LECTURE 6:
ENTERPRISE APPLICATION INTEGRATION (EAI), SERVICE-ORIENTED ARCHITECTURE (SOA) & MIDDLEWARE IN ENTERPRISE ARCHITECTURE
Lecture Contents

• Intro to SOA & Middleware in Enterprise Information Systems
  – Need for EAI, Middleware & SOA in WF currently with business trends
  – EAI & Middleware and where this fits in to Business Processes
  – Evolution from EAI to SOA
  – A first mention of Web Services & it’s role
  – A bit of theory about SOA with an example of it’s application
• Types of communication in EIS
• Types of Distributed Communications
• Message-Oriented Middleware: IBM MQ Systems, RabbitMQ
• Distributed Object-based Systems
• Background: Apache Web Servers, Server Clusters & Names
• RESTful Web Services
SECTION 6.1: INTRODUCTION TO SOA & MIDDLEWARE IN ENTERPRISE INFORMATION SYSTEMS
Recap on Workflow

• Workflow (definition from WorkFlow Management Coalition):
  – “The computerised facilitation/automation of a BP, in whole or part”
  – Workflow technology is often an appropriate solution to BPR activities.
  – Traditionally managed by software (Workflow Management Systems WFMS)

• Thus workflows involve the coordinated execution of multiple ‘tasks’/’activities’ performed by different processing entities, nowadays mostly in distributed heterogeneous environments

• These are very common in enterprises of even moderate complexity

• A workflow system can be defined as a collection of processing steps organized to accomplish some BP
Changes in Context: (cont’d)
Recent Broad Goals/Trends

Goals:
- Low cost
- Streamlined & efficient process
- Monitor & track process execution
- Detect and manage exception
- In-time response, etc
- **Solution:** IT

Business Trends

- Scalewise:
  - Intra-Enterprise
  - Inter-Enterprise
  - Global Interaction

IT Trends

- Mainframe
- Set of Servers
- Set of Services

- Timewise
  - Manual
  - Electronic
  - Web
The Lie of the Land...

- The diagram shows a layer-wise outline of some of the technologies that will be examined in this lecture & how they relate to each other.
- Up to now, only three layers have been considered (mostly BP layer)
Changes in Context: Terminology

• Integrating enterprises’ existing IS applications to run BPs with many s/w systems has used *Enterprise Application Integration (EAI)* technology:
  – User Interface Integration,
  – Data Integration
  – Method or Function Integration
  – Business Process Integration

• *Middleware* is communication facilitator in EAI and this is handled by the *Enterprise Service Bus (ESB)* (akin to a message router in EAI).

*s/w layer allowing many systems to seem to users as a single coherent system with a variety of functionality*
Changes in Context (/2): EAI & Middleware

• In multi-layer architecture the business-logic be put?
  – Can’t put in client (UI) tier
    • Leads to Fat client, reimplemented for each different client type
    • Redistributing clients after each software update
  – Not Data tier as different applications have different uses for same data
  – Has to reside on Middle Layer

• Enterprise Application Integration (EAI)
  – Integrates applications and enterprise data sources so that they can easily share business processes and data
  – Integration done without much changes of applications/data sources
  – All data conversion, security, comms between computers is seamless
Changes in Context (/3): Challenges to BPM

• Methods of Business Process Management are useful when optimising BPs *within* an enterprise.

• Some business environments require many different process designs
  – BP Mass-customization => Automatic BP creation (eg patient health records)
  – BPs evolve dynamically as they execute, through the exchange of information among participants whose relationships evolve as a result

• But BPM is neither scalable nor adaptive by nature
  – So can use a framework based only on BPM to build business applications but too tightly coupled to adapt to future changes.
  – For each change, business dept must interact with IT dept to change software.

• Still need BPM as processes will need to be optimised (Bajwa et al 2008)
• Need increased agility in BPs for loosely-coupled business networks
Typical status quo in Many Enterprise IT Architectures

• Functional and technical application monoliths ubiquitous
  – Stovepipe architectures, application scope creep, redundant implementations, data management and many other agility issues
  – Architectural governance or guidance missing

• Development and integration projects costly and long running
  – Proprietary point-to-point connections, often developed from scratch
  – File transfer is a frequently used integration pattern with numerous architectural drawbacks
  – ‘Roll-your-own’ philosophy works short term, but leads to maintenance headaches eventually

• As a result, horizontal initiatives are much harder to implement than they have to be
  – Example: single customer relationship management solution on top of several line-of-business applications (packages and custom developed)
Service-Oriented Architecture (SOA)

Ecosystem Evolution

Component-Based Development

Enterprise Application Integration (EAI)

- Point-to-Point connection between applications
- Simple, basic connectivity

- EAI connects applications via a centralized hub
- Easier to manage larger number of connections

Messaging Backbone

Service-Oriented Architecture

- Integration of services thro an Enterprise Service Bus (ESB)
- Flexible connections with well defined, standards-based interfaces

- Different parties (even in the same company) may have different:
  - OS, interface, data format, infrastructure, interaction protocols, language, etc

- Automating Supply Chain Mgmt implies bringing all of these together

- As seen, EAI currently solves this but evolution towards SOA
  - Supports flexible s/w dvpt thro ‘loose service coupling’ => no need to talk to IT.
What is Service Oriented Architecture (SOA)?

