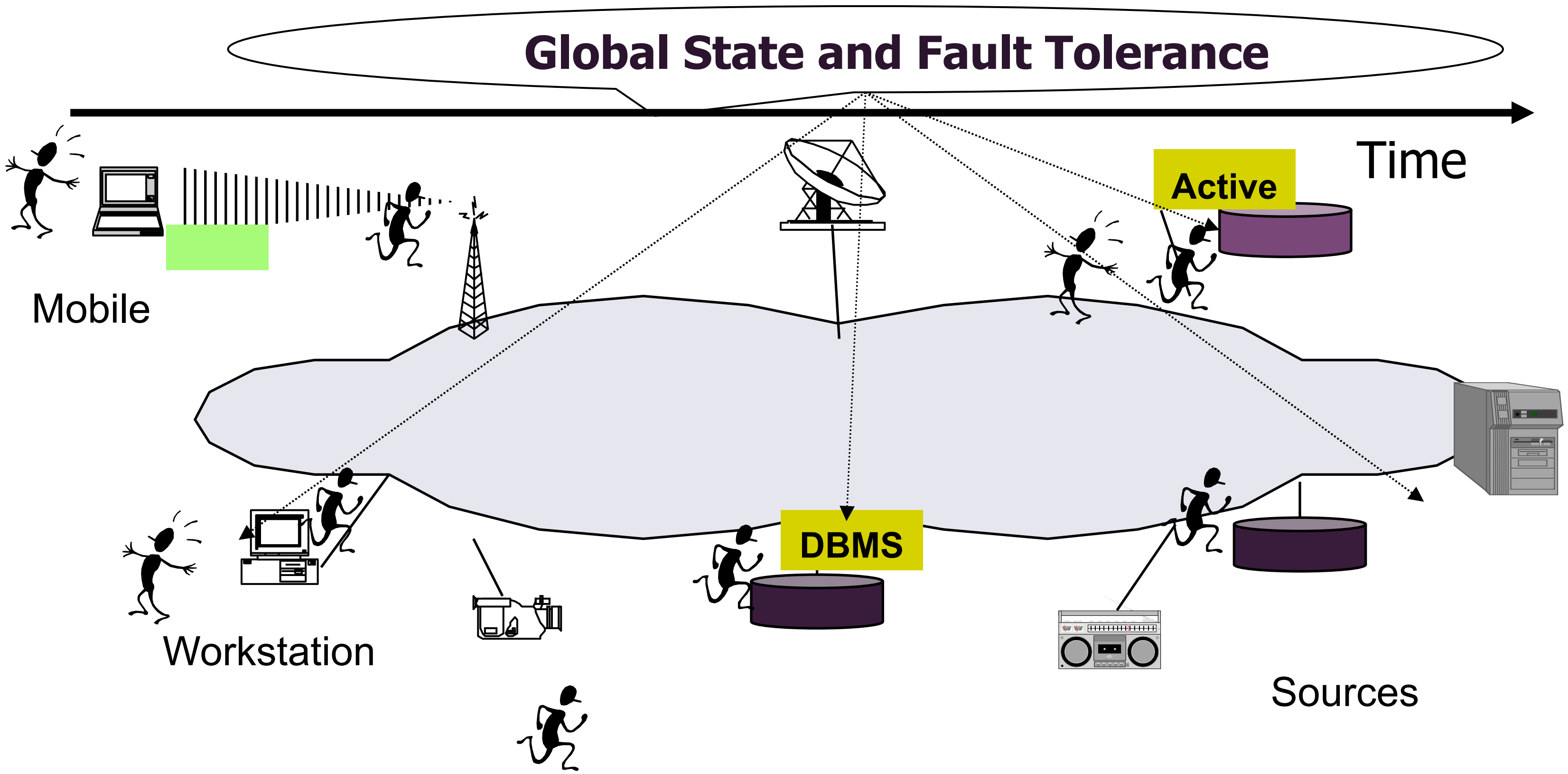
?

How is the Web concerned with these challenges

Summary

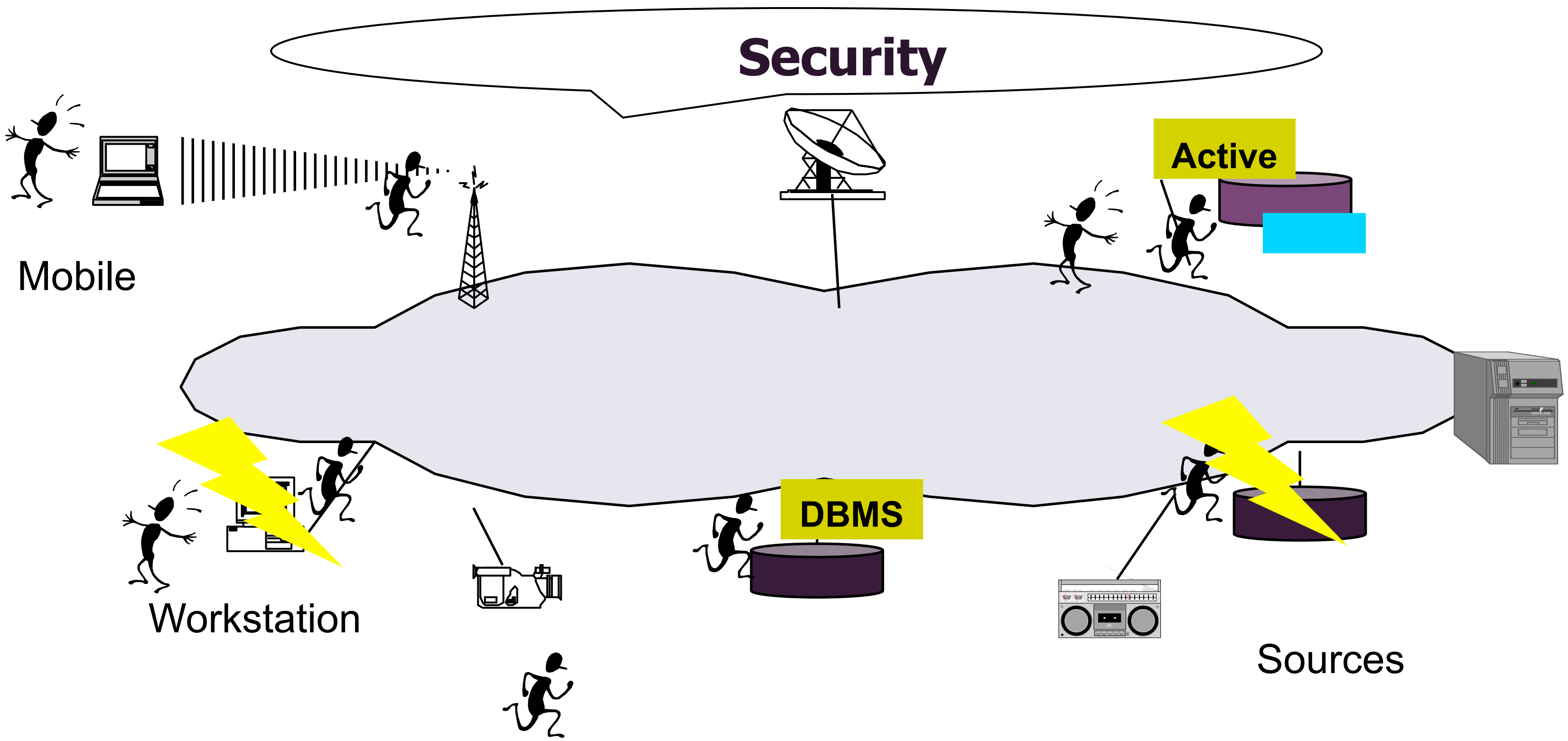

Naming and assignation
Active
Mobile
Communication
DBMS
Workstation
Sources

Summary



Global State and Fault Tolerance
Time
Active
Mobile
DBMS
Workstation
Sources

Summary



Desirables Properties

- **Tolerant to failures**
  - Should continue working even if some of its components fails
- **Tolerant to communication problems**
  - Ex. message lost and network unavailability
- **Tolerant to security attacks**
  - Ex. Confidentiality, system integrity, Denial of Service (DoS)
- **Consequences**:
  - Decisions have to be taken locally because the system' global state is unknown, it can scale dynamically and there are no guarantees concerning the network

Outline

- ✓ **Distributed Systems Basics**
  - ✓ Definition and examples
  - ✓ Challenges
  - ✓ Case study: the World Wide Web
- **System Architectures**
  - Elements
  - Layers model
  - N-Tier model
  - Case study: the Web Browser
- **Client-Server Model**
  - Characteristics
  - Implementation
  - Example: RMI

System Architecture

- Description of a system based on its composing elements and the relationships among them
  - Elements represent the *building blocks* of a system

    In practice an element may represent an individual computer or multiple computers communicating via a network

  - Relationships are described using *architectural patterns*

Architectural Elements

- Identified by answering the following questions:
  - What are the entities that are communicating in the distributed system?
  - How do they communicate (*i.e. communication paradigm*)?
  - What *roles and responsibilities* do they have in the overall architecture?
  - How are they mapped on the physical distributed infrastructure (i.e., placement)?

