Software Architecture

Leftovers from last week.

Dr. Philipp Leitner

@xLeitix

University of Zurich, Switzerland
(Some) Common Architectural Styles

Program structure

- Layers
- Client / Server
- Peer-to-Peer
- MVC

Data Flow

- Pipes and Filters
- Event-Driven Architecture
- Blackboard (Factory / Worker)
Filters are the components that read an input data stream and transform it into an output data stream.

Pipes are the connectors that provide the output of a filter as input to another filter.
Pipes and Filters: Example

```bash
find . -type f -print | sort | uniq
```
Pipes and Filters: Advantages

• **Simple**: no complex component interactions
• Filters as black-box; substitutable
• High maintainability and reusability of individual components
Pipes and Filters: Disadvantages

• Whether separation into processing steps is feasible strongly depends on application domain and problem

• Filters require a common data format

• Redundancy in parsing/unparsing

• Process overhead
Event-driven Architecture (EDA)

Components are event emitters (publishers, agents) or consumers (subscribers, sinks)

An event dispatcher distributes events to subscribers

→ fundamentally asynchronous
Java Swing
(or more generally - most GUI frameworks)
EDA: Advantages

- Components are loosely coupled
- High (runtime) extensibility and reusability; components can be easily exchanged
- Especially apt to unpredictable and asynchronous environments
- Often “feels” fast and responsive
EDA: Disadvantages

• No guarantees regarding execution or order of event processing
• Data exchange other than with events problematic
• Component behavior is tightly coupled with execution environment
• Difficult to “grok”
Blackboard

Two kinds of components:

- Central data management
- Independent components for computation (knowledge sources)

Activation of computation:

- Database trigger
- Actual state
Blackboard: Advantages

• High changeability and maintainability
• Reusable knowledge sources
• **Factory/Worker:** support for fault-tolerance, robustness and redundancy because of loose coupling of workers
Blackboard: Disadvantages

• Hard to test
• Difficult to establish good control strategy
• Low efficiency
Distributed Systems Architectures

Common Architectural Styles and Patterns.

Dr. Philipp Leitner

@xLeitix

University of Zurich, Switzerland
Motivation

http://www.ejbtutorial.com/distributed-systems/challenges-for-a-distributed-system
Motivation

Client-Server Systems

Web browser → Internet → Web server machine

Web server software

Launches and proxies requests to

Ruby on Rails FastCGI process

Peer-to-Peer Systems

Internet

Super Node

Login Server

Skype Client

Figure 1: Skype Network

Implementation of Connectors?
Hierarchy of Styles of Distributed Interaction

- Service-Oriented Systems (SOAP / REST)
- Component-Based Systems (Enterprise Java Beans)
- Distributed Object Systems (.NET Remoting)
- Remote Procedure Call Systems (XML-RPC)
- Message Passing (Java Message Service)
- Streaming (Sockets)
Streaming

- Connector is a (semi-)permanent communication channel between client and server
- Variants:
  - Unidirectional versus bidirectional
  - Unicast versus multicast
Variants

Unidirectional

Server 1 * Client

Unicast

Server 1 1 Client

Bidirectional

Server 1 * Client

Multicast

Server 1 * Client
Example in Ruby

Server

```ruby
require 'socket'

# server

server = TCPServer.open(8080)
loop {
  Thread.start(server.accept) do |client|
    client.puts(Time.now.ctime)
    client.puts "Closing the connection. Bye!"
    client.close
  end
}
```

Client

```ruby
require 'socket'

# client

s = TCPSocket.open("localhost", 8080)
while line = s.gets
  puts line.chomp
end
s.close
```
Streaming: Advantages

- Low overhead
- Use cases that require continuous data transfer (e.g., VoIP) easy to implement
- No leaky abstraction
Streaming: Disadvantages

- Implementation maybe messy and verbose
- No built-in mechanisms for confirmation, resending, etc.
- Client and server need common understanding of data and protocols
Message Passing

Queues (channels) connect programs and transmit messages

A sender (or producer) is a program that sends a message by writing it to a queue

A receiver (or consumer) is a program that receives a message by reading it from a queue
Messaging: Advantages

Referential decoupling
Producers and consumers do not need to know each other
Producers and consumers can come and go dynamically

Temporal decoupling
Producers and consumers do not need to be online at the same time (if the queue stores messages)
Consumers can process messages at their own pace

Inherently asynchronous model
The store-and-forward processing model of message queues lends itself very well to async communication
Messaging: Disadvantages

Complex programming model
- Message sequencing (out-of-order messages)
- Message loss
- Message duplication
- Message congestion

