LTAT.06.007 Distributed Systems
Lecture 2 – Systems architectures

Huber Flores, PhD
ASSOCIATE PROFESSOR

Tartu, Estonia 18/02/2020
Recap

- Studied concepts of distributed systems
  - Definitions
  - Types
  - Properties
  - Purpose
Agenda

• Understanding different system architectures
• General organization
  • Components, interactions and roles
# Architecture

## Definition
- Determines the organization of a distributed system
  - Interaction
  - Behavior

## Characteristics
- Software architecture
  - Logical organization and interaction of software components
- System architecture
  - Instantiation of a software architecture on real machines
# Architecture styles

## Definition

The notion of an architectural style

- Formulated in terms of components, their connections and the data exchanged between them
- A **component** is a modular unit with well-defined required and provided interfaces, replaceable within its environment
- A **connector** is a mechanism mediating communication, collaboration, coordination or cooperation among components
## Architecture styles

**Basic idea**

- (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 example**

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

Important architectural styles for distributed systems

• Layered architectures
• Object-based architectures
• Data-centered architectures
• Event-based architectures
Layered style

Observations

- 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 from layer to layer:
  - Requests go down
  - Results flow upward
- Widely adopted in networking
Layered style

Different layered organizations

- Layered networking architectures

  Hybrid Internet protocol stack

<table>
<thead>
<tr>
<th>Layer</th>
<th>OSI</th>
<th>TCP/IP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Physical</td>
<td>Host-to-network</td>
</tr>
<tr>
<td>2</td>
<td>Data link</td>
<td>Data link</td>
</tr>
<tr>
<td>3</td>
<td>Network</td>
<td>Network</td>
</tr>
<tr>
<td>4</td>
<td>Transport</td>
<td>Transport</td>
</tr>
<tr>
<td>5</td>
<td>Session</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Presentation</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Application</td>
<td>Application</td>
</tr>
</tbody>
</table>

- Layering allows mastering the complexity
  - Explicit structure allows identification, relationship of complex system’s pieces
  - Modularization eases maintenance, updating of system
  - Change of implementation of a layer’s service transparent to the rest of system
Example: communication protocols

Protocol, service, interface

Diagram:

- Layer N
- Interface
- Service
- Protocol
- Party A
- Party B
Example: two-party communication

Server

```
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 received data plus an "*"
8 conn.close()  # close the connection
```

Client

```
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
```
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.

Observation
• Objects correspond to components
• Components are connected via a (remote) procedure call mechanism
  • RMI, RPC
Object-based style

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

**Observation**
- Objects are serialized when transmitted through the network
- Java reflection is a good example of object transferred between two different parties
Resource-centered style

**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.

- Resources are identified through a single naming scheme
  - REST, SOA
- 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
### Resource-centered style

#### Basic operations over resources

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PUT</strong></td>
<td>Create a new resource</td>
</tr>
<tr>
<td><strong>GET</strong></td>
<td>Retrieve the state of a resource in some representation</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>Delete a resource</td>
</tr>
<tr>
<td><strong>POST</strong></td>
<td>Modify a resource by transferring a new state</td>
</tr>
</tbody>
</table>
Example: Amazon’s Simple Storage Service

<table>
<thead>
<tr>
<th>Essence</th>
</tr>
</thead>
<tbody>
<tr>
<td>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:</td>
</tr>
<tr>
<td><a href="http://BucketName.s3.amazonaws.com/ObjectName">http://BucketName.s3.amazonaws.com/ObjectName</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Typical operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>All operations are carried out by sending HTTP requests:</td>
</tr>
<tr>
<td>• Create a bucket/object: <strong>PUT</strong>, along with the URI</td>
</tr>
<tr>
<td>• Listing objects: <strong>GET</strong> on a bucket name</td>
</tr>
<tr>
<td>• Reading an object: <strong>GET</strong> on a full URI</td>
</tr>
</tbody>
</table>
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.

Example: Amazon S3 SOAP interface

<table>
<thead>
<tr>
<th>Bucket operations</th>
<th>Object operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListAllMyBuckets</td>
<td>PutObject Inline</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.”