Architectural Elements

Communication entities and paradigms

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

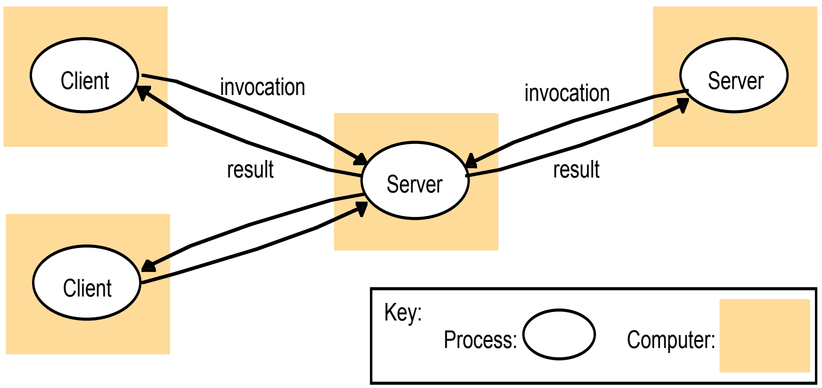
Architectural Elements

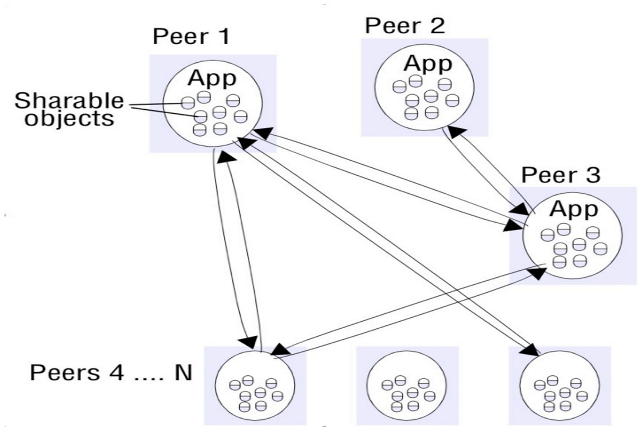
Roles and Responsibilities

- Describe an element task in a given architecture at runtime:
  - **Client-Server**
    - Simple approach for sharing of data and resources
    - **(!!)** Scales poorly
  - **Peer-to-Peer**
    - All elements in the architecture play similar roles (*peers*)
      - *i.e. there is no distinction between client and server*
    - All elements offer the same interfaces to each other (operations)

Architectural Elements

Roles and Responsibilities





Client-Server

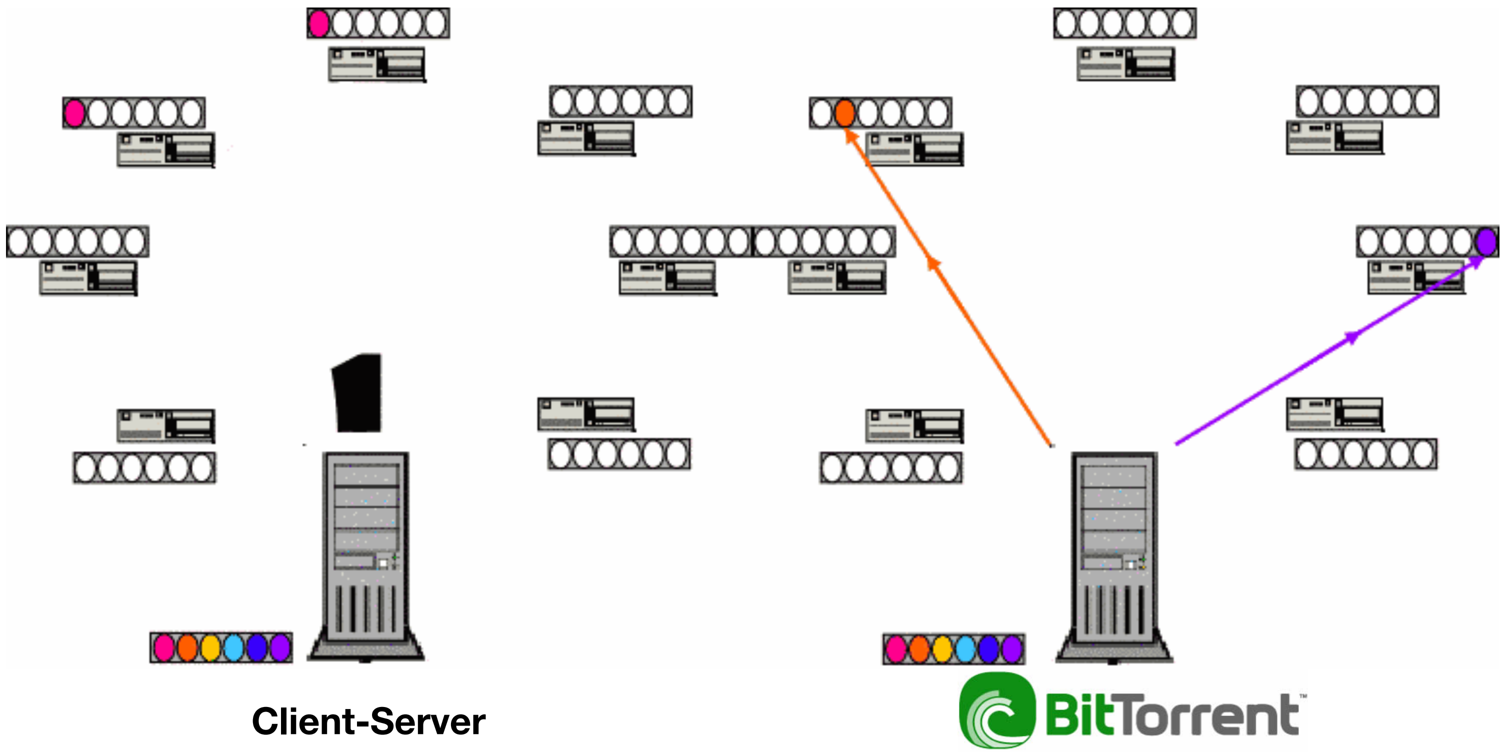
- Servers may in turn be clients of other servers

Peer-to-Peer

- Shared objects are place, retrieve and replicated among peers
- Complex than the client-server model

Architectural Elements

Roles and Responsibilities Example



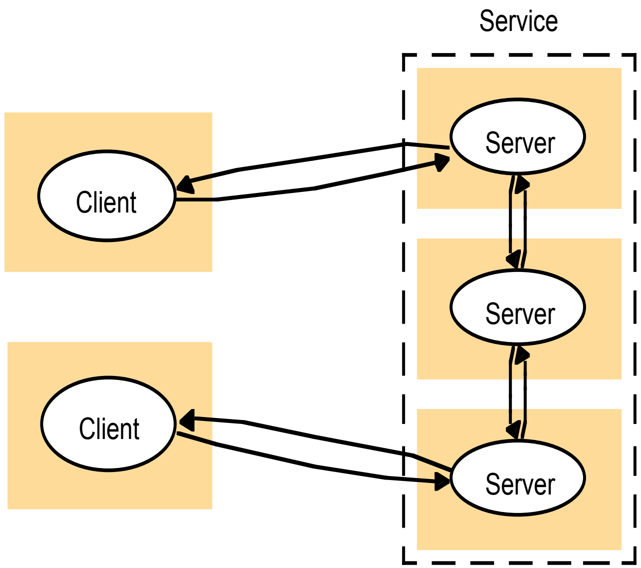
Architectural Elements

Placement

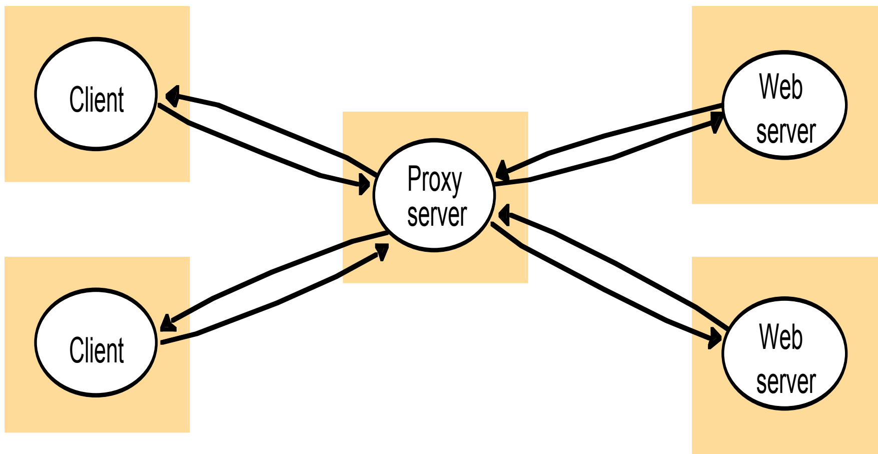
- Map elements to the underlying *physical infrastructure*
- Considerations
  - Communication patterns between entities
  - Reliability on given computers and computers current loading
  - Quality of communication between different machines
    - Ex. video games
- Strategies
  - Mapping of services to multiple servers
  - Caching
  - Mobile code

Architectural Elements

Placement Examples



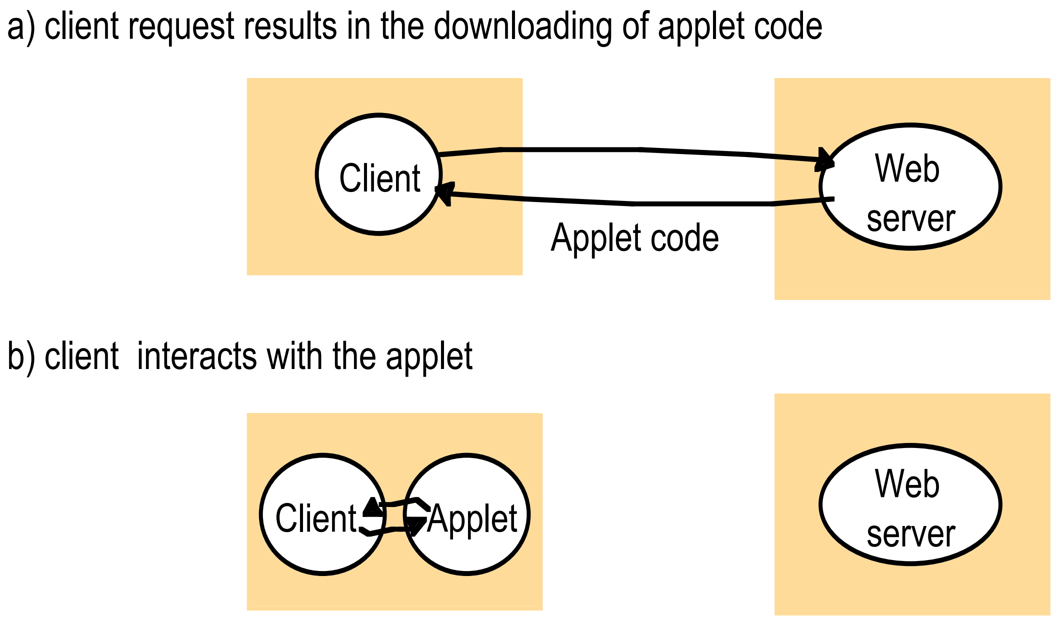
Service provided my multiple services



Caching

Architectural Elements

Placement Examples



Mobile Code

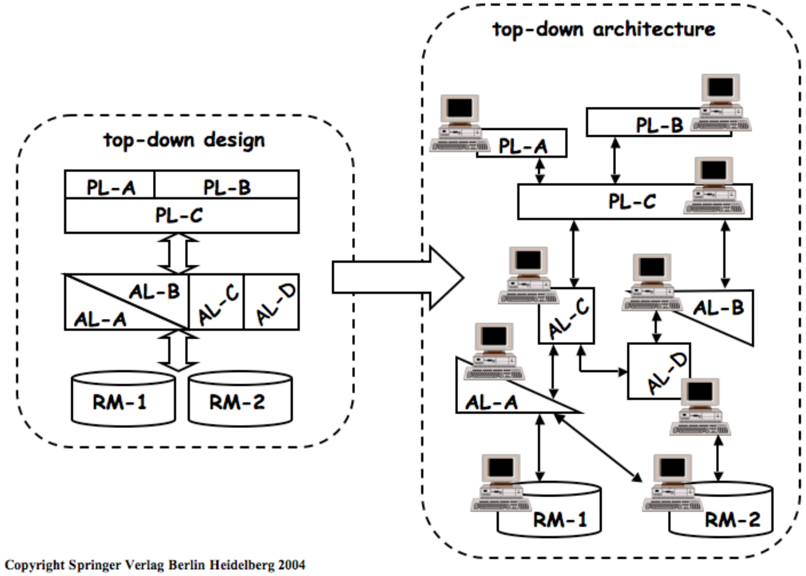
Architectural Patterns

- Build on top of the architectural elements
- Structural patterns that have shown to work in given circumstances
- Represented using architectural models
  - **Layers**
    - Abstracts complexity by partitioning the system in a number of layers, with a given layer making use of the services offered by the layer below, thus being unaware of implementation details
  - **N-tiers**
    - Complementary to the layering model
    - Organize functionality of a given layer and place this functionality into appropriate servers or physical nodes