Performance overhead
- Additional hop in communication (message queue)
- Message queue potentially becomes performance bottleneck
Remote Procedure Calls

Make the invocation of remote code (more or less) seem like local calls

Hide remoting as far as possible (abstraction!)

```java
ServerStub myServer = (ServerStub) Naming.lookup("//localhost/MyServer");
String response = myServer.getHelloWorldMessage();
System.out.println(response);
```
“All non-trivial abstractions are leaky”

http://www.joelonsoftware.com/articles/LeakyAbstractions.html
Remote Procedure Calls

Main interaction protocol:

- (Blocking) request/response style

```
Client

| calculateRevenue() |

Server

| return |
```
Implementation using Message Passing

Message(req):
{op : calculateRevenue,
param : 16 }

Message(resp):
{status : success,
return : 1466.17 }
Simplest RPC-style

Server provides (independent) procedures that can be called
Example in Groovy and XML-RPC

**Server**

```groovy
def server = new XMLRPCServer()

// define operations
server.calculateRevenue = { contractNr ->
  // ... do stuff
}

def serverSocket = new ServerSocket(8080)
server.startServer(serverSocket)
```

**Client**

```groovy
def serverProxy = new XMLRPCServerProxy("http://localhost:8080")
serverProxy.calculateRevenue(contract)
```

**Wire**

```xml
<methodCall
  <methodName>calculateRevenue</methodName>
  <params>
    <param>
      <value><i4>156</i4></value>
    </param>
  </params>
</methodCall>
```
Distributed Objects

Natural progression for object-oriented programming languages

• Imperative language work well with RPC

• OO language work well with distributed objects

Server provides objects (data and methods) that clients can interact with
On State

**Stateless Interaction**
(server can be a pure function)

- Client
  - `calculateRevenue(obj)`
  - `return(3478.18)`

(state is maintained by client or in a database, every request can be served by different server instances)

**Stateful Interaction**

- Server
  - `calculateRevenue()`
  - `return(3478.18)`

- Client
  - `setObject(obj)`

(state is maintained by server, subsequent requests need to be routed to same server, requires session)
Distributed Objects

Client Host

Client

Server Host

Server
- List<Contract> contracts
+ void addNewContract(Contract contract)
+ long calculateRevenue(Contract contract)
Example in C# and .NET Remoting

Server

TcpServerChannel channel = new TcpServerChannel(8080);
ChannelServices.RegisterChannel(channel);
WellKnownServiceTypeEntry remObj = new
    WellKnownServiceTypeEntry(typeof(IContractRemoteObj),
        "Contract", WellKnownObjectMode.SingleCall);
RemotingConfiguration.RegisterWellKnownServiceType(remObj);

Client

TcpClientChannel channel = new TcpClientChannel();
ChannelServices.RegisterChannel(channel);
IContractRemoteObj remote_object =
    (IContractRemoteObj)Activator.GetObject(
        typeof(IContractRemoteObj),
        "tcp://localhost:9000/ContractRemoteObj"
    );
remote_object.AddNewContract(contract);
Component-Based Systems

Component:

- Reusable collection of remote objects
- All of which may be stateful or stateless
Service-Based Systems

Applications are built as **composition** of services (lego principle)
Example: Netflix
Service Characteristics

Core characteristics of services:

- Standardised interface
- Replaceable
- Context-free (and hence reusable)
- Stateless
- (Observable Quality-of-Service)
Standardised Interface

Service has a “contract” that defines what the service does, and how it can be used (the API)

Usually assumed to be programming language and environment independent

Can be explicitly specified
(e.g., WSDL)

Or implicit
(e.g., REST)

Eases Integration!
Many different services could (potentially) implement a given contract.

- Maybe different copies of the same service
- Maybe functionally different implementations

Your application should be able to switch between those services at will.

Eases Maintenance, allows for failover and redundancy!
Services should not know or care about the context they are used in.

Example:

- A service that manages products can be used in the context of ordering, or for a recommender service, or for inventory management.

Fosters reuse!
Stateless

State is maintained on the client side, the service does not know more about the interaction than what is necessary to serve the current request.