<table>
<thead>
<tr>
<th>SOAP</th>
<th>RESTful</th>
</tr>
</thead>
</table>
| import bucket  
bucket.create("mybucket") | PUT "http://mybucket.s3.amazonaws.com/" |
Event-based style

**Essence**
Processes/Component/Objects communicate through propagation of events

**Observations**
- Events can optionally carry data
- Publish/subscribe systems
  - Processes publish events
  - Only processes having subscribed to particular events will receive them
- Allows loose coupling of processes
  - Processes need not explicitly refer to each other (referential decoupling)
Shared data spaces

Observations

• 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

### Types

- Centralized
- Decentralized
## Comparison (Junginger & Lee, 2004)

<table>
<thead>
<tr>
<th>Feature</th>
<th>Peer-to-peer</th>
<th>Centralized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scalability</td>
<td>High</td>
<td>Limited</td>
</tr>
<tr>
<td>Resource availability</td>
<td>High</td>
<td>Limited</td>
</tr>
<tr>
<td>Fault tolerance</td>
<td>High</td>
<td>Limited</td>
</tr>
<tr>
<td>Infrastructure</td>
<td>Self-organizing</td>
<td>Needs setup and administration</td>
</tr>
<tr>
<td>Infrastructure costs</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Storage of global data</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Control</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Trusted</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Enterprise/legacy system integration</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
System architectures

Combination

• Hybrid architectures
  • Centralized and decentralized
Centralized system architectures

Essence

- Client-Server model
  - **Server** (process) implements a specific service
  - **Client** (process) request a service from a server by sending a request and waiting for a reply

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
Centralized system architectures

Behavior

• Request-reply behavior
  • Call semantics and transmission failure
  • Ideally: exactly-once
  • Zero-or-more (“maybe”): service may or may have not 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 (repeatable) operation can be repeated multiple times without harm
Application layering

Example

The general organization of an Internet search engine
Observations

- **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 user interfaces

- **Processing level**
  - Contains core functionality of an application

- **Data level**
  - Contains programs that maintain the data on which the applications operate
  - Persistency
  - Consistency
Multitiered architecture

Essence

• Simplest organization is to have only two types of machines
  • A client machine containing only the programs implementing (part of) the user-interface level
  • A server machine containing the rest (programs implementing the processing and data level)
Multitiered architecture

Traditional two-tiered configurations

- Fat clients ((d)-(e)) vs thin clients ((a)-(c))
Multitiered architecture

Being client and server at the same time

- Defined as three-tiered architecture
  - Single server is replaced by multiple servers running on distributed machines
  - Server sometimes acts as a client (concept as old as the existence of the first computer)
## Multitiered architecture

### Alternative configurations

- **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.
Decentralized architectures

**Essence**
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 **servent**).

**Example**
Communication protocols enable the components for decentralized architectures.
- Bluetooth
- WiFi-Direct
Decentralized architectures

Observations

- Peer-to-peer systems
  - Horizontal distribution configurations

- Representation of peer-to-peer architectures using overlay networks
  - Nodes represent processes
  - Links represent communication channels

- Structured vs unstructured peer-to-peer architectures
Structured peer-to-peer architectures

**Essence**

- Overlay network is constructed using a deterministic procedure.
- 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.

\[
\text{key(data item)} = \text{hash(data item’s value)}
\]

- P2P system now responsible for storing (key,value) pairs.
Structured peer-to-peer architectures

Approaches

DHT (Distributed Hash Table) is the most-used procedure
• Data items are assigned a random key from a large identifier space
• Nodes are assigned a random number from the same space
• Efficient and deterministic scheme uniquely mapping the key of a data item to the identifier of a node using some distance metric
• When looking up a data item, the network address of the node responsible for that data item is returned
• Many DHT variations (e.g. Chord, CAN, Pastry, Bamboo, Tapestry, Kademlia)

Example: Hypercube

Looking up d with key \( k \in (0,1,2,\ldots,2^4-1) \) means routing request to node with identifier \( k \).
Structured peer-to-peer architectures

Example: Chord

- 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 $id > k$, called the successor of key $k$.
- The ring is extended with various shortcut links to other nodes.
Structured peer-to-peer architectures

Example: Chord

![Chord diagram]

- Actual node
- Associated data keys
Structured peer-to-peer architectures

Example: CAN

- CAN (Content Addressable Network)
- d-dimensional Cartesian coordinate space is completely partitioned among nodes
- Each data item is assigned a unique point in this space (corresponding node is responsible for the data item)
Structured peer-to-peer architectures

Example: CAN

Mapping of data items onto nodes

Splitting a region when a node joins
Structured peer-to-peer architectures

DHT Comparison (Hautakorpi & Camarillo 2007)

