CLIENT-SERVER ARCHITECTURE
Recap

- Public, private, and hybrid cloud?
- SaaS, PaaS and IaaS?
Subjects

• Design considerations
• Architectural styles
• System architectures
• Example
Design Considerations
Design Considerations

- Support sharing of resources
  - The network is the computer
- Distribution transparency
- Openness
- Scalability
## Design Considerations: Distributed Transparency

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and how an object is accessed</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where an object is located</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that an object may be moved to another location while <em>in use</em></td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that an object may move to another location</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that an object is replicated</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that an object may be shared by several independent users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide the failure and recovery of an object</td>
</tr>
</tbody>
</table>
Design Considerations: Distributed Transparency

• Aiming at full distribution transparency may be too much
  • There are communication latencies that cannot be hidden
  • Completely hiding failures of networks and nodes is (theoretically and practically) impossible
  • You cannot distinguish a slow computer from a failing one

• Full transparency will cost performance, exposing distribution of the system
  • Keeping replicas exactly up-to-date with master takes time
  • Immediately flushing write operations to disk for fault tolerance
Design Considerations: Distributed Transparency

• Exposing distribution may be good
• Making use of location-based services (finding your nearby friends
• When dealing with users in different time zones
• When it makes it easier for a user to understand what’s going on
  • E.g., when a server does not respond for a long time, report it as failing

• Distribution transparency is a nice goal, but achieving it is a different story.
Design Considerations: Openness

Interact with services from other open systems, irrespective of the underlying environment:

• Systems should conform to well-defined interfaces
• Systems should easily interoperate
• Systems should support portability of applications
• Systems should be easily extensible
Design Considerations: Failures

- No single point of failure
- Data replication
- Automated actions on failure
- Logging
Design Considerations: Scalability

At least three components:

• Number of users and/or processes (size scalability)
• Maximum distance between nodes (geographical scalability)
• Number of administrative domains (administrative scalability)

• Most systems account only, to a certain extent, for size scalability.
• Often a solution: Multiple powerful servers operating independently in parallel. Today, the challenge still lies in geographical and administrative scalability
Design Considerations: Scalability

Root causes for scalability problems with centralized solution:

- The computational capacity, limited by the CPUs/GPUs.
- The storage capacity, including the transfer rate between CPUs and disks.
- The network between the user and the centralized service.
Design Considerations: Geographical Scalability

• Can not simply go from LAN to WAN.
  • Many distributed systems assume synchronous client-server interactions:
    • Client sends request and waits or answer. Latency may easily prohibit this scheme.

• WAN links are often inherently unreliable:
  • Simply moving streaming video from LAN to WAN is bound to fail.

• Lack of multipoint communication
  • So that a simple search broadcast cannot be deployed. Solution is to develop separate naming and directory services (having their own scalability problems)
Design Considerations: Administrative Scalability

• For example, conflicting policies concerning usage (and thus payment), management, and security.
  • Typical issue in hybrid clouds.
Design Considerations: Techniques for scaling

- Loose coupling of the components
- Stateless design
- Database choice and design
Design Considerations:
Techniques for scaling

- Partition data and computations across multiple machines
- Move computations to clients (Java applets)
- Decentralized naming services (DNS)
- Decentralized information systems (WWW)
Design Considerations: Techniques For Scaling

- Replication and caching: Make copies of data available at different machines
  - Replicated file servers and databases
  - Mirrored web sites
  - Web caches (in browsers and proxies)
  - File caching (at server and client)
Architecture
Architecture

• **Software architecture**
  • Logical organization and interaction of software components

• **System architecture**
  • Instantiation of a software architecture on real machines
Architectural Styles

• The notion of an architectural style
  • Formulated in terms of (replaceable) components, their connections and the data exchanged between them
  • A component is a modular unit with well-defined requirements and provided interfaces, replaceable within its environment
  • A connector is a mechanism mediating communication, coordination or cooperation among components
Architectural Styles

• Important architectural styles for distributed systems
  • Layered architectures
  • Object-based architectures
  • Data-centered architectures
  • Event-based architectures
Layered Architectures

- Component at layer $L_i$ is allowed to call component at the underlying layer $L_{i-1}$, but not the other way around.

- Control generally flows layer to layer:
  - Request go down
  - Results flow upward

- Widely adopted in network

- Example: OSI model
Layered Architectures

- Layered networking architectures
- Layering allows mastering the complexity
  - Explicit structure allows identification and relationship of complex system’s pieces
  - Modularization eases maintenance and updating of system
  - Change of implementation of a layer’s service transparent to the rest of the system
Layered Architectures

Protocol, service, interface
Application Layering

- **Application-interface** layer contains units for interfacing to users or external applications.

- **Processing layer** contains the functions of an application, i.e., without specific data.