• Data & BP sharing between applications are EAI’s primary purposes.
  – It links enterprise applications so to communicate with one another and carry out “batch” data transfers
  – But EAI also defines principles for linking multiple systems, such as *message-oriented middleware (MOM)*, of which more later.
  – EAI is maybe old with SOA, but still EAI tools useful for large scale integrations

• SOA provides ‘transactional’ data transfer, needs no third-party s/w:
  – It differs from EAI in that it does not depend on a third-party solution.
  – Links interacting & contracted *services* via comms protocol (i.e. Web Services)

• Services are useful because they:
  – Can be reused *in heterogeneous environments* at multiple levels, including code, platform, so enhancing flexibility in the design of enterprise applications
  – Are implemented by 1/more code components in homogeneous environments
  – Aggregating one or more components into a service and making them accessible through messages using open standards.
SOA Fundamentals: Modularity, Layering & Loose Coupling

Example:
An insurance company uses three SAP R/3, MS Visual Basic, and COBOL applications to manage customer information, check for fraud, and calculate payments. The user interfaces (UIs) are the only access points.

A multi-step, multi-user business process for claim handling, executing in IBM WebSphere, is supposed to reuse the functions in the existing applications. How to integrate the new business process with the three legacy applications in a flexible, secure, and reliable way?
SOA Principle 1: Modularity (i.e. Separation of Concerns)

• Motivation:
  – Integrating monolithic applications (“stovepipes”) is hard (e.g., traditional Enterprise Resource Planning packages)

• Solution
  – Refactor to services, expose service interface only, hide implementation details (a.k.a. encapsulation)

• Consequences
  – Service contracts have to be defined and interpreted (by tools and/or at runtime)
  – Services have to be located and invoked in a coordinated manner
  – Service invocations cannot have undesired side effects (data mgmt?)
SOA principle 2: Layering (logical and/or physical)

• Motivation
  – Service characteristics such as interface granularity & lifecycle vary: e.g. technical logging service vs. claim checking business process

• Solution
  – Organize SOA into 3++ architectural layers

• Consequences
  – More abstraction (i.e. services can be composed out of other services leading to composite applications),
  – This requires communications infrastructure (usually Web Services)
  – 1st law of distribution: “best remote call is the one you don’t make”
SOA principle 3: Loose coupling through messaging

• Motivation
  – Once applications are modularized, dependencies between services occur

• Solution
  – Couple services loosely (several dimensions: location-, time-, invocation context-independent)
  – e.g. messaging system decouples in time, location, language dimensions
  – Messaging via any MOM system (eg. HTTP, IBM MQSeries, RabbitMQ)
  – These are a flavour of transport used in an application built to SOA principles,
  – Any transport that can send/receive bags of bytes with your messages will work

• Consequences
  – Messaging means single implementation by default (no remote objects)
  – Asynchronous communication complicates systems management
  – e.g. is there any answer coming from the receiver at all???
SOA principle n: Service virtualization and flexible infrastructure

• Motivation
  —“I don’t care about a particular provider, just chose the one that at this point in time is best for me”

• Solution
  —From WWW to service bus/cloud
  —Two-level programming

• Consequences
  —Many open issues e.g., trust and privacy, precise semantics, QoS

• First isolated steps
  —Software as a Service (SAAS), e.g. Salesforce.com CRM & Amazon Storage
  —Dynamic matchmaking of services on demand
Slide Aside: What is virtualization?

- A simple example of virtualization: my desktop

My new W7 desktop
HK background)

My old XP desktop
(Mountain background)

- My old (physical) machine (XP, applications & all) has been virtualized using VMWare software - This is hardware virtualization (see fig)
- Another form of virtualization you will be familiar with is software virtualization (e.g. Java Virtual Machine or JVM)
- There are many other forms of virtualization at different levels.
SOA in Practice: Use of Web Services

• Web Services:
  • Web Services = SOA + Standards (REST/HTTP/XML etc)
  • WS: natural evolution of m/w & EAI platforms to build distributed applications
  • No difference from middleware itself except being invoked via Internet
  • A standardized means of dealing with integration, where traditional methods are vendors/application/language specific

• Web Services is a model for using the Web:
  – To automatically initiate processes via the Web using programs
  – A method for describing, publishing & initiating processes dynamically in a distributed environment
  – Not necessarily using a Web browser

• If you can imagine a way of electronically delivering something:
  – Of value to a customer
  – That will solve a problem, or
  – Provide some service to them

