Distributed Applications

Basic choices

▶ Architecture
  ▶ Client/Server architecture
  ▶ Web-Architecture

▶ Middleware
  ▶ Communication between program components
  ▶ Requirements
    ▶ Language independence
    ▶ Platform independence
    ▶ Location independence

▶ Security
Client/Server Architecture

- Application divided in client-part and server-part
- Five possible divisions of standard (six) layer architecture (thin client → fat client)
- Characteristics fixed in the requirements (# of users, operating systems, database systems, . . . )

advantages: traceability of user session, special protocols, design influenced by # users

disadvantages: scalability, distribution of client software, portability
Web Architecture

- Client: only I/O layer; Server: everything else
- Client requirements: Web browser (user interface)
- Server requirements:
  - Web server (distribution of documents, communication with application)
  - Application server (application-specific and application-general objects)
  - Database server (persistent data)

advantages: scalability (very high number of users, in particular with replicated servers), maintainability (standard components), no software distribution required

disadvantages: restriction to HTTP, stateless and connectionless protocol requires implementation of session management, different Web browsers need to be supported (Internet Programming)

Current technology addresses some of the disadvantages: Servlets, ASP, . . .
Refinement: N-tier Architecture

▶ Physical deployment follows the logical division into layers (tiers)

▶ Why?
  ▶ Separation of concerns (avoids e.g. mixing of presentation logic and business logic)
  ▶ Scalability
  ▶ Standardized frameworks (e.g., Java Platform, Enterprise Edition, Java EE 6) handle issues like security and multithreading automatically

▶ Example (Java EE):
  ▶ Presentation: Web browser
  ▶ Presentation logic: Web Tier (JSP/servlets, JavaServer Faces, JavaBeans)
  ▶ Business logic: Business Tier (Enterprise JavaBeans, Web Services)
  ▶ Data access: Enterprise Information System Tier (Java Persistence API, JDBC, Java Transaction API)
  ▶ Backend integration (legacy systems, DBMS, distributed objects)
Enterprise JavaBeans (EJB): Goals

- Part of Java Platform, Enterprise Edition (Java EE 6)
- A SPECIFICATION! but implementations are available
- Server-side component architecture for enterprise applications in Java
- Defines interaction of components with their container
- Development, deployment, and use of web services
- Abstraction from low-level APIs
- Deployment on multiple platforms without recompilation
- Interoperability
- Components developed by different vendors
- Compatible with other Java APIs

---

1 → main target: business logic, between UI and DBMS
2 directory services, transaction management, security, resource pooling, fault tolerance
EJB Component Types

Session Beans

- Interfaces to server-side operations
- Typically business methods
- Three kinds
  - Stateless Session Bean: no state carried over between method invocations; one Bean instance can be shared between multiple clients
  - Stateful Session Bean: maintains state between method invocations; one Bean instance per client
  - Singleton Bean: one instance for all
EJB Component Types /2

Message-Driven Beans

- Event Listeners
- Asynchronous Messaging

Entity Bean

- Object View of RDBMS; object-relational mapping
- Persistence defined separately with JPA (Java Persistence API)
EJB Component Types /3

- All components implemented as POJOs (plain old Java objects)
- No subclassing or implementing of particular interfaces required
- Special roles imposed by annotations

All invocations through interfaces

- Local interface: for method invocations inside the same VM
- Remote interface: for method invocations with unknown location (less efficient)
- Implementing one bean means implementing several interfaces and classes consistently
EJB Example: Remote Interface
A plain Java interface

```java
public interface CalculatorCommonBusiness {
    /**
     * Adds all arguments
     * @return The sum of all arguments */
    int add(int... arguments);
}

public interface CalculatorRemoteBusiness extends CalculatorCommonBusiness {
}
```
public class CalculatorBeanBase implements CalculatorCommonBusiness {
   /**
    * {@link CalculatorCommonBusiness#add(int...)}
    */
   @Override
   public int add(final int... arguments) {
      // Initialize
      int result = 0;
      // Add all arguments
      for (final int arg : arguments) {
         result += arg;
      }
      // Return
      return result;
   }
}
EJB Example: Bean Class
A plain Java class with annotations

```java
import javax.ejb.LocalBean;
import javax.ejb.Stateless;
@Stateless (name = CalculatorEJB)
@Local (CalculatorRemoteBusiness.class)
public class SimpleCalculatorBean extends CalculatorBeanBase {
    /*
     * Implementation supplied by common base class
     */
}
```
Lower Level Services
Lower Level Services