- **Data layer** contains the data that a client wants to manipulate through the application components.
APPLICATION LAYERING

EXAMPLE: A SIMPLE SEARCH ENGINE
Application Layering

User-interface Level
- Typically implemented by the client
- Consists of programs that allow end users to interact with applications
- Great variation in functionality provided by the user interfaces

Processing Level
- Contains core functionality of an application

Data Level
- Contains programs that maintain the data on which the application operate
- Persistency
- Consistency
Object-based Architectures

- Objects correspond to components
- Components are connected via a (remote) procedure call mechanism
RESTful Architectures

• View a distributed system as a collection of resources, individually managed by components. Resources may be added, removed, retrieved, and modified by (remote) applications.
  • Resources are identified through a single naming scheme
  • All services offer the same interface
  • Messages sent to or from a service are fully self-described
  • After executing an operation at a service, that component forgets everything about the caller
## Basic Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>Create a new resource</td>
</tr>
<tr>
<td>GET</td>
<td>Retrieve the state of a resource in some representation</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete a resource</td>
</tr>
<tr>
<td>POST</td>
<td>Modify a resource by transferring a new state</td>
</tr>
</tbody>
</table>
Data-Centered Architectures

• Processes communicate through a common (passive or active) repository

• Examples
  • Distributed file systems
  • Web-based data services
## Temporal and Referential Coupling

<table>
<thead>
<tr>
<th>Referential Coupling</th>
<th>Temporally coupled</th>
<th>Temporally decoupled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Referentially coupled</td>
<td>Direct</td>
<td>Mailbox</td>
</tr>
<tr>
<td>Referentially decoupled</td>
<td>Event-based</td>
<td>Shared data space</td>
</tr>
</tbody>
</table>
Event-based Architectures

- Processes communicate through propagation of events
- Events can optionally carry data
- **Publish/subscribe systems**
  - Processes publish events
  - Only processes having subscribes to particular events will receive them
- Allows loose coupling of processing
  - Processes need not explicitly refer to each other (referential decoupling)
Shared Data Spaces

- Event-based architecture combined with data-centered architecture
- Processes are also decoupled in time (they need not both be active when communication takes place)
- Data can be accessed also using a description instead of explicit reference
System Architectures

• Centralized architectures
• Decentralized architectures (such as Peer-to-Peer)
• Hybrid architectures
Centralized Architectures

- Client-server model
- Server
- Client
- Request-reply behaviour
Call Semantics And Transmission Failures

• Ideally: Exactly-once
• Zero-or-more ("maybe"): Service may or may not have been called
• At-least-once: Keep requesting service until valid response arrives at client
• At-most-once: No reply may mean that no execution took place
• Idempotent vs non-idempotent operations
  • Idempotent (repeatable) operation can be repeated multiple times without harm
Multi-tiered Architectures

- Simplest organization is to have only two types of machines
  - Client: Machine containing only the programs implementing (part of) the user-interface level
  - Server: Machine containing the rest (programs implementing the processing/application and data level)
Multi-tiered Architectures

- Three-tiered architecture
  - Single server is replaced by multiple servers running on distributed machines
  - Server sometimes acts as the client
- Vertical distribution
  - Logically different components are placed on different machines
Web: Original Simple Client-server Architecture

- **Server**
  - Maintains collection of documents
  - Accepts requests for fetching document and transfers it to client
  - Can also accept requests for storing new documents

- **Client** interacts with servers through a special application (browsers)
Typical operation sequence

1. Browser requests a document referred by means of URL
2. Web server fetches document from local file
3. Web server transmits document to browser
4. Browser displays document
Web: Multi-tiered Architectures (1)

- **CGI**: Common Gateway Interface
  - Standard way for web server executing a program taking user data as input
Web: Multi-tiered Architectures (2)

- **Server-side script (JavaScript/NodeJS, PHP, Python, Go)**
  - Executed by server when document has been fetched locally
  - Result of executing script (not the script itself) is sent along with the rest of the document to the client

- **Client-side script (JavaScript)**
  - For dynamic generation of web pages at the client
  - Script interpreted and executed by JS engine typically built into the web browser

- **Applets (e.g., Java applets, ActiveX controls)**
  - Pre-compiled stand-alone applications sent to client and executed in browser’s address space

- **Servlets (e.g., JSP, ASP)**
  - Pre-compiled program executed in server’s address space

- **Helper applications**
  - Independent applications assisting browser in displaying documents

- **Plug-in**
  - Small program dynamically loaded from local file system browser for handling specific document type

- **Web proxy**
  - Originally, support to handle application-level protocols
  - Now, a cache shared by multiple browsers for enhanced performance
Web: Multi-tiered Architectures (2)

- **Server-side script** (JavaScript/NodeJS, PHP, Python, Go)
  - Executed by server when document has been fetched locally
  - Result of executing script (not the script itself) is sent along with the rest of the document to the client

- **Client-side script** (JavaScript)
  - For dynamic generation of web pages at the client
  - Script interpreted and executed by JS engine typically built into the web browser

- **Applets** (e.g., Java applets, ActiveX controls)
  - Pre-compiled stand-alone applications sent to client and executed in browser's address space

- **Servlets** (e.g., JSP, ASP)
  - Pre-compiled program executed in server’s address space

- **Helper applications**
  - Independent applications assisting browser in displaying documents

- **Plug-in**
  - Small program dynamically loaded from local file system browser for handling specific document type

- **Web proxy**
  - Originally, support to handle application-level protocols
  - Now, a cache shared by multiple browsers for enhanced performance