Architectural Patterns

Layer Model

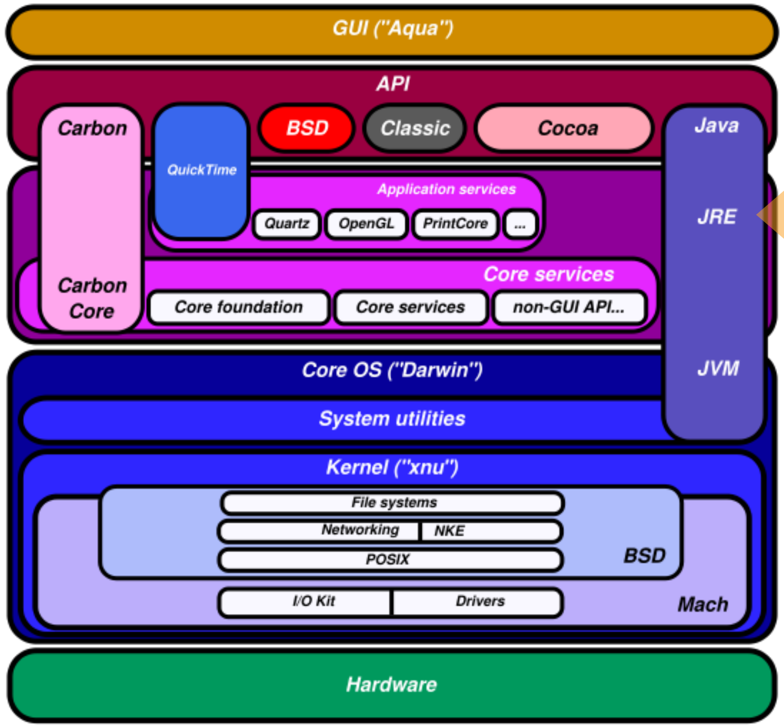
- The functionality of a system is divided in several layers
- Layers are typically not stand-alone components. Their functionality depends on other layers
- Distributed systems are designed from the beginning taking into account its composing layers

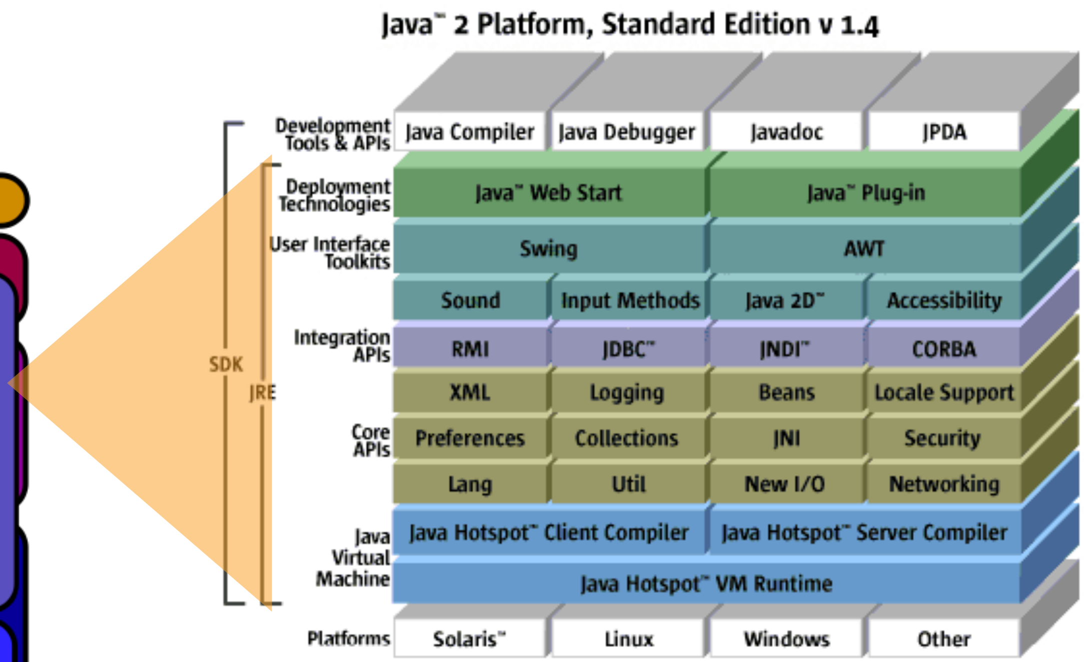


Architectural Patterns

Layer Model Example

OS X





Architectural Patterns

Layer Model Example

- **Presentation**

  Offers operations to a client for interacting with the system

- **Application Logic**

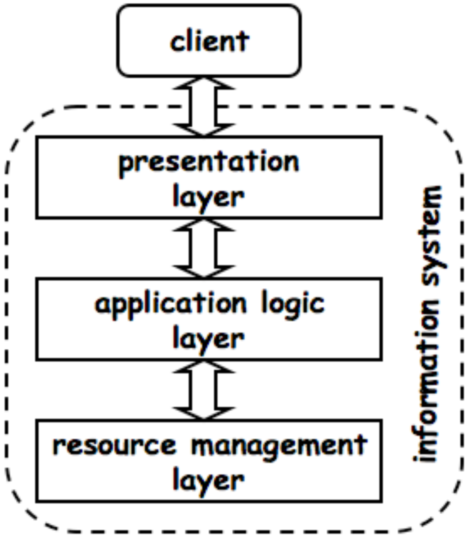
  Determines what the system actually does

  Enforces the business rules and establishes the business process

- **Resource Manager**

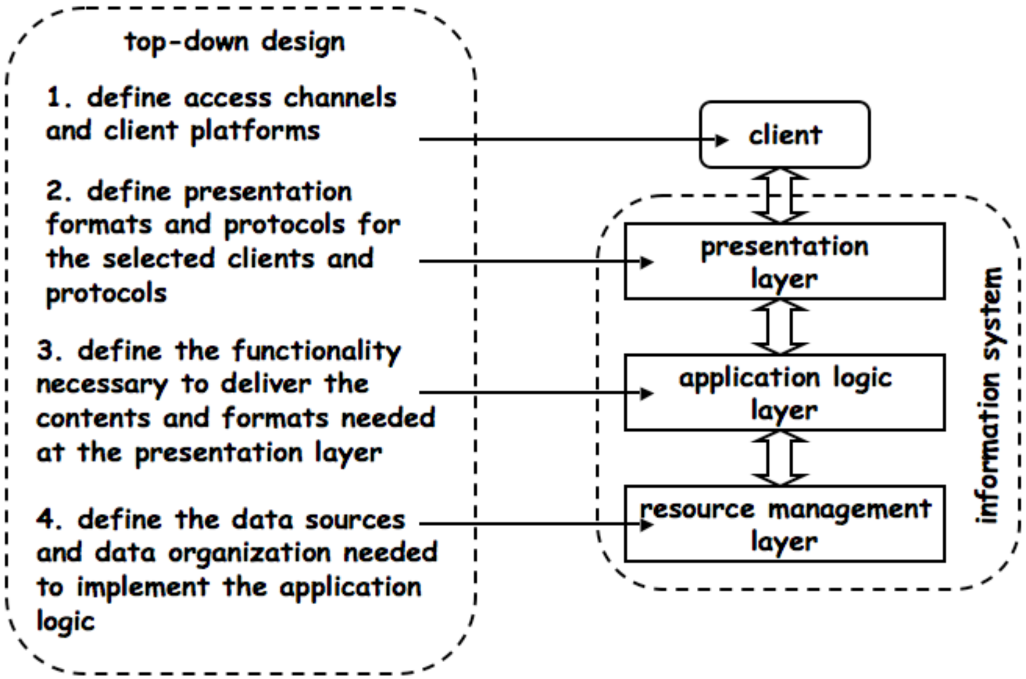
  Deals with the business logic data (e.g., storage, indexing, and retrieval).

