Architectural styles

Basic idea

A style is formulated in terms of

- (replaceable) components with well-defined interfaces
- the way that components are connected to each other
- the data exchanged between components
- how these components and connectors are jointly configured into a system.

Connector

A mechanism that mediates communication, coordination, or cooperation among components. Example: facilities for (remote) procedure call, messaging, or streaming.
Layered architecture

Different layered organizations

(a) Request/Response downcall

Layer N
Layer N-1
Layer 2
Layer 1

(b) One-way call

Layer N
Layer N-1
Layer N-3
Layer N-2

(c) Upcall

Layer N
Layer N-1
Layer N-2
Handle
Upcall
Example: communication protocols

Protocol, service, interface

Layered communication protocols
Application Layering

Traditional three-layered view

- **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

This layering is found in many distributed information systems, using traditional database technology and accompanying applications.
Application Layering

Traditional three-layered view

- **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

Observation

This layering is found in many distributed information systems, using traditional database technology and accompanying applications.
**Application Layering**

**Example: a simple search engine**

- **User interface**
- **Query generator**
- **Ranking algorithm**
- **HTML generator**

**Layers:**
- **User-interface level**
- **Processing level**
- **Data level**

- Data level:
  - Database with Web pages
  - Database queries
  - Web page titles with meta-information
  - Ranked list of page titles

- Processing level:
  - Ranking algorithm
  - HTML page containing list

- User-interface level:
  - User interface
  - Keyword expression

**Diagram:**
- User interface receives keyword expression and sends it to the query generator.
- The query generator generates database queries.
- The ranking algorithm processes the database queries and generates a list of ranked page titles.
- The HTML generator generates the HTML page containing the list of page titles.

**Diagram Notes:**
- **Data level**
- **Processing level**
- **User-interface level**
Object-based style

Essence
Components are objects, connected to each other through procedure calls. Objects may be placed on different machines; calls can thus execute across a network.

Encapsulation
Objects are said to encapsulate data and offer methods on that data without revealing the internal implementation.
RESTful architectures

Essence

View a distributed system as a collection of resources, individually managed by components. Resources may be added, removed, retrieved, and modified by (remote) applications.

1. Resources are identified through a single naming scheme
2. All services offer the same interface
3. Messages sent to or from a service are fully self-described
4. 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>
Example: Amazon’s Simple Storage Service

Essence

Objects (i.e., files) are placed into buckets (i.e., directories). Buckets cannot be placed into buckets. Operations on ObjectName in bucket BucketName require the following identifier:

http://BucketName.s3.amazonaws.com/ObjectName

Typical operations

All operations are carried out by sending HTTP requests:

- Create a bucket/object: PUT, along with the URI
- Listing objects: GET on a bucket name
- Reading an object: GET on a full URI
Many people like RESTful approaches because the interface to a service is so simple. The catch is that much needs to be done in the parameter space.

### Amazon S3 SOAP interface

<table>
<thead>
<tr>
<th>Bucket operations</th>
<th>Object operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListAllMyBuckets</td>
<td>PutObjectInline</td>
</tr>
<tr>
<td>CreateBucket</td>
<td>PutObject</td>
</tr>
<tr>
<td>DeleteBucket</td>
<td>CopyObject</td>
</tr>
<tr>
<td>ListBucket</td>
<td>GetObject</td>
</tr>
<tr>
<td>GetBucketAccessControlPolicy</td>
<td>GetObjectExtended</td>
</tr>
<tr>
<td>SetBucketAccessControlPolicy</td>
<td>DeleteObject</td>
</tr>
<tr>
<td>GetBucketLoggingStatus</td>
<td>GetObjectAccessControlPolicy</td>
</tr>
<tr>
<td>SetBucketLoggingStatus</td>
<td>SetObjectAccessControlPolicy</td>
</tr>
</tbody>
</table>
On interfaces

Simplifications

Assume an interface `bucket` offering an operation `create`, requiring an input string such as `mybucket`, for creating a bucket “mybucket.”

SOAP

```python
import bucket
bucket.create("mybucket")
```

RESTful

```python
PUT "http://mybucket.s3.amazonsws.com/"
```
On interfaces

Simplifications

Assume an interface `bucket` offering an operation `create`, requiring an input string such as `mybucket`, for creating a bucket “mybucket.”

SOAP

```python
import bucket
bucket.create("mybucket")
```
Simplifications

Assume an interface `bucket` offering an operation `create`, requiring an input string such as `mybucket`, for creating a bucket “mybucket.”