• Then you have a viable example of a Web service!
More on Web Services

• Put simply, WS is only a server that listens for/ replies with (usually) REST commands generally via HTTP
• WS is an interface that describes:
  – a collection of n/w accessible operations
  – through standardized XML Messaging
• Web services encapsulate business functions:
  – Check credit card number, Payment processing, Stock quotes, Request quote
  – Can be used to compose BPs (e.g. Travel planning, Health care, Etc, etc)
• Benefits of Web Services Include:
  – Decoupling of service interfaces from the implementation
  – Enabling dynamic service binding
  – Providing interoperability among different platforms
  – Existing applications can be wrapped as Web services
  – Client & Service can use different platforms & programming languages
  – Services can be composed to make composite services
SOA in Practice: Example of SCM

**Supply Chain Management**

- No parts at Plant? ERP system messages HQ -> queries ERP system at other plant for item. None at HQ? HQ sends e-order to supplier’s ERP system
- EA for inventory query/supply order:
  - EA needs 4 systems connected by 3 proprietary interfaces.
  - Mainframe at 1st plant connects to HQ’s Windows servers -> connects to 2nd plant’s IS & supplier Sun box.
  - As seen, this tightly coupled integration is inflexible & costly to modify/maintain.
  - E.g., in the EA, to add new suppliers, competitive bidding on supplier contracts are complex/expensive.
SOA in Practice: Example of SCM (/2)

- **B2B Commerce Facilitation with SOA**
- Converting to an SOA allows for B2B commerce without system reworking systems.
- As well as eliminating proprietary interfaces, SOA enables 1ˢᵗ plant to check directly with 2ⁿᵈ plant & place orders without need for HQ’s computer.
- HQ sees transactions with own WS & to-and-fro messages btw 2ⁿᵈ plant & supplier.
SOA in Practice: Example of SCM (/3)

- SOA increases B2B commerce by manufacturer holding competitive bidding system.
- Suppliers bidding to win business required to use WS to connect to bidding system.
- Again, can do with traditional technology but costs are so high that it’s rarely done.
- SOA allows manufacturer to manage suppliers/ costs & suppliers can get business.
- Further, when suppliers are replaced or new suppliers added, IT can now respond quickly and inexpensively to the business decisions.
SECTION 6.2: MIDDLEWARE IN DISTRIBUTED SYSTEMS
Role of Middleware in Distributed Applications

- **Observation**
  - Role: to provide common services/protocols in Distributed Applications
  - Can be used by many different distributed applications

- **Middleware Functionality**
  - (Un)marshalling of data: necessary for integrated systems
  - Naming protocols: to allow easy sharing, discovery of resources
  - Enforces business rules
  - Security protocols: for secure communication
  - Scaling mechanisms, such as for replication & caching (e.g. decisions on where to cache etc.)
  - Rich set of communications protocols: to allow some applications to transparently interact with others regardless of location.
SECTION 6.2.1: BASICS OF MESSAGE PASSING
Introduction to Message Passing

• With networks & distributed systems in which processors are linked by a communications medium, message passing is common.

• There are 2 basic message passing primitives, send & receive

  send primitive: sends a message (data) on a specified link from one process to another,

  receive primitive: receives a message on a specified link from other processes.

• The send primitive has different semantics depending on whether the message passing is synchronous or asynchronous.
  – Asynchronous: (non-blocking)
    Sender resumes as soon as the message is passed to comms software
  – Synchronous: sender is blocked until
    • The OS or middleware notifies acceptance of the message, or
    • The message has been delivered to the receiver, or
    • The receiver processes it & returns a response
Synchronous Message Passing

• In synchronous message passing each channel forms a direct link between two processes.
• Suppose process A is sending data to process B: When process A executes **send** primitive it waits/blocks until process B executes its **receive** primitive.
• Before the data can be transmitted both A & B must ready to participate in the exchange.
• Similarly the **receive** primitive in one process will block until the send primitive in the other process has been executed.
Asynchronous Message Passing

• In asynchronous message passing **receive** has the same meaning/behaviour as in synchronous message passing.
• The **send** primitive has different semantics.
• This time the channel between processes A & B isn’t a direct link but a message queue.
• Therefore when A sends a message to B, it is appended to the message queue associated with the asynchronous channel, and A continues.
• To receive a message from the channel, B executes a **receive** removing the message at the head of the channel’s message queue and continues.
• If there is no message in the channel the receive primitive blocks until some process adds a message to the channel.
SECTION 6.2.2: INTRODUCTION TO COMMUNICATIONS MIDDLEWARE
A Bit About Socket Communications

• **Sockets**
  – Most network comms uses the *client server* or *(request response)* model.
    i.e. One process, the client, connects to the other (server), e.g. seeking info.
  – Client needs to know of the existence of and the address of the server,
    But server needn’t know client’s address/existence before connection.
  – After connection is established, both sides can send and receive information.
  – Making a connection differs for client/ server, but both use basic sockets.
  – A socket is one end of an interprocess communication channel.
  – Both processes establish their own socket.