  Can be any system providing querying capabilities and persistence (e.g. DBMS)



Architectural Patterns

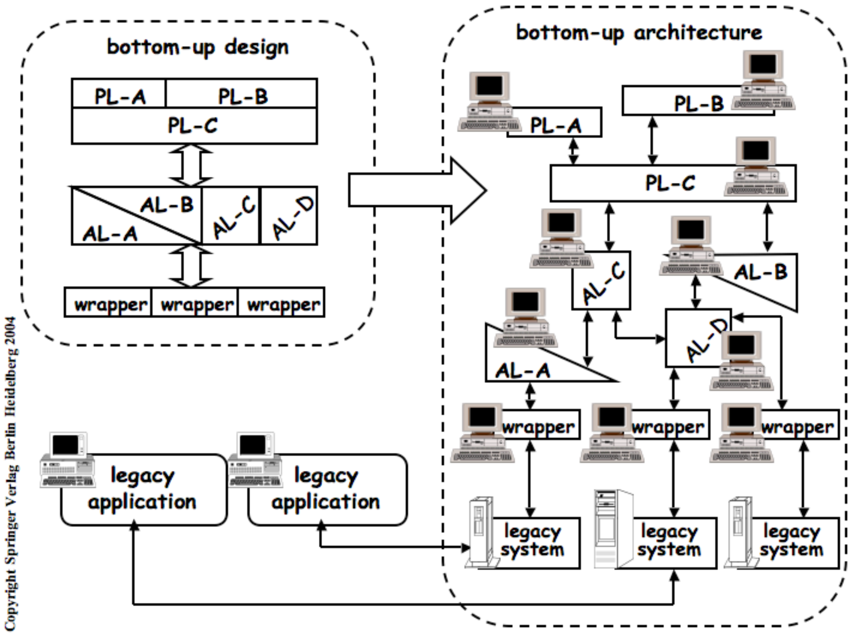
Top-down Layer Design



Architectural Patterns

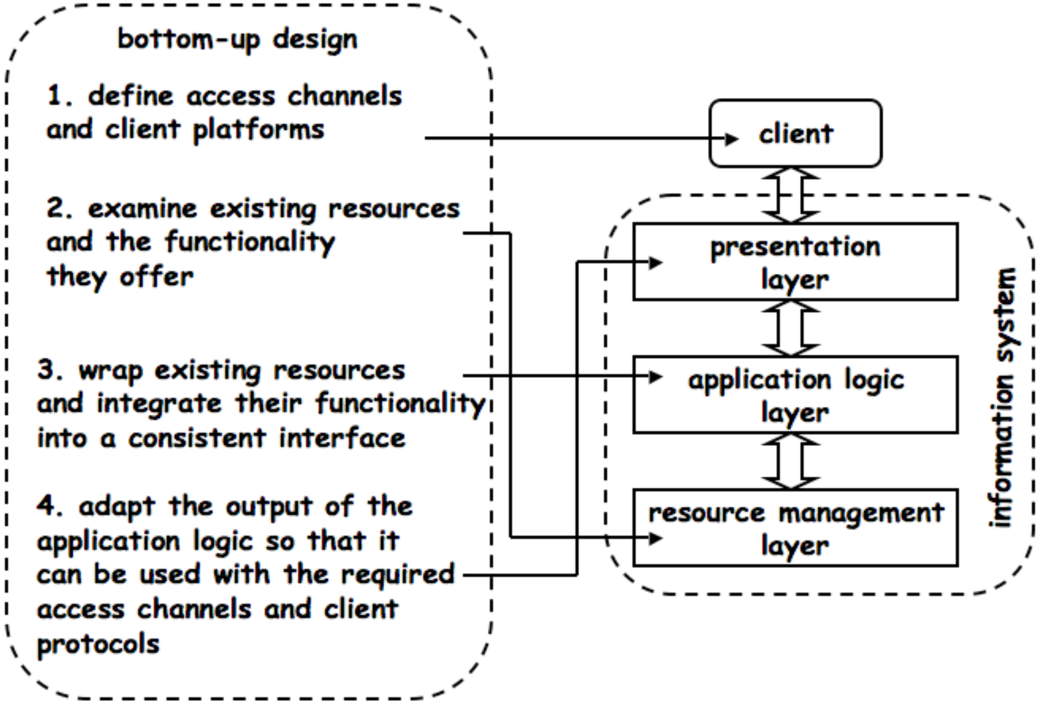
Bottom-up Layer Design

- Approach widely used because legacy systems exist and cannot be easily replaced
- Much of the work consist in coping with heterogeneity through the use of middleware's



Architectural Patterns

Bottom-up Layer Design



Architectural Patterns

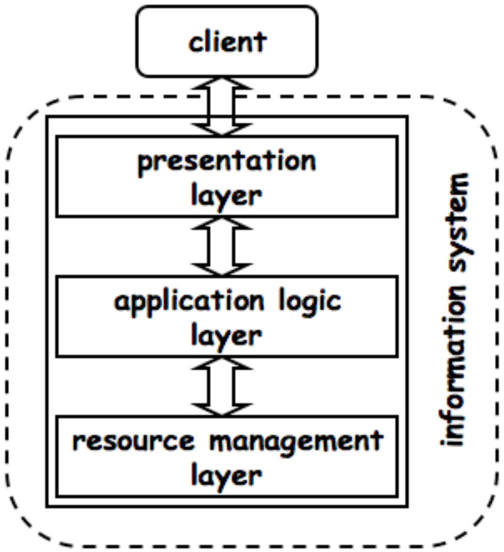
N-Tier model

- Organizes architectural elements based on their distribution
  - **1-Tier** (Monolithic)
  - **2-Tiers** (Client-Server)
  - **3-Tiers** (Middleware)
  - **N-Tiers**

Architectural Patterns

1-Tier (monolithic)

- All the layers are centralized in a single place
- Managing and controlling resources is easier
- Can be optimized by blurring the separation between layers



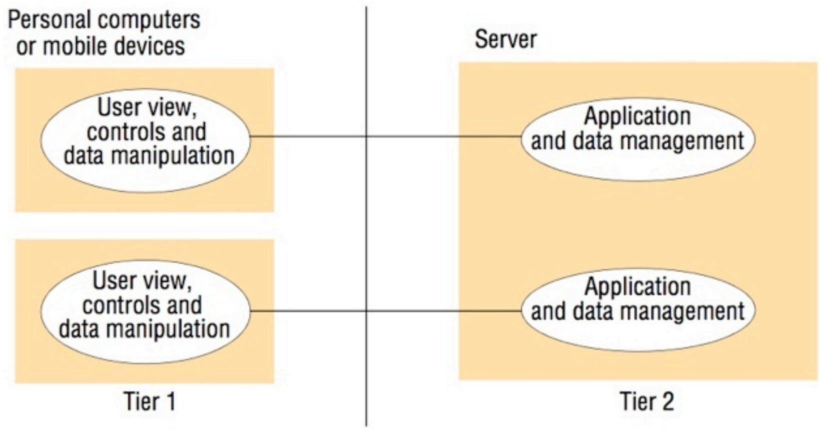
Architectural Patterns

2-Tier (client-server)

- Several presentation layers can be defined depending on what each client needs to do
- Takes advantage of clients computing power for creating more sophisticated presentation layers

  → Saves computer resources on the server
- The resource manager only sees one client: the application logic

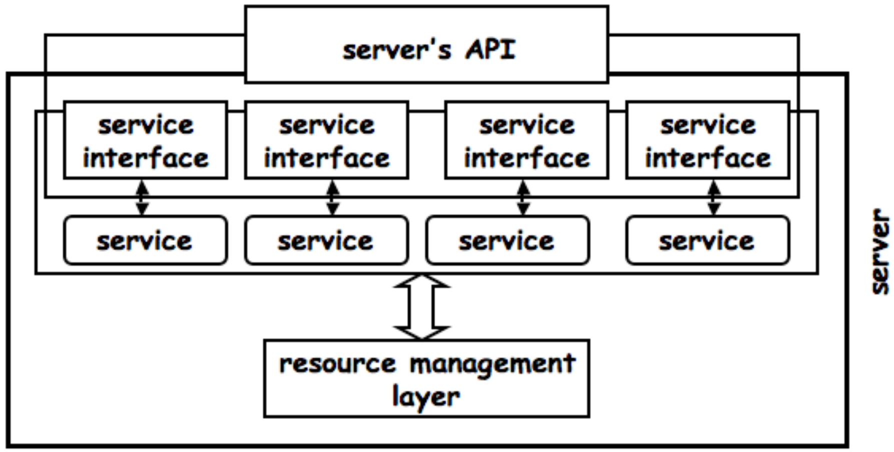
  → Helps with performance since no extra sessions are maintained



Architectural Patterns

2-Tier (client-server)

- Introduces the notion of (web) service and service interface
  - → The client invokes a service implemented by a server through an interface
- All the services provided by a server define its API (Application Programing Interface)



Architectural Patterns

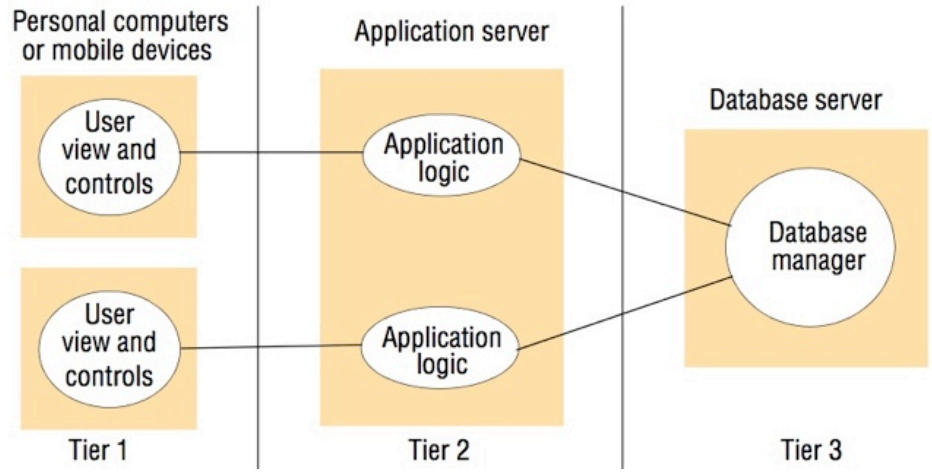
2-Tier (client-server)

- **Advantages**
  - Can off-load work from server to clients
  - Server design is still tightly coupled and can be optimized by ignoring presentation issues
  - Relatively easy to manage from a software engineering point of view
- **Disadvantages**
  - A single server can only manage a limited number of clients
  - There is no failure encapsulation. If a server fails, no clients can work
  - The load created by a client will directly affect other clients since they compete for the same resources

Architectural Patterns

3-Tier (middleware)

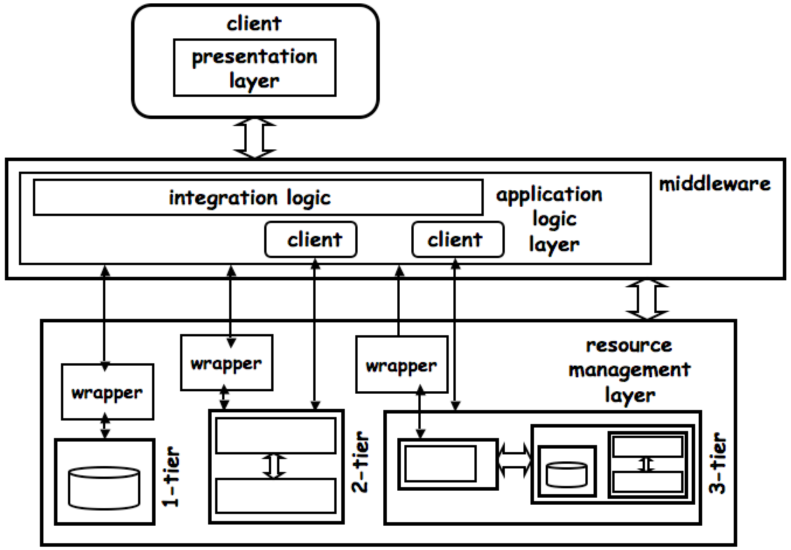
- Fully separates the three layers
- Simplifies the design of clients by reducing the number of interfaces it needs to know