**SOAP**

```java
import bucket
bucket.create("mybucket")
```

**RESTful**

```
PUT "http://mybucket.s3.amazonaws.com/"
```
Coordination

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 and Shared data space

Component ➔ Subscribe ➔ Event bus ➔ Publish ➔ Component

Component ➔ Notification delivery ➔ Component

Component ➔ Publish ➔ Shared (persistent) data space ➔ Subscribe ➔ Component

Component ➔ Data delivery ➔ Component
Using legacy to build middleware

Problem
The interfaces offered by a legacy component are most likely not suitable for all applications.

Solution
A wrapper or adapter offers an interface acceptable to a client application. Its functions are transformed into those available at the component.
Organizing wrappers

Two solutions: 1-on-1 or through a broker

Complexity with $N$ applications

- **1-on-1**: requires $N \times (N - 1) = \mathcal{O}(N^2)$ wrappers
- **broker**: requires $2N = \mathcal{O}(N)$ wrappers
Developing adaptable middleware

Problem
Middleware contains solutions that are good for most applications ⇒ you may want to adapt its behavior for specific applications.
Intercept the usual flow of control

- Request-level interceptor
- Message-level interceptor
- Interceptors
- Interceptor organization
- Interceptors

Client application

B.doit(val)

Intercepted call

Application stub

invoke(B, &doit, val)

Nonintercepted call

Object middleware

send(B, "doit", val)

Local OS

To object B
Centralized system architectures

Basic Client–Server Model

Characteristics:
- There are processes offering services (servers)
- There are processes that use services (clients)
- Clients and servers can be on different machines
- Clients follow request/reply model with respect to using services

Simple client-server architecture
Multi-tiered centralized system architectures

Some traditional organizations

- **Single-tiered**: dumb terminal/mainframe configuration
- **Two-tiered**: client/single server configuration
- **Three-tiered**: each layer on separate machine

Traditional two-tiered configurations
Being client and server at the same time

Three-tiered architecture
Alternative organizations

**Vertical distribution**

Comes from dividing distributed applications into three logical layers, and running the components from each layer on a different server (machine).

**Horizontal distribution**

A client or server may be physically split up into logically equivalent parts, but each part is operating on its own share of the complete data set.

**Peer-to-peer architectures**

Processes are all equal: the functions that need to be carried out are represented by every process ⇒ each process will act as a client and a server at the same time (i.e., acting as a *servant*).
Structured P2P

Essence

Make use of a **semantic-free index**: each data item is uniquely associated with a key, in turn used as an index. Common practice: use a **hash function**

$$\text{key(} \text{data item} \text{)} = \text{hash(} \text{data item's value} \text{)}.$$ 

P2P system now responsible for storing (key, value) pairs.

Simple example: hypercube

Looking up $d$ with key $k \in \{0, 1, 2, \ldots, 2^4 - 1\}$ means routing request to node with identifier $k$. 

Example: Chord

**Principle**
- Nodes are logically organized in a ring. Each node has an $m$-bit identifier.
- Each data item is hashed to an $m$-bit key.
- Data item with key $k$ is stored at node with smallest identifier $id \geq k$, called the successor of key $k$.
- The ring is extended with various shortcut links to other nodes.
Example: Chord

*lookup*(3)@9: 28 → 1 → 4
Unstructured P2P

Essence

Each node maintains an ad hoc list of neighbors. The resulting overlay resembles a random graph: an edge $\langle u, v \rangle$ exists only with a certain probability $P[\langle u, v \rangle]$.

Searching

- **Flooding**: issuing node $u$ passes request for $d$ to all neighbors. Request is ignored when receiving node had seen it before. Otherwise, $v$ searches locally for $d$ (recursively). May be limited by a Time-To-Live: a maximum number of hops.

- **Random walk**: issuing node $u$ passes request for $d$ to randomly chosen neighbor, $v$. If $v$ does not have $d$, it forwards request to one of its randomly chosen neighbors, and so on.
Super-peer networks

Essence

It is sometimes sensible to break the symmetry in pure peer-to-peer networks:

- When searching in unstructured P2P systems, having index servers improves performance
- Deciding where to store data can often be done more efficiently through brokers.
Edge-server architecture

Essence
Systems deployed on the Internet where servers are placed at the edge of the network: the boundary between enterprise networks and the actual Internet.
Collaboration: The BitTorrent case

Principle: search for a file $F$

- Lookup file at a global directory $\Rightarrow$ returns a torrent file
- Torrent file contains reference to tracker: a server keeping an accurate account of active nodes that have (chunks of) $F$.
- $P$ can join swarm, get a chunk for free, and then trade a copy of that chunk for another one with a peer $Q$ also in the swarm.