<table>
<thead>
<tr>
<th></th>
<th>Chord</th>
<th>CAN</th>
<th>Pastry</th>
<th>Bamboo</th>
<th>Tapestry</th>
<th>Kademlia</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel lookups</td>
<td>not suitable</td>
<td>no (can benefit)</td>
<td>not suitable</td>
<td>yes (on I)</td>
<td>no (can benefit)</td>
<td>yes</td>
</tr>
<tr>
<td>Proximity support</td>
<td>per-hop (not on I)</td>
<td>landmark ord.</td>
<td>yes, from others</td>
<td>yes</td>
<td>essentially no</td>
<td>no</td>
</tr>
<tr>
<td>Graceful departure</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Replication &amp; caching</td>
<td>basic support</td>
<td>versatile</td>
<td>fairly good</td>
<td>no default method</td>
<td>no default method</td>
<td>good</td>
</tr>
<tr>
<td>Complexity</td>
<td>simple</td>
<td>simple</td>
<td>quite complex</td>
<td>quite complex</td>
<td>quite complex</td>
<td>simple</td>
</tr>
<tr>
<td>Bandwidth consumption</td>
<td>moderate</td>
<td>moderate</td>
<td>high</td>
<td>moderate (constant)</td>
<td>quite high</td>
<td>moderate</td>
</tr>
<tr>
<td>Node join &amp; departure</td>
<td>quite simple</td>
<td>very simple</td>
<td>complex join</td>
<td>quite simple</td>
<td>complex join</td>
<td>simple</td>
</tr>
<tr>
<td>Configuration parameters</td>
<td>a few</td>
<td>many</td>
<td>some</td>
<td>some</td>
<td>a small affect</td>
<td>a few</td>
</tr>
<tr>
<td>Extendability</td>
<td>quite good</td>
<td>rich already</td>
<td>quite good</td>
<td>quite good</td>
<td>quite good</td>
<td>quite good</td>
</tr>
<tr>
<td>Notification framework</td>
<td>no</td>
<td>no</td>
<td>already exists</td>
<td>use Pastry’s</td>
<td>use Pastry’s</td>
<td>no</td>
</tr>
</tbody>
</table>
Unstructured peer-to-peer architectures

Essence

Overlay network is constructed using randomized algorithms, resulting in a random graph

- Each node maintains a list of neighbors (partial view), which is constructed in random way
- Data items are placed randomly on nodes

The resulting overlay resembles a random graph: an edge \((u,v)\) exists only with a certain probability \(P[(u,v)]\).
Unstructured peer-to-peer architectures

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
P[k] probability that item is found after k attempts:

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

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

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

Flooding

• Flood to \( d \) randomly chosen neighbors
• After \( k \) steps, some \( R(k) \) will have been reached (assuming \( k \) is small).

\[
R(k) = d \cdot (d - 1)^{k-1}
\]

• With fraction \( r/N \) nodes having data, if \( r/N \times (R(k)) > 1 \), 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.
Super-peer networks

<table>
<thead>
<tr>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Scalability problems with flat peer-to-peer structures</td>
</tr>
<tr>
<td>• Superpeer ~ special node maintaining an index of data items or acting as a broker</td>
</tr>
<tr>
<td>• Long-lived processes with high availability</td>
</tr>
<tr>
<td>• Hierarchical organization of nodes into superpeer network</td>
</tr>
<tr>
<td>• Superpeers are often organized in a peer-to-peer network</td>
</tr>
<tr>
<td>• Regular peer is connected as a client to a superpeer (fixed relationship)</td>
</tr>
<tr>
<td>• Flexible means for nodes to join and leave the network</td>
</tr>
<tr>
<td>• <strong>Leader-election problem:</strong> how to select the nodes eligible to become superpeers?</td>
</tr>
</tbody>
</table>
Hybrid architectures

**Essence**

Client-server solutions are combined with decentralized architectures

**Example: Edge server architecture**

- Clients connect to Internet via edge servers
- Motivation: reduced response times, load balancing, scalability
- E.g. web proxy (content-blind vs content-aware cache), CDN (content distribution network) server
Hybrid architectures

Collaborative distributed systems (e.g., BitTorrent)

• File is divided into chunks stored in different nodes
• Global directory holds reference to .torrent metadata file
• Tracker keeps track of nodes having chunks of the file

![Diagram of BitTorrent network]
Architectures versus middleware

Essence

Tradeoff between
- The distribution transparency provided by a middleware following a particular architectural style
- Should simplify designing applications

And
- Application requirements
- Middleware may no longer adapt for given application

Mechanisms for modifying the behavior of middleware wrt. Application requirements
- Interceptors
- Adaptive software
Interceptors

Essence
Break the normal flow of control and allow other (application specific) code to be executed

Types

• Request-level interceptor
  • E.g. make call for each replica of object B

• Message-level interceptor
  • E.g. assist in transferring invocation to target object
Summary

- System architectures
  - Types
  - Objectives
  - Characteristics
  - Components
  - Roles

- Behavior and organization
  - Localization, e.g., overlay networks
  - Operations (Default / Possible)
Next lecture

Processes
Questions?

E-mail: huber.flores@ut.ee