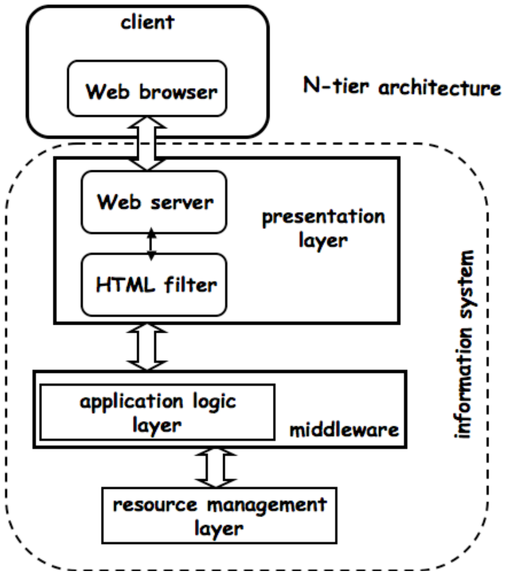


Architectural Patterns

N-Tier

- Architecture resulting from connecting several 3-tier systems to each other

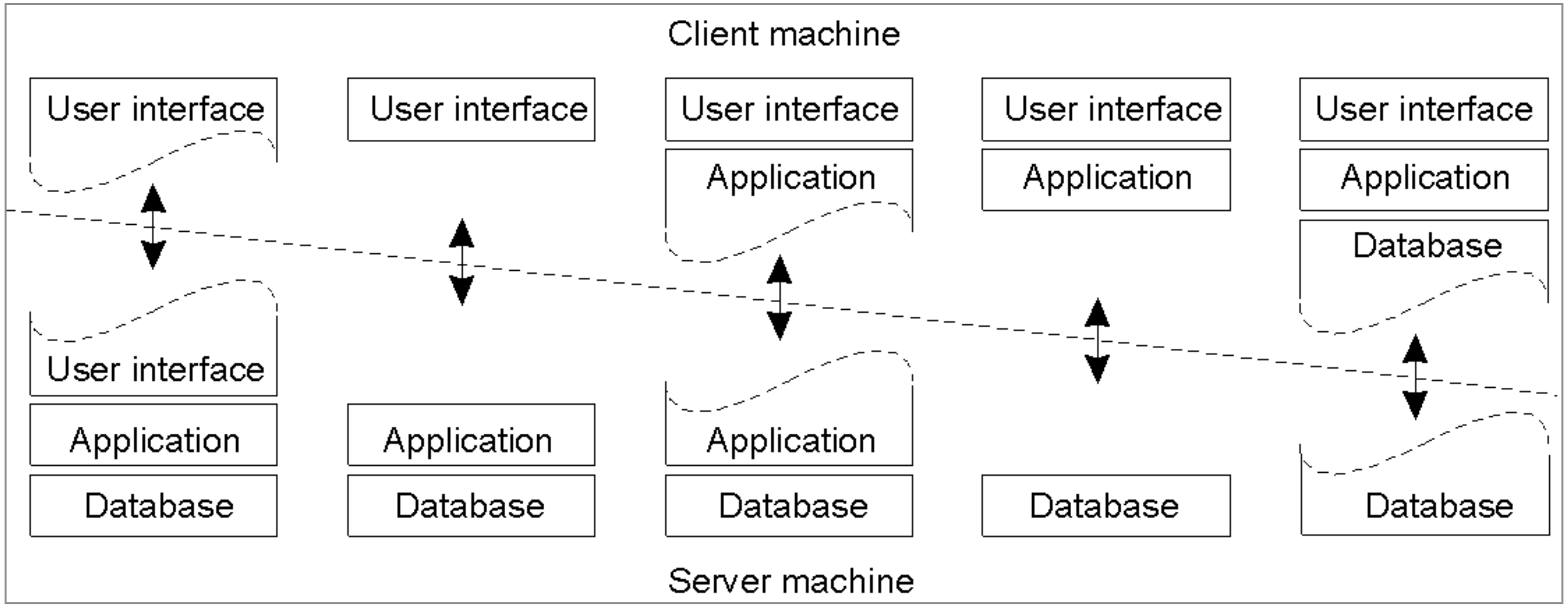




Architectural Patterns

2-Tier (client-server)

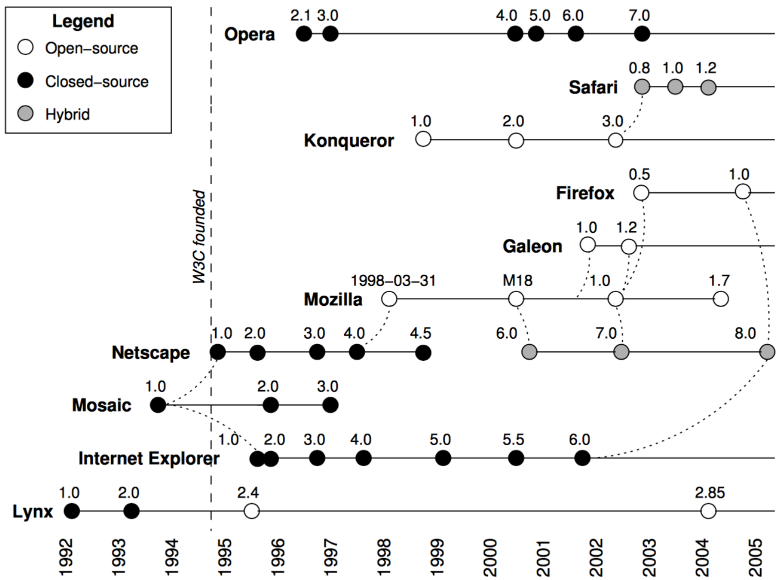
- Alternatives to client-server organizations

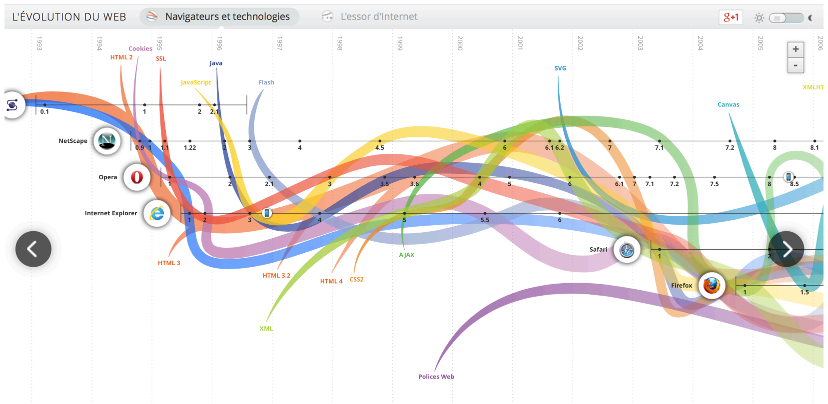


Case Study: Web Browser Architecture

Case Study: Web Browser Architecture

Browser Evolution



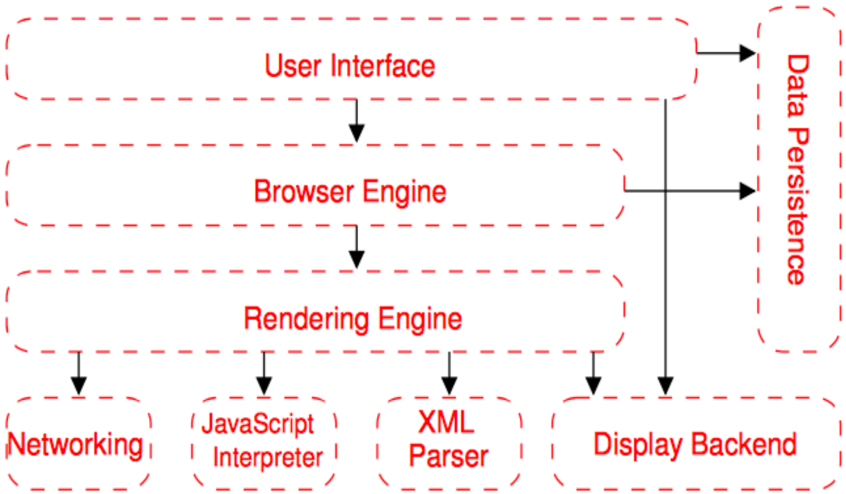


www.evolutionoftheweb.com

Case Study: Web Browser Architecture

Reference Architecture

- Shows a browser' fundamental components and their relationships
- Aids in analyzing trade-offs between different design options
- Template for designing new browsers and re-engineering existing ones