• Unfortunately, they:
  – Divide the message into packets & send via various routings to receiver
  – Reliable but require receiver To be able to interpret the message
  – Can disrupt the order of the packets.
Classification of Middleware

• Classify middleware technologies into the following groups:

1. **Bog-standard Sockets**
   • The basis of all other middleware technologies.

![Diagram of Sockets](image)

2. **RPC – Remote Procedure Call (more later)**
   • RPCs provide a simple way to distribute application logic on separate hosts
   • Allow one host to request a service from a host on another computer in a network without having to understand network details.

![Diagram of RPC](image)

Stubs are pieces of code that can connect to other network procedures but pretend to be local procedure calls. Have to wrap/unwrap data/results.
Classification of Middleware (/2)

3. **TPM - Transaction Processing Monitors:**
   - TPMs are a special form of MW targeted at distributed transactions.

4. **DAM - Database Access Middleware:**
   - DBs can be used to share & communicate data between distributed applications.
Classification of Middleware (/3)

5. **Distributed Tuple:**
   - Distributed tuple spaces implement a distributed shared memory space.
   - In practice this works like a DB, separating ‘sender’/’receiver’ in time.

![](diagram1.png)

6. **DOT (Distributed Object Technology):**
   - Here both sender/receiver share an object which they both operate on.
   - Example of this is Enterprise Service Bus.

![](diagram2.png)
Classification of Middleware (/4)

7. **MOM (Message Oriented Middleware):**
   - In MOM, messages are exchanged asynchronously between distributed applications (senders and receivers).

![Diagram of MOM](image)

8. **Web services:**
   - Web services provide access to services via a defined interface, typically accessible through the web protocol HTTP.

![Diagram of Web Services](image)
Summary of Communications Middleware

• Essentially a range of types of communications middleware
• All can be used to implement others, all are suited to different cases
  – All carry some payload from one side to another <with details>
  – Some of these payloads are ‘active’ and some are ‘passive’
  – Also differ in granularities and whether synchronous or not.
SECTION 6.3: OBJECT-BASED COMMUNICATION IN DISTRIBUTED SYSTEMS
Remote Procedure Call (RPC)

- **Rationale:** Why RPC?
- **Distribution Transparency:**
  - Send/Receive don’t conceal comms at all (requester/client has to know details on the server) – need to achieve *access* transparency.
- **Answer: Totally New ‘Communication’ System:**
  - Allows programs to communicate by calling another program’s methods.
- **Mechanism**
  - RPC is synchronous so when a process on machine A calls a method on machine B, calling process on A is suspended and
  - Execution of the called procedure takes place on B.
  - No message passing at all is visible to the programmer.
  - Application developers familiar with simple communications model.
### Basic RPC Operation

1. Client procedure calls client stub
2. Stub builds message, calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to stub.
5. Stub unpacks parameters, calls server.
6. Server works, returns result to stub.
7. Stub builds message, calls local OS.
8. OS sends message to client’s OS.
9. Client OS gives message to client stub.
10. Stub unpacks result, returns to client.
Distributed Objects

• **Introduction**

  – As mentioned above a key aspect of communications middleware is that one type can be used to imitate/implement another.
  
  – In *distributed object-based* systems, notion of an object plays a key role in establishing *distribution transparency* (i.e. that the receiver can be distant)
  
  – Key object feature is having data & operations on that data (aka *methods*)
  
  – Methods are made available through an *interface*.
  
  – Everything is treated as an object & clients are offered services/resources as objects that they can invoke.
  
  – Distributed objects form an important paradigm because it is relatively easy to hide distribution aspects behind an object's interface.
  
  – As object can be almost anything, so a useful paradigm for building systems.
  
  – Process can only change object’s data by invoking methods made available to it via an object's interface.
Distributed Objects (/2)

• **Architecture**
  – This separation between interfaces and the objects implementing them is crucial for distributed systems.
  – Separation allows for placing interface at one machine, with object itself on another machine.
  – This organization is commonly referred to as a *distributed object definition*.

![Organization of a Distributed Object with a Client-Side Proxy](image-url)
Distributed Objects (/3)

- **Remote Method Invocation (RMI)**
  - Java Remote Method Invocation (RMI) system allows an object running in one JVM to invoke methods on an object running in another Java VM.
  - RMI gives applications *transparent, lightweight* access to *remote objects*.
  - RMI defines a high-level protocol and API.
  - Programming distributed applications in Java RMI is simple:
    - It is a single-language system.
    - Remote object coder must consider behaviour in a concurrent environment.

