Lecture 02: Architectures

Last update: September 9, 2019
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-2
Layer N-3

(c) Upcall

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

Protocol, service, interface

Layered architectures
Two-party communication

**Server**

```python
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept()  # returns new socket and addr. client
4 while True:  # forever
5     data = conn.recv(1024)  # receive data from client
6     if not data: break  # stop if client stopped
7     conn.send(str(data)+"*")  # return sent data plus an "*"
8     conn.close()  # close the connection
```

**Client**

```python
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT))  # connect to server (block until accepted)
4 s.send('Hello, world')  # send some data
5 data = s.recv(1024)  # receive the response
6 print data  # print the result
7 s.close()  # close the connection
```
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
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
- Database with Web pages
- Database queries
- HTML generator
- Ranking algorithm
- HTML page containing list
- Ranked list of page titles
- Web page titles with meta-information

Layered architectures

Architectures: Architectural styles
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
On interfaces

Issue

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.”
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")
```
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

```http
PUT "http://mybucket.s3.amazonaws.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")
```

RESTful

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

Conclusions

Are there any to draw?
Coordination

Temporal and referential coupling

<table>
<thead>
<tr>
<th>Referentially coupled</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

- **Event bus**
  - Subscribe
  - Publish
  - Notification delivery

- **Shared (persistent) data space**
  - Publish
  - Subscribe
  - Data delivery
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.
Interceptors

Interceptors can be classified into two main types:

1. **Request-level interceptron**
2. **Message-level interceptron**

In the context of middleware organization:

- **Client application**
- **Application stub**
- **Object middleware**
- **Local OS**
- **To object B**

**Intercepted call**

```
B.doit(val)
```

**Nonintercepted call**

```
invoke(B, &doit, val)
send(B, "doit", val)
```
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
Some traditional organizations

- **Single-tiered**: straightforward 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

- Client
- Application server
- Database server

Request operation → Request data
Wait for reply → Wait for data
Return data → Return reply
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**

\[
key(data\ item) = hash(data\ item's\ value).
\]

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.
Flooding versus random walk

Model

Assume $N$ nodes and that each data item is replicated across $r$ randomly chosen nodes.

Random walk

$\mathbb{P}[k]$ probability that item is found after $k$ attempts:

$$
\mathbb{P}[k] = \frac{r}{N} (1 - \frac{r}{N})^{k-1}.
$$

$S$ (“search size”) is expected number of nodes that need to be probed:

$$
S = \sum_{k=1}^{N} k \cdot \mathbb{P}[k] = \sum_{k=1}^{N} k \cdot \frac{r}{N} (1 - \frac{r}{N})^{k-1} \approx \frac{N}{r} \text{ for } 1 \ll r \leq N.
$$
Flooding versus random walk

Flooding

- Flood to $d$ randomly chosen neighbors
- After $k$ steps, some $R(k) = d \cdot (d - 1)^{k-1}$ will have been reached (assuming $k$ is small).
- With fraction $r/N$ nodes having data, if $\frac{r}{N} \cdot R(k) \geq 1$, we will have found the data item.

Comparison

- If $r/N = 0.001$, then $S \approx 1000$
- With flooding and $d = 10, k = 4$, we contact 7290 nodes.
- Random walks are more communication efficient, but might take longer before they find the result.
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.
Skype’s principle operation: \( A \) wants to contact \( B \)

### Both \( A \) and \( B \) are on the public Internet
- A TCP connection is set up between \( A \) and \( B \) for control packets.
- The actual call takes place using UDP packets between negotiated ports.

### \( A \) operates behind a firewall, while \( B \) is on the public Internet
- \( A \) sets up a TCP connection (for control packets) to a super peer \( S \)
- \( S \) sets up a TCP connection (for relaying control packets) to \( B \)
- The actual call takes place through UDP and directly between \( A \) and \( B \)

### Both \( A \) and \( B \) operate behind a firewall
- \( A \) connects to an online super peer \( S \) through TCP
- \( S \) sets up TCP connection to \( B \).
- For the actual call, another super peer is contacted to act as a relay \( R \): \( A \) sets up a connection to \( R \), and so will \( B \).
- All voice traffic is forwarded over the two TCP connections, and through \( R \).
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.
BitTorrent under the hood

Some essential details

- A tracker for file $F$ returns the set of its downloading processes: the current **swarm**.
- A communicates only with a subset of the swarm: the **neighbor set** $N_A$.
- If $B \in N_A$ then also $A \in N_B$.
- Neighbor sets are regularly updated by the tracker.

Exchange blocks

- A file is divided into equally sized **pieces** (typically each being 256 KB).
- Peers exchange **blocks** of pieces, typically some 16 KB.
- $A$ can upload a block $d$ of piece $D$, only if it has piece $D$.
- Neighbor $B$ belongs to the **potential set** $P_A$ of $A$, if $B$ has a block that $A$ needs.
- If $B \in P_A$ and $A \in P_B$: $A$ and $B$ are in a position that they can **trade** a block.
BitTorrent phases

Bootstrap phase

A has just received its first piece (through optimistic unchoking: a node from $N_A$ unselfishly provides the blocks of a piece to get a newly arrived node started).

Trading phase

$|P_A| > 0$: there is (in principle) always a peer with whom $A$ can trade.

Last download phase

$|P_A| = 0$: $A$ is dependent on newly arriving peers in $N_A$ in order to get the last missing pieces. $N_A$ can change only through the tracker.
BitTorrent phases

Development of $|P|$ relative to $|N|$.

![Graph showing the development of $|P|$ relative to $|N|$ for different values of $|N|$ (5, 10, 40).]