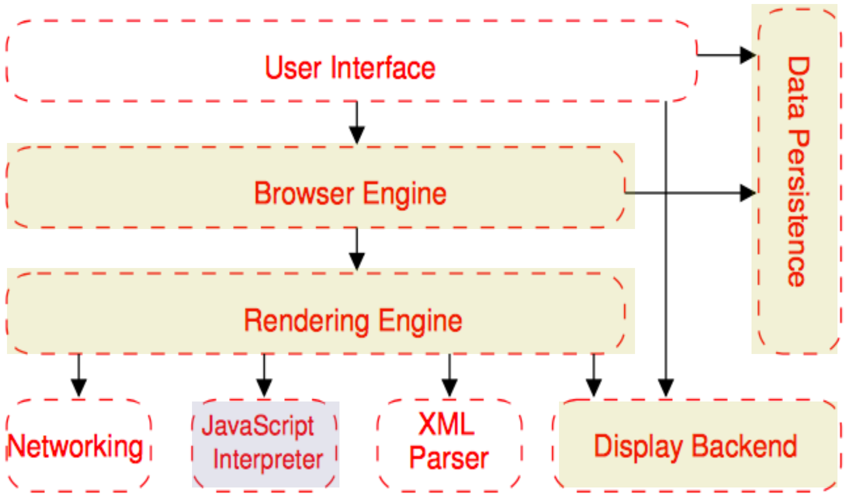


Derived from Mozilla and Konqueror

Case Study: Web Browser Architecture

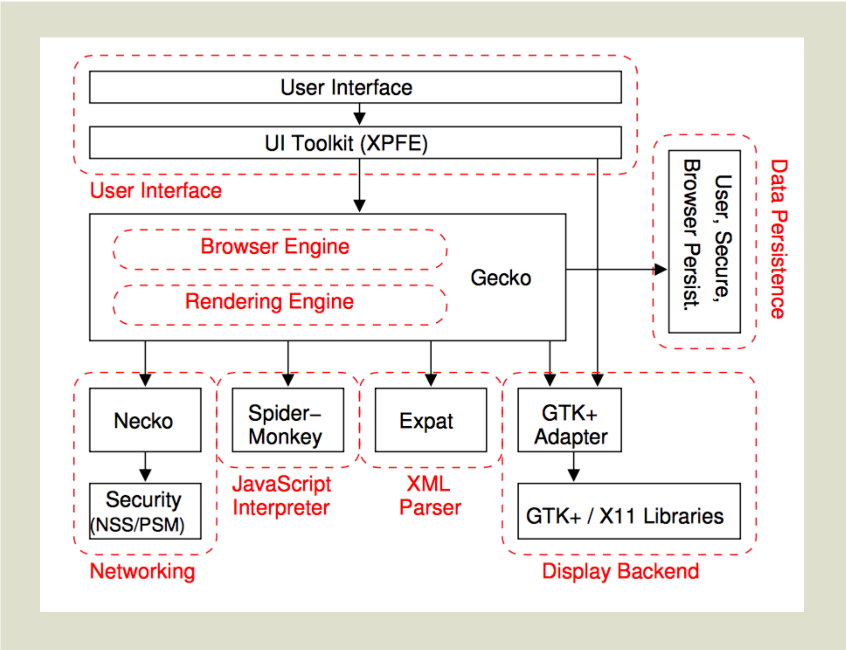
Reference Architecture

- **Browser Engine**
  Provides a high-level interface for querying and manipulating the *Rendering Engine*
- **Rendering Engine**
  Performs parsing and layout for HTML documents (optionally styled with CSS)
- **Display Backend**
  Provides drawing and windowing primitives, user interface widgets, and fonts
- **Data Persistence**
  Stores data associated with the browsing session on disk (including bookmarks, cookies, and cache)

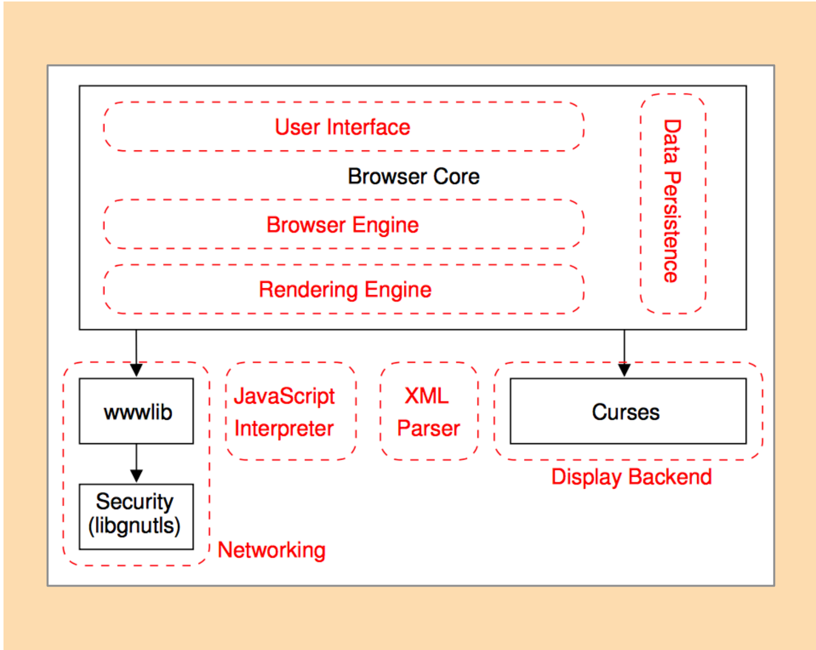


Case Study: Web Browser Architecture

Examples



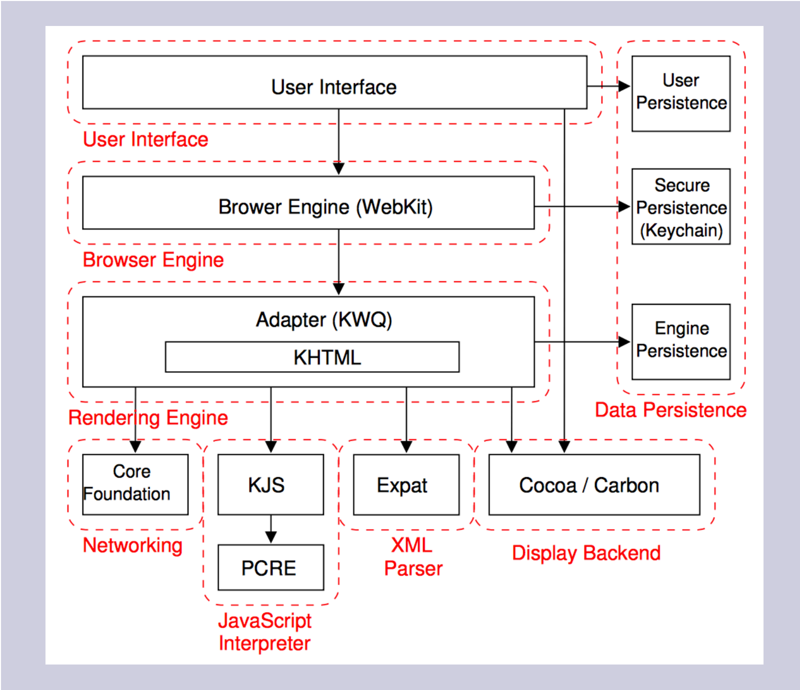
Mozilla



Lynx

Case Study: Web Browser Architecture

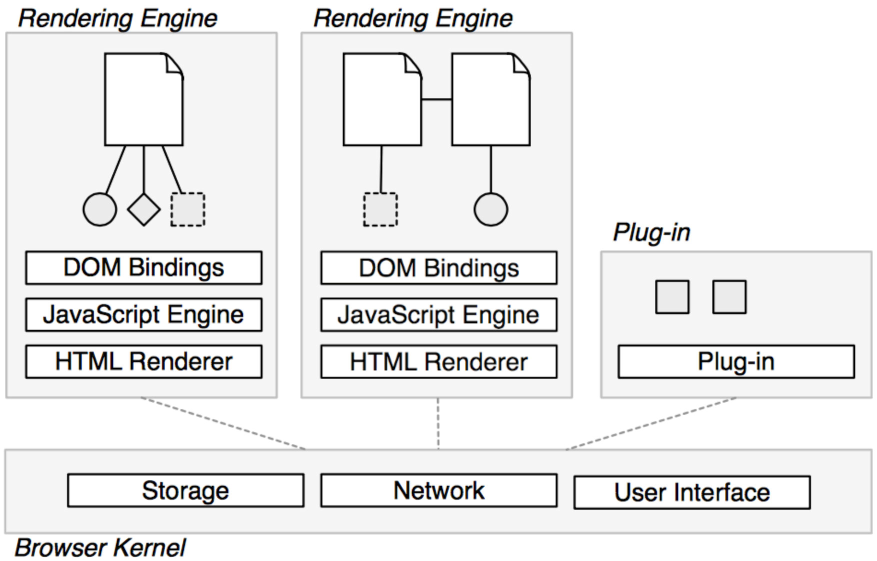
Examples

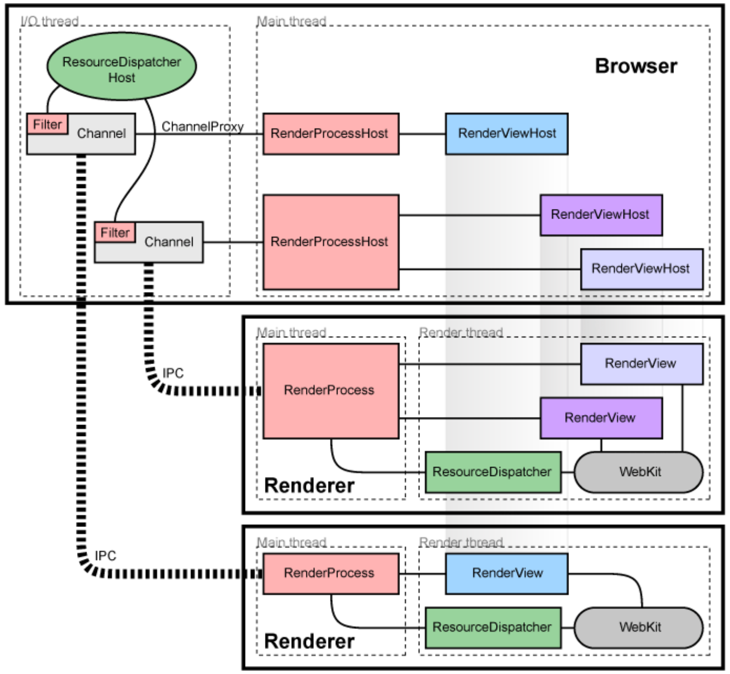


Safari

Case Study: Web Browser Architecture

Isolating Programs



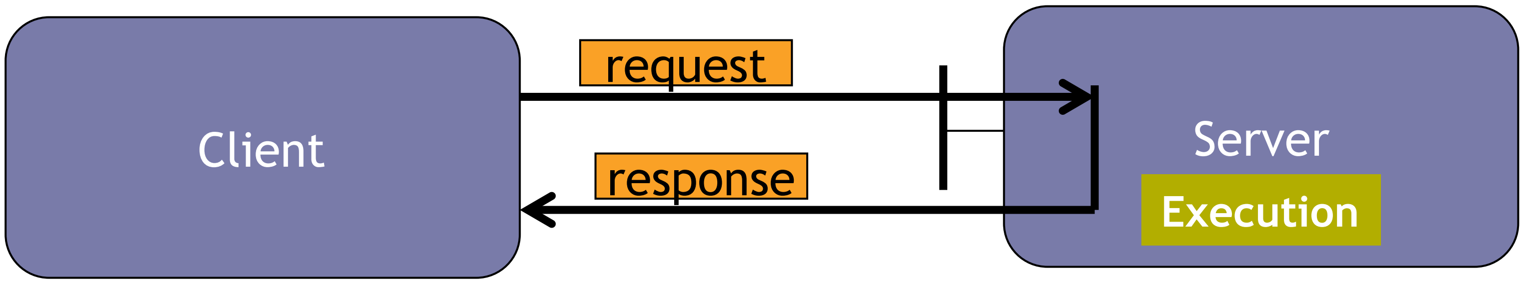


Chrome

Outline

- ✓ **Distributed Systems Basics**
  - ✓ Definition
  - ✓ Minimal requirements
  - ✓ Desirable properties
- ✓ **System Architectures**
  - ✓ Layers
  - ✓ N-Tier model
- **Client-Server Model**
  - Characteristics
  - Implementation
  - Example: RMI