- **Java RMI Applications**
  - RMI is supported by two java packages `java.rmi` & `java.rmi.server`
  - An application that uses RMI has 3 components:
    - an *interface* that declares headers for remote methods;
    - a *server* class that implements the interface; and
    - one or more *clients* that call the remote methods.
Distributed Objects (/4)

- **RMI Architecture**
  - *Stub*: lives on the client; pretends to be the remote object
  - *Skeleton*: lives on the server; talks to the true remote object
  - *Reference Layer*: determines if referenced object is local or remote
  - *Transport Layer*: packages remote invocations; dispatches messages between stub and skeleton
Distributed Objects (/5)

• **Java RMI Basics:** (Assumes client stub, server skeleton in place)
  – Client invokes method at **stub**
  – **Stub** marshals request and sends it to server
  – Server ensures referenced object is active:
    • Create separate process to hold object
    • Load the object into server process
    • ...
  – Object **skeleton** unmarshalls request & referenced method is invoked
  – If request contains object reference, invocation is applied recursively (i.e., server acts as client)
  – Result is marshalled and passed back to client
  – Client **stub** unmarshalls reply & passes result to client application
Distributed Objects (/6)

**RMI Registry**

- A simple server-side ‘phonebook-style’ naming facility allowing remote clients to get a reference to a remote object
  - Servers name & register their objects to be accessed remotely with the RMI Registry.
  - Clients use the name to find server objects and obtain a remote reference to those objects from the RMI Registry.
- Registry service is background program with a list of registered server names on a host and invoked by: `rmiregistry port &`
- Registry service is provided by a Naming object providing two key methods:
  - *Bind*: to register a name and server
  - *Lookup*: to retrieve the server bound to a name
import java.rmi.*;
import java.rmi.server.*;
public class Database extends UnicastRemoteObject
        implements DatabaseInterface {
    private int data = 0; // the database

    public Database(int value) throws RemoteException {
        data = value;
    }

    public int read () throws RemoteException {
        return data;
    }

    public void write (int value) throws RemoteException {
        data = value;
        System.out.println ("New value is: " + data);
    }
}
RMI Example (/2): Database Server

```java
import java.rmi.*;
import java.rmi.server.*;
public class DatabaseServer {

    public static void main(String[] args) {
        try {
            // create Database Server Object
            Database db = new Database(0);

            // register name and start serving
            String name = "rmi://fuji:9999/DB";
            Naming.bind(name, db);
            System.out.println(name + " is running");
        } catch (Exception ex) {
            System.err.println(ex);
        }
    }
}
```
import java.rmi.*;
public class DatabaseClient {
    public static void main (String[] args) {
        try {
            // set RMI Security Manager
            System.setSecurityManager(new RMISecurityManager() {
                public void checkConnect(String host, int port) {
                }
                public void checkConnect(String host, int port, Object Context) {
                }
            });
            // get database object
            String name = "rmi://fuji:9999/DB";
            DatabaseInterface db = (DatabaseInterface) Naming.lookup(name);
            int value, rounds = Integer.parseInt(args[0]);
            for (int i = 0; i < rounds; i++) {
                value = db.read();
                System.out.println("read: " + value);
                db.write(value+1);
            }
        } catch (Exception ex) {
            System.err.println (ex);
        }
    }
}
RMI Example (/4): Building the Application

• Steps involved in Building the Application:

1. Compile the code:
   ```
javac Database.java DatabaseClient.java
   DatabaseInterface.java DatabaseServer.java
   ```

2. Generate stub and skeleton class files:
   ```
   rmic Database (note: not needed for Java 5 or later)
   ```

3. Start the RMI registry:
   ```
   rmiregistry 9999 &
   ```

4. Start the Server:
   ```
   java -Djava.security.policy=java.policy DatabaseServer
   ```

5. Start the Client:
   ```
   java -Djava.security.policy=java.policy DatabaseClient 10
   ```
SECTION 6.4: MESSAGE QUEUING SYSTEMS
Message-Oriented Persistent* Comms

- **Rationale: Why Another Messaging System?:**
- **Scalability:**
  - Other messaging systems, do not scale well geographically.
- **Granularity:**
  - Sockets supports messaging $O(\text{ms})$. Distributed messaging can take min/hours.
- **What about RPC?:**
  - In DS can’t assume receiver is “awake” => default “synchronous, blocking” nature of RPC often too restrictive.
- **How about Sockets, then?:**
  - Wrong level of abstraction (only “send” and “receive”).
  - Too closely coupled to TCP/IP networks – not diverse enough
- **Answer: Message Queueing Systems:**
  - MQS give extensive support for **Reliable Asynchronous Communication**.
  - Offer medium-term storage for messages – don’t require sender/receiver to be active during message transmission.
  * Can store message if Not delivered immediately
Message-Oriented Persistent Comms. (/2)

• Message Queuing Systems:
  – *Basic idea*: applications communicate by putting messages into and taking messages out of “message queues”.
  – Only guarantee: your message will eventually make it into the receiver’s message queue => “loosely-coupled” communications.
  – Asynchronous persistent communication thro middleware-level queues.
  – Queues correspond to buffers at communication servers.