Eases horizontal scaling!
Common Ways to Implement Services

SOAP / WSDL

• Standardised, heavy-weight
• Mostly used in enterprise context

REST

• Light-weight, thin layer on top of HTTP
• Commonly used for Web APIs

• Others:
  • OSGi, SCA, Jini, …
REST

Acronym for “Representational State Transfer”

- Client manages the state
- And “transfers” the state to the server with every request
Five Main REST Characteristics

- **Resource-Orientation**
  - Key elements of any RESTful service are resources

- **Naming**
  - Every resource is associated with an unique and descriptive name

- **Uniform Interface**
  - Every resource is accessed through the same interface

- **Statelessness**
  - Every request is self-contained

- **Layering**
  - Intermediaries can be inserted transparently
  - Proxies, caches, ...
An Example Interaction

Client

get: /products
{prod1, prod2, … }

get: /products/prod1
{id: 1, name: XXX}

put: /products/prod1
{id: 1, name: YYY}

Server

Products
prod1
prod2

Orders
GET /products HTTP/1.1
Host: my.remote.server
Accept: application/json

HTTP/1.1 200 OK
Date: Tue, 02 Nov 2016 14:38:00 CET
Content-Type: application/json

{[prod1, prod2, prod3, prod4, prod5]}
“(...) the principle is that the hypermedia in each server response will contain links that correspond to all the actions that the client can currently perform. (...) every server response describes the new actions that are available. (...) A client of a RESTful application need only know a single fixed URL to access it. All future actions should be discoverable dynamically from hypermedia links included in the representations of the resources that are returned from that URL. (...)”

[Wikipedia]
## Uniform Interface: HTTP

<table>
<thead>
<tr>
<th>Verb</th>
<th>Safe</th>
<th>Idempotent</th>
<th>Body</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td><strong>GET</strong> a resource identified by an URL</td>
</tr>
<tr>
<td>PUT</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td><strong>UPDATE</strong> a resource identified by a client-specified URL</td>
</tr>
<tr>
<td>POST</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td><strong>CREATE</strong> a new resource</td>
</tr>
<tr>
<td>DELETE</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td><strong>DELETE</strong> a resource identified by a URL</td>
</tr>
</tbody>
</table>

Representations

REST foresees that resources may be marshalled into many different formats for wire transfer

- “Representation”, Content Type

Common examples:

- XML, JSON, CSV (for programmatic clients)
- HTML, TXT (for human clients)
Content Type Negotiation

Server decides which representation to use for an answer based on:

- Capabilities of the server
- Wishes of client

Communication through HTTP headers:

```
GET /products HTTP/1.1
Host: my.remote.server
Accept: application/json,application/xml,text/html
```

```
HTTP/1.1 200 OK
Date: Tue, 02 Nov 2016 14:38:00 CET
Content-Type: application/json
```

```
{[prod1, prod2, prod3, prod4, prod5]}
```
Summary and Comparison

- Service-Oriented Systems
- Component-Based Systems
- Distributed Object Systems
- Remote Procedure Call Systems
- Message Passing
- Streaming
Sync / Async Interactions?

- Can be either
  - Synchronous
    - Service-Oriented Systems
  - Synchronous
    - Component-Based Systems
  - Synchronous
    - Distributed Object Systems
  - Synchronous
    - Remote Procedure Call Systems
  - Asynchronous
    - Message Passing
  - Not applicable
    - Streaming
Duration of an Individual Connection?

- Long
  - Service-Oriented Systems
- Long
  - Component-Based Systems
- Long
  - Distributed Object Systems
- Long
  - Remote Procedure Call Systems
- Short
  - Message Passing
- Arbitrary, can be very long
  - Streaming
Unicast or Multicast?

- **Unicast**
  - Service-Oriented Systems

- **Unicast**
  - Component-Based Systems

- **Unicast**
  - Distributed Object Systems

- **Unicast**
  - Remote Procedure Call Systems

- **Both**
  - Message Passing

- **Both**
  - Streaming
<table>
<thead>
<tr>
<th>Statefulness</th>
<th>Service-Oriented Systems</th>
<th>Component-Based Systems</th>
<th>Distributed Object Systems</th>
<th>Remote Procedure Call Systems</th>
<th>Message Passing</th>
<th>Streaming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateless</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mostly stateful</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stateful</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stateless</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stateless</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not applicable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Coupling

Maybe referentially decoupled, temporally coupled

Referentially and temporally coupled

Referentially and temporally coupled

Referentially and temporally coupled

Ideally referentially and temporally decoupled

Referentially and temporally coupled

Service-Oriented Systems

Component-Based Systems

Distributed Object Systems

Remote Procedure Call Systems

Message Passing

Streaming
Support for Heterogeneous Clients

- Yes
- Mostly No
- No
- Maybe
- Mostly Yes
- Yes

- Service-Oriented Systems
- Component-Based Systems
- Distributed Object Systems
- Remote Procedure Call Systems
- Message Passing
- Streaming
Next Week:

- Next week we will discuss ATAM, a concrete evaluation model for software architectures.
- The week after, we slightly change topic and focus on cloud computing technologies.
- We will come back to the (micro-)services topic when we discuss scalability and resilience in cloud computing.