Client-Server Abstraction



Client-Server Characteristics

- **State Management**
  - Server-side: persistent or not
  - Client-side: stateful or stateless
- **Communication Model**
  - Connected or disconnected mode (datagrams)
  - Synchronous or asynchronous
- **Server-side Execution Model**
  - One or more processes
  - Pool of processes or processes on-demand

Server without Persistent Data

- The execution only uses the input parameters
  - Does not modify the state of the server
- Ideal situation for:
  - Fault tolerance
  - Controlling concurrency
- Example
  - A service for computing mathematical functions

Server with Persistent Data

- Successive executions manipulates persistent data
  - Modifies the execution context
  - Introduces problems for controlling concurrent access to resources
  - Fault tolerance is not guaranteed
- Examples
  - Database Server
  - Distributed File System

Stateless Service

- The server does not keep track of client requests
- Successive request are independents
  - Even if global data is modified, the current request dost not have any relation with previews ones
  - The order among request is not important
- Example

  The service of *clock synchronization* of a network

  → NTP service (Network Time Protocol)

Stateful Service

- Requests are executed based on the state produced by previews requests
- Order among requests is important
- Examples
  - Sequential access to the content of a file
    → depends on the file's pointer position
  - Calling a remote method
    → the result of the call depends on the state of the object

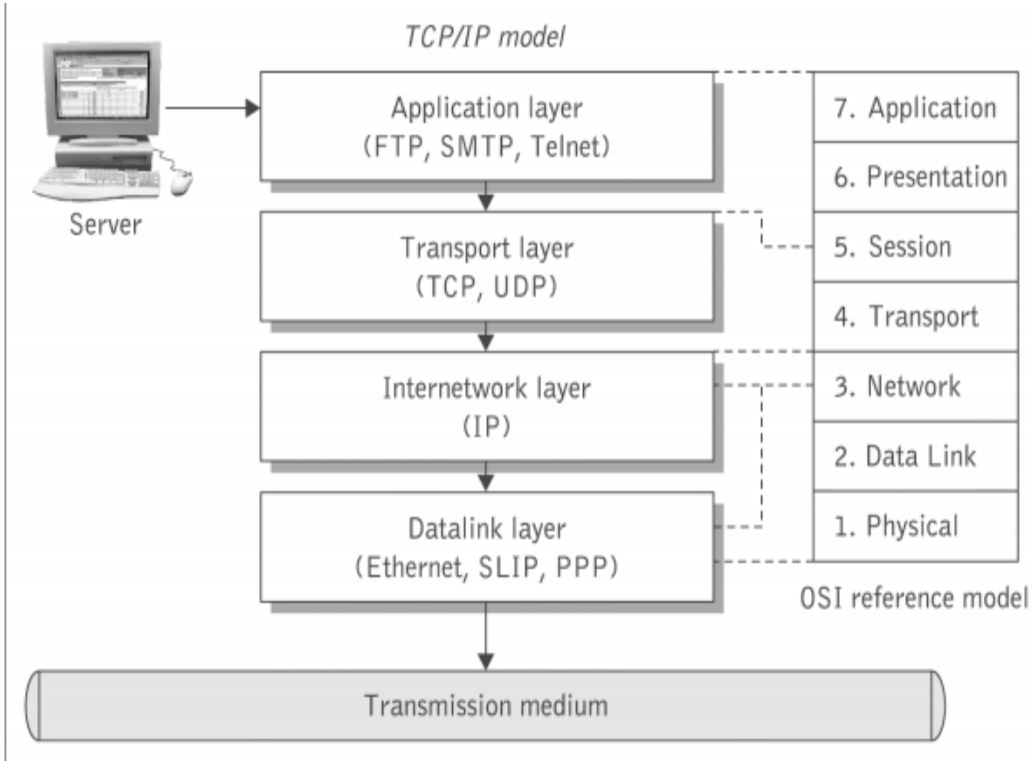
Client-Server Characteristics

- ✓ **State Management**
  - ✓ Server-side: persistent or not
  - ✓ Client-side: stateful or stateless
- **Communication Model**
  - Connected or disconnected mode (datagrams)
  - Synchronous or asynchronous
- **Server-side Execution Model**
  - One or more processes
  - Pool of processes or processes on-demand

Connection Modes

- The main difference resides in the reliability of message delivery
- **Connection oriented**
  - Message delivery is guaranteed
  - Order among messages is respected
  - Free of error (delivery is retried when necessary)
- **Datagram oriented**
  - Follows the "best-effort" approach (i.e., there is not guarantees of message delivery)
  - Message can arrived duplicated
  - Order is not respected

Communication Protocols

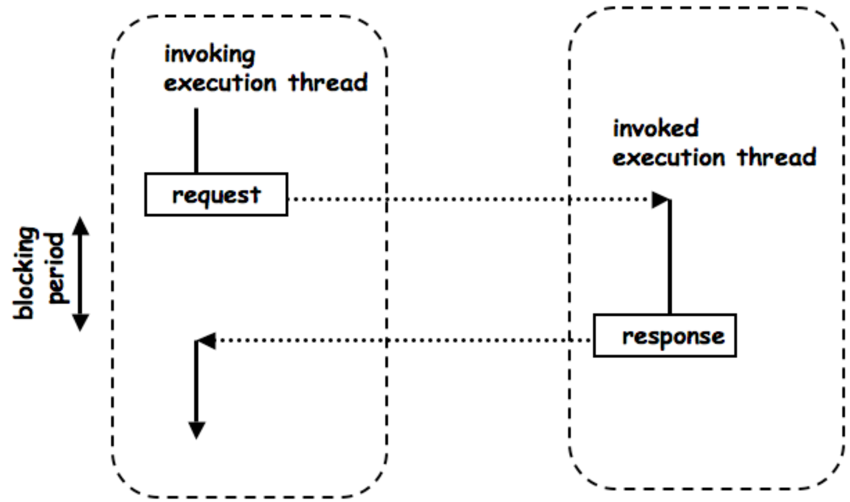


Synchronous Interaction

- Traditionally used for developing distributed systems
  - Client waits while server processes a request (blocking call)
  - Requires both parties to be on-line

- **Advantage**
  - Simple to understand and implement
  - Failures are simple to manage

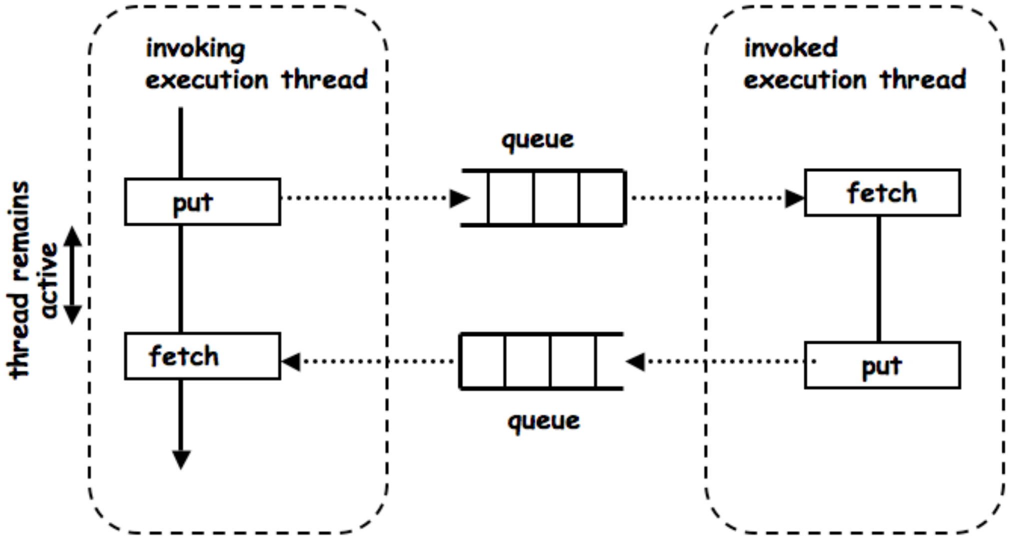
- **Disadvantages**
  - Connection overhead
  - Higher probability of failures
  - Solutions:
    - → Transactions
    - → Asynchronous interactions



Asynchronous Interaction (i)

- Calls to servers are non-blocking thus clients can continue running
  - Clients checks at different times to see if a response is ready
  - Typically implemented via message queues
- **Disadvantage**
  - Adds complexity to client architecture
- **Advantages**
  - More modular
  - More distribution modes (multicast, replication, message coalescing, etc.),
  - More natural way to implement complex interactions between heterogeneous systems

Asynchronous Interaction (ii)



Client-Server Characteristics

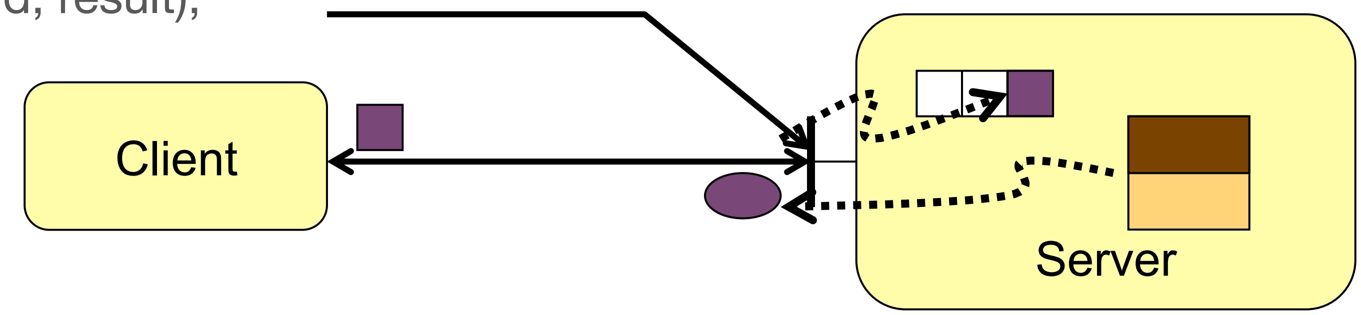
- ✓ **State Management**
  - ✓ Server-side: persistent or not
  - ✓ Client-side: stateful or stateless
- ✓ **Communication Model**
  - ✓ Connected or disconnected mode (datagrams)
  - ✓ Synchronous or asynchronous
- **Server-side Execution Model**
  - One or more processes
  - Pool of processes or processes on-demand