• Four Commands:

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>Append a message to a specified queue.</td>
</tr>
<tr>
<td>Get</td>
<td>Block until the specified queue is nonempty, and remove the first message.</td>
</tr>
<tr>
<td>Poll</td>
<td>Check a specified queue for messages, and remove the first. Never block.</td>
</tr>
<tr>
<td>Notify</td>
<td>Install a handler to be called when a message is put into the specified queue.</td>
</tr>
</tbody>
</table>
Message-Queuing System Architecture

• **Operation:**
  – Messages are “put into” a *source queue*.
  – They are then “taken from” a *destination queue*.
  – Obviously, a mechanism has to exist to move a message from a source queue to a destination queue.
  – This is the role of the *Queue Manager*.
  – These are message-queuing “relays” that interact with the distributed applications and with each other.
  – Not unlike routers, these devices support the notion of a DS “overlay network”.
Role of Message Brokers

• **Rationale:**
  Often need to integrate new/existing apps into a “single, coherent Distributed Information System (DIS)”.

• **Problem:** different message formats exist in legacy systems
• Can’t “force” legacy systems into single, global message format.
• “Message Broker” allows us to live with different formats
• Centralized component that takes care of application heterogeneity in an MQ system:
  – Transforms incoming messages to target format
  – Very often acts as an application gateway
  – May provide subject-based routing capabilities ⇒ *Enterprise Application Integration*
Message Broker Organization

- General organization of message broker in a MQS – also known variously as an “interface engine”.

Queuing layer allows source Client to look up send queue for destination client.

To the source & Destination client Message Broker just looks like any other App.
IBM’s WebSphere MQ

• **Basic concepts:**
  – Application-specific messages are put into, removed from queues
  – Queues reside under the regime of a queue manager
  – Processes can put messages only in local queues, or thro an RPC

• **Message transfer**
  – Messages are transferred between queues
  – Message transfer between process queues requires a channel
  – At each endpoint of channel is a *message channel agent*
  – Message channel agents are responsible for:
    • Setting up channels using lower-level n/w comm facilities (e.g. TCP/IP)
    • (Un)wrapping messages from/in transport-level packets
    • Sending/receiving packets
IBM’s WebSphere MQ (/2)

• Supported Topologies are:

1. **Hub/spoke** topology (point-to-point queues)
   • Apps subscribe to "their" QM.
   • Routes to hub QM def’d in spoke QMs.

2. **Distributed Publish/Subscribe**:
   • Apps subscribe to topics & publish messages to multiple receivers.
   • 2 Topologies: **Clusters** and **Trees**:

   **Cluster**: Cluster of QMs connected by channels. Published messages sent to all connected QMs of the published topic.

   **Tree**: Trees allow reducing number of channels between QMs.
IBM’s WebSphere MQ (/2)

- **Principles of Operation:**
  - Channels are inherently unidirectional
  - Automatically start MCAs when messages arrive
  - Any network of queue managers can be created
  - Routes are set up manually (system administration)

General organization of IBM's WebSphere Message-Queuing System
IBM’s WebSphere MQ (/3)

- **Routing:** Using logical names, in combination with name resolution to local queues, possible to route message to remote queue
  - Sending message from one QM to another (possibly remote) QM, each message needs destination address, so a transmission header is used
  - MQ Address has two parts:
    1. Part 1 is the *Destination QM Name* (say QMX)
    2. Part 2 is the *Name of the Destination Queue* (i.e. QMX’s destination Queue)
  - As each QM has unique name each QM knows each other by an **Alias**

![Diagram of MQ routing and address structure]
Advanced Message Queuing Protocol (AMQP)

• Why AMQP?

1. Lack of standardization:
   • Little standardization in MOM products (mostly proprietary solutions).
     – E.g. 1: JMS Java-dependent, doesn’t specify wire protocol only an API. => different JMS providers not directly interoperable on wire level.
     – E.g. 2: IBM Websphere clunky and expensive

2. Need for bridges\(^1\) for interoperability:
   • To achieve interoperability between different queueing systems, 3rd party vendors offer *bridges*.
   • These complicate the architecture / topology, increase costs while reduce performance (additional delay).

\(^1\)Entities that help in different stages of message mediation
AMQP (/2)

• **Characteristics of AMQP:**
  
  – What is it? Open protocol for enterprise messaging, supported by industry (JP Morgan, Cisco, Microsoft, Red Hat, Microsoft etc.).
  
  – Open/ Multi-platform / language messaging system.
  
  – AMQP defines:
    
    1. Messaging capabilities (called *AMQP model*)
    
    2. *Wire-level protocol* for interoperability
  
  – AMQP messaging patterns:
    
    1. Request-response: messages delivered to a specific queue (like C/S)
    
    2. Publish/Subscribe: messages delivered to a set of receiver queues
    
    3. Round-robin: message distribution to set of receivers based on availability

• **AMQP Model (simplified):**
AMQP Example: RabbitMQ Model

- Direct exchange:
  - Key: q1
  - Reply-to: tmp1
  - Cor-id: 1

- Topic exchange:
  - Key: STK.US.C

- Fan-out:
  - Key: key1

- Filter exchange:
  - Key: STK.US.* AND type == pdf

- Queue server:
  - q1: !private
  - q3:
  - q4:
  - q5: !durable
  - q6:

- Channel:
  - s1
  - s2
  - s3
  - s4

- TCP connection:

**Application Thread**

**Queue Server**

**Channel**

**Message**

**Distribution**

**Fan-out**

**Queue Server**

**Exchange**

**Binding**

Lecture 6: Technical Architecture: Middleware

© Houcheng Lin, slideshare.net/HouchengLin/rabbitmq-model-and-some-example-applications

CA4101 Lecture Notes (Martin Crane 2016)
• Download Erlang, RabbitMQ for your OS (My versions were RabbitMQ 3.6.5, Erlang 19.1)
• Start it running (may need to explicitly ‘start the service’)
• Open a web-browser, in the command line type http://localhost:15672/
• Login as guest, password as guest
• Should see something like this:
For **Receive** bit

- From the command prompt run IDLE (Python GUI) – this opens a Python shell
- Go to `c:\Python27\Scripts` directory
- Open `receive.py` and run it
• For **Send** bit
  
  – From the command prompt run IDLE (Python GUI) – this opens a Python shell
  – Go to `c:\Python27\Scripts` directory
  – Open `send.py` and run it

```python
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello1')

channel.basic_publish(exchange='',
    routing_key='hello1',
    body='Hello World!

print(" [x] Sent 'Hello World!'")
connection.close()
```
• Afterwards should see something like this:
SECTION 6.5: DISTRIBUTED WEB-BASED SYSTEMS
Introduction to Web Services

- WS offered by one electronic device to other, communicating via web
- Here HTTP originally designed for human-to-machine comms, is used for M2M chatter, e.g. in XML.
- In practice, WS typically gives web-interface to a DB server, used by another web server, or a mobile application, providing UI to end user.
- Other systems used to interact with WS using SOAP* messages, typically using HTTP XML serialization.
- Later this was extended to include
  - REST-compliant Web services, where service’s main point is to process Web resources *representations* using uniform set of stateless operations
  - Arbitrary Web services, which may expose an arbitrary set of operations

*Simple Object Access Protocol*, now largely falling out of use, though with some specialist applications
Background to Web Services

- **Apache Web servers**
  - *Observation:* More than 24% of all 1 billion* Websites are Apache.
  - The server is internally organised more or less according to the steps needed to process an HTTP request.
  - The anatomy of an Apache Web Server is shown below:

*340,793,662 of 1,436,724,046 in Nov 2016, source news.netcraft.com/archives/category/web-server-survey/
**Background to Web Services (/2)**

- **Server Clusters**
  
  - **Essence:** To improve performance and availability, WWW servers are often clustered in a way that is transparent to clients.
  
  - Below a server cluster is used in combination with a front end to implement a Web service.
Background to Web Services (/3)

• Naming: The Naming Service
  – Names play a very important role in all computer systems.
  – Used to share resources, uniquely id entities, refer to locations
  – An important issue with naming is that a name can be resolved to the entity it refers to, aka Name resolution
  – To resolve names, need to implement a Naming System.
  – Name in distributed systems: string of bits/chars that refer to it.
  – To operate on an entity need an Access Point (an address.)
  – Note: A location-independent name for an entity $E$, is independent from the addresses of the access points offered by $E$.
  – Entities can offer more than one access point
    • e.g. phone is person’s access point, phone number as address.
  – Entities may change access points over time just as changing jobs or Internet Service Providers, means changing e-mail addresses.
Background to Web Services (/4)

- **Naming: Identifiers**
  - **Pure name**
    - A name that has no meaning at all; it is just a random string.
    - Pure names can be used for comparison only.
  - **Identifier**: A name having the following properties:
    - P1: Each identifier refers to at most one entity
    - P2: Each entity is referred to by at most one identifier
    - P3: An identifier always refers to the same entity (prohibits reusing an identifier)
  - **Observation**
    - Identifier needn’t necessarily be a pure name i.e. can have content
Introduction to REST

- **REST**, or **REpresentational State Transfer**, is a distributed communication architecture
  - It is fast becoming the lingua franca for Cloud Computing
  - The central abstraction in REST is the **Resource**.
  - A resource in the RESTful sense is anything that has an URI.
  - In practice, resource is an info item that has hyperlinks to it.
REST

• REST tries to isolate complexity at Clients & Service:
  – Service:
    • Could need logic/computation to process XML to maintain Resources & generate their representation.
  – Client:
    • May have to process XML to extract info from XML representation.

• But this complexity is kept from the transport level.

• Resources have certain properties:
  – Representation: usually some form of xml.
  – State: i.e. they are mutable.