Execution models

- **Iterative execution**
  - Based on a single process
- **Concurrent execution**
  - Based on multiple processes or threads
    - Processes are created on-demand
    - Processes are selected from a "pool of processes"

Single Process Execution

```
while (true) {
  receive(client_id, message);
  extract(message, service_id, params);
  result = do_service[service_id](params);
  send(client_id, result);
}
```



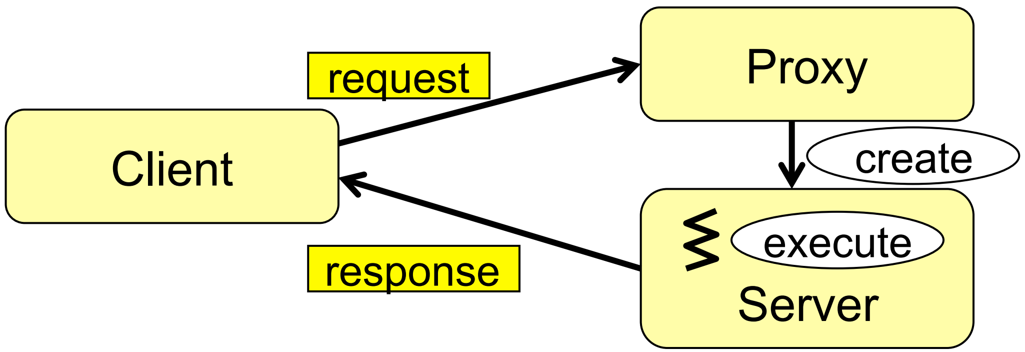
Processes Created On-Demand

Proxy

```
while (true) {
  receive(client_id, message);
  extract(message, service_id, params);
  create_process(client_id, service_id, params);
}
```

Server

```
// código a ejecutar
result = do_service[service_id](params);
send(client_id, result);
exit;
```



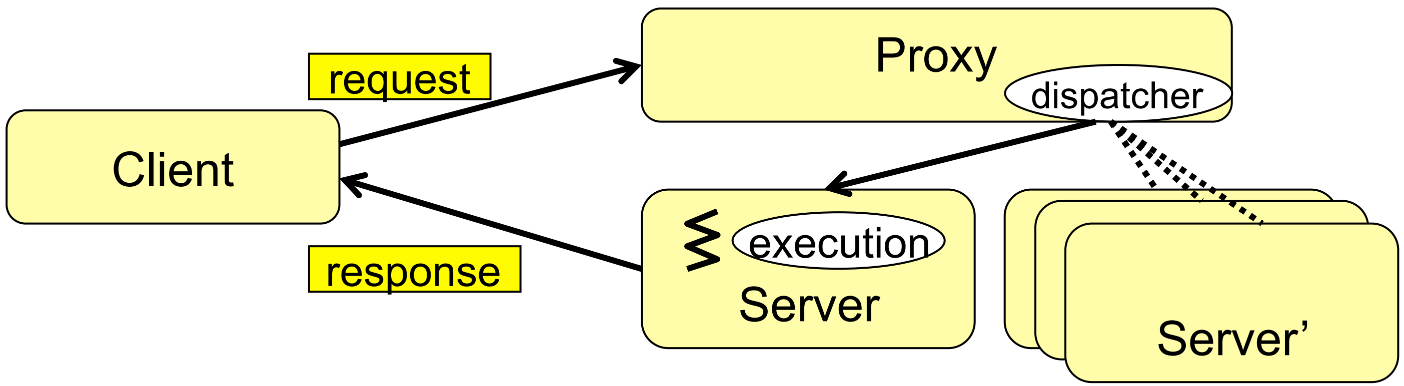
Pool of Processes

Proxy

```
while (true) {
  receive(client_id, message);
  extract(message, service_id, params);
  dispatch(client_id, service_id, params);
}
```

Servicio

```
// código a ejecutar
result = do_service[service_id](params);
send(client_id, result);
exit;
```



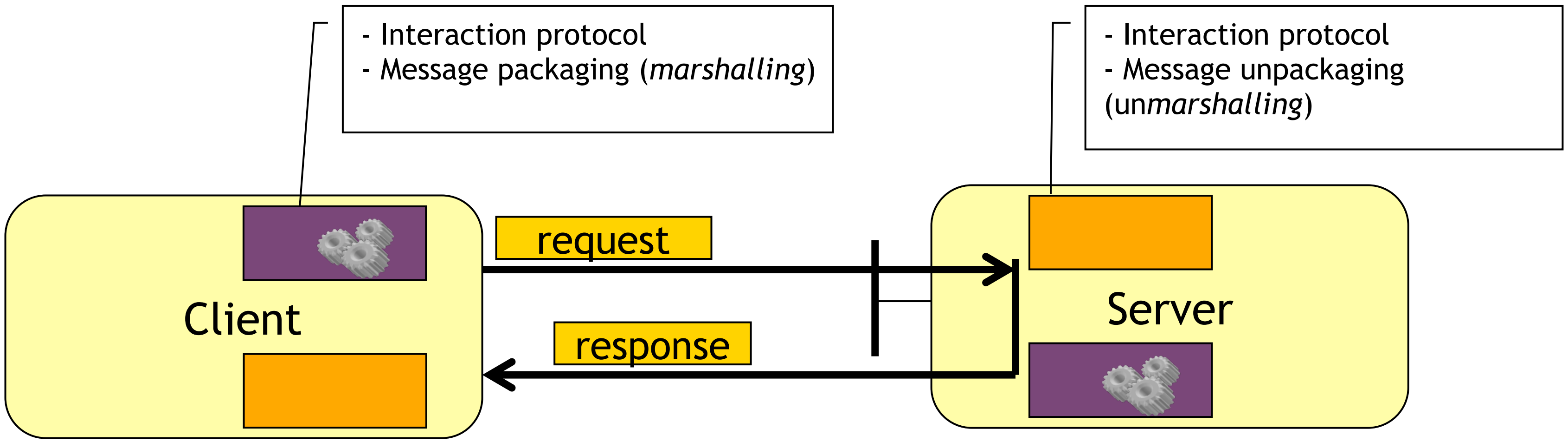
Client-Server Characteristics

- **State Management**
  - Server-side: persistent or not
  - Client-side: stateful or stateless
- **Communication Model**
  - Connected or disconnected mode (datagrams)
  - Synchronous or asynchronous
- **Server-side Execution Model**
  - One or more processes
  - Pool of processes or processes on-demand

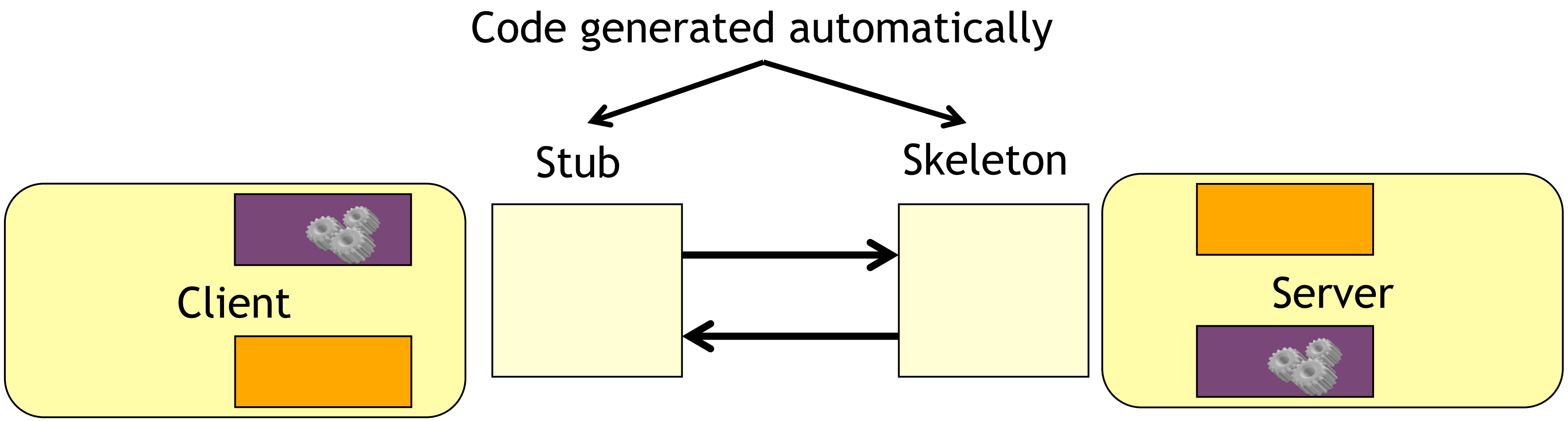
Client-Server Implementation

- Based on systems low level primitives
  - Socket Programming
- Based on middleware's
  - Remote Procedure Calls (RPC)
  - Remote Method Invocation (RMI)

C/S: Low Level Primitives



C/S: Middleware RPC

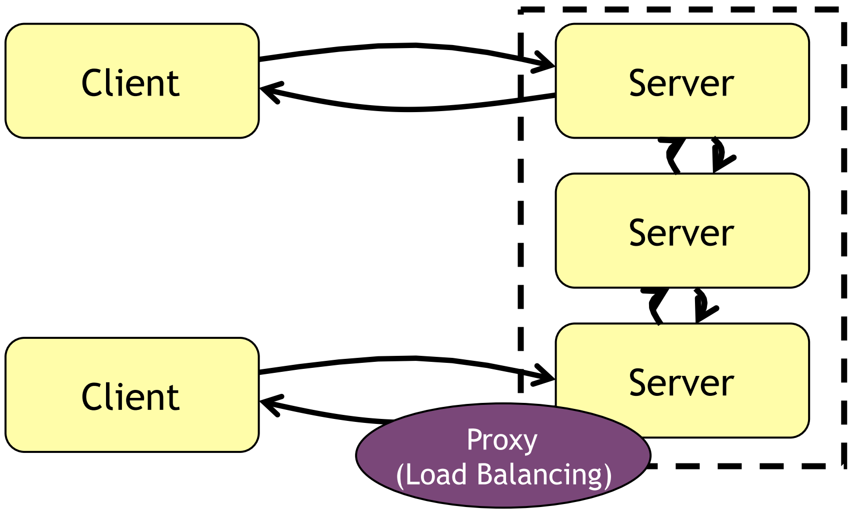


Multiple clients <–> single server

- **Limitations**
  - Server-side can suffer from bottlenecks
  - Vulnerable to failures
  - Difficult to scale

Multiple clients <–> multiple servers

- **Advantages**
  - Load balancing
  - Fault tolerance and scalability



?