• Note:
  – In a RESTful request on it, resource itself stays service-side.
  – If request succeeds, requester gets resource’s representation (this transfers from server to requester machine).
HTTP (REST) Operations & Example

**Essence**: Communication in Web generally based on HTTP.

- Some HTTP Operations and their effects is shown below:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Idempotent?</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>Yes</td>
<td>Request to return the header of a document</td>
</tr>
<tr>
<td>Get</td>
<td>Yes</td>
<td>Request to return a document to the client</td>
</tr>
<tr>
<td>Put</td>
<td>Yes</td>
<td>Request to store a document</td>
</tr>
<tr>
<td>Post</td>
<td>No</td>
<td>Provide data that are to be added to a document</td>
</tr>
<tr>
<td>Delete</td>
<td>Yes</td>
<td>Request to delete a document</td>
</tr>
</tbody>
</table>

- The diagram below is client communications with a resource
- GET request returns my bio (html or video)
- Typical html resource representations have other resource links
- Could then be target of HTTP requests with appropriate CRUD verbs
Another Example in REST

• Shown is client comms with a REST bank application:
  – GET/PUT/POST/DELETE replaced with analogous bank methods
  – If we have US$100 on deposit, can do all four methods:

```xml
GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?><account>
  <account_number>12345</account_number>
  <balance_currency="usd">100.00</balance>
  <link rel="deposit" href="/account/12345/deposit" />
  <link rel="withdraw" href="/account/12345/withdraw" />
  <link rel="transfer" href="/account/12345/transfer" />
  <link rel="close" href="/account/12345/close" />
</account>
```

– With US$25 overdrawn, we only have one option.

```xml
GET /account/12345 HTTP/1.1

HTTP/1.1 200 OK
<?xml version="1.0"?><account>
  <account_number>12345</account_number>
  <balance_currency="usd">-25.00</balance>
  <link rel="deposit" href="/account/12345/deposit" />
</account>
```
A Subtlety: URIs are Opaque

• A URI is meant to be opaque
  – Means that URI: `http://bedrock/citizens/fred` has no inherent connection to the URI: `http://bedrock/citizens/`
  – Although Fred happens to be a citizen of Bedrock.
  – Of course, good designers devise URIs akin to what they identify, but URIs have no intrinsic hierarchical structure.

• A Note of caution
  – URI syntax like that for file system navigation, but this can mislead:
    – URIs are opaque identifiers, each naming exactly one resource.
    – i.e. recall your diagrams from signavio
Bibsonomy Site

- This is a link to my Bibsonomy Account with some bookmarks/links
- Useful as a sandbox for playing with RESTful stuff.
RESTCLIENT GET Operation & Result

- RESTCLIENT allows you to test GET/PUT/POST/DELETE in REST
- This is the result of a GET operation on the URL
- Status code ‘200 OK’ tells us it’s gone through ok 😊
RESTCLIENT GET Operation & Result (/2)

- The stupendous result of the previous operation 😊

```xml
<bibsonomy stat="ok">  
  <user name="mcrane" name="mcrane" email="mcrane@gmail.com" spammer="false" prediction="0" algorithm="admin" toClassify="0" href="https://www.bibsonomy.org/api/users/mcrane" >  
    <groups start="0" end="0" />  
  </user>  
</bibsonomy>
```
RESTCLIENT POST Operation & Result

• You will need to supply your own hash for this command
• The body and response of a bookmark you want to POST
• Copy this hash – you’ll need it if you want to alter/delete the POST

---

**Request**

- **Method**: POST
- **URL**: https://www.bibsonomy.org/api/users/martycrane/posts
- **Body**:
  ```xml
  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <bibsonomy>
  <post description="Boiler Replacement pages">
  <user name="martycrane">
  <tag name="Boiler">
  <group name="public">
  <bookmark url="http://www.seai.ie/power_of_one/heat_your_home_for_less/Replacing_Your_Boiler/" title="SEAI Replacing Your Boiler"/>
  </post>
  </bibsonomy>
  ```

**Response**

- **Response Body**:
  ```xml
  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <bibsonomy status="ok">
  <resource id="5184714384881675055290f47af0462"/>
  ```
RESTCLIENT POST Operation & Result (/2)

- The fabulous result of the previous POST operation 😊
RESTCLIENT PUT Operation & Result

- Do a PUT command if you want to alter a previous POST
- The body/response of a bookmark you have POSTed & want to PUT
- Note the use of the resource hash to change the bookmark
RESTCLIENT PUT Operation & Result (/2)

- The breathtaking result of the previous PUT operation 😊
RESTCLIENT DELETE Operation & Result

- Do a DELETE command if you want to get rid of a previous POST
- The body/response of a bookmark you want DELETED
- Note again the use of the resource hash to DELETE the bookmark
....Now you don’t - the result of the previous DELETE operation
Lecture Summary

• SOA not just up-and-coming technology but here already
• Middleware enables much comms functionality in DS
• Especially the many types of interaction needed
• With rational reasons for every one!
  – *Sockets* the original of the species, still fundamental
  – *Remote Procedure Call* (RPC) enables transparency
  – But need *Message Queuing Systems* for persistent comms
    • IBM Websphere is ok but a bit old, clunky & tired at this stage?
    • AMQP open source, more flexible, better Industrial support?
    • Open source ones RabbitMQ, ZeroMQ, Kafka etc all examples
    • Useful for subject-based and/or publish/subscribe systems
  – Now a lot of M2M chatter takes place over Web Services
• RESTful web services are a powerful xml form of comms
References