Connection of resources in Client/Server architecture

1. Sockets (TCP/IP, . . .)
2. RPC
3. RMI
4. SOAP (Simple Object Access Protocol)/Web Services
Sockets

- Software terminal of a network connection (a data structure)
- Two modes of communication to host
  - Reliable, bidirectional communication stream or
  - Unreliable, unidirectional one-shot message
- Local variant: inter-process communication (IPC)
- Low level:
  - Manipulation of octet-streams required
  - Custom protocols
Sockets in Java
Server: Read two numbers and output their sum

ServerSocket serverSocket = new ServerSocket(1234);
while ( true ) {
    Socket client = serverSocket.accept();
    InputStream input = client.getInputStream();
    OutputStream output = client.getOutputStream();
    int value1 = input.read();
    int value2 = input.read();
    output.write(value1 + value2);
    input.close();
    output.close();
}

Sockets in Java

Client: Send two numbers and obtain their sum

```java
Socket server = new Socket("localhost", 1234);
InputStream input = server.getInputStream();
OutputStream output = server.getOutputStream();
output.write(1);
output.write(2);
int result = input.read();
input.close();
output.close();
```
Sockets in Java

Client: Send two numbers and obtain their sum

Socket server = new Socket("localhost", 1234);
InputStream input = server.getInputStream();
OutputStream output = server.getOutputStream();
output.write(1);
output.write(2);
int result = input.read();
input.close();
output.close();

Aside

- How do we ensure that client and server fit together?
- We’ll consider an approach later on...
Remote Procedure Call (RPC)

- Procedure call across process and system boundaries (heterogeneous)
- Transparent to client code, but some specialities
  - Error handling: failures of the remote server or network
  - No global variables or side-effects
  - Performance: RPC usually one or more orders of magnitude slower
  - Authentication: may be necessary for RPC
Anatomy of RPC

- Define interface in terms of XDR (eXternal Data Representation)
  - XDR is a data representation format
  - XDR is independent of a particular host language and host architecture (network format)

- **Marshalling**: data conversion from internal representation (host language data) to standardized external representation
  Synonyms: Serialization, pickling

- Stub functions for each remotely callable procedure
  - client code is written in terms of calls to client stubs
  - server code is called from server stubs

- Stub functions generated by RPC compiler from interface definition
## Timeline of an RPC

<table>
<thead>
<tr>
<th>time</th>
<th>client stub</th>
<th>server stub</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>marshall parameters to XDR</td>
<td>invoked by incoming connection</td>
</tr>
<tr>
<td>↓</td>
<td>connect to server</td>
<td>receive parameters</td>
</tr>
<tr>
<td></td>
<td>transmit parameters</td>
<td>unmarshall parameters</td>
</tr>
<tr>
<td></td>
<td><strong>wait for server response</strong></td>
<td>call actual implementation</td>
</tr>
<tr>
<td></td>
<td>receive results</td>
<td>marshall results</td>
</tr>
<tr>
<td></td>
<td>unmarshall results from XDR</td>
<td>exit</td>
</tr>
</tbody>
</table>
Remote Method Invocation (RMI)

- Object-oriented RPC
- Specific to Java
- Implements method calls
  - Dynamic dispatch
  - Access to object identity (this)
- Object serialization (marshalling)
- Access via interfaces
- Easy to use
- Latest variant: asynchronous method invocation
- “Experience has shown that the use of RMI can require significant programmer effort and the writing of extra source code”
  
Simple Object Access Protocol (SOAP)

- Transport protocol specification for method invocations
- Based on HTTP plus extensions\(^3\)
- Encodes information using XML / XML Schema\(^4\)

```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope ...>
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

\(^{3}\)reason: internet security, firewalls

\(^{4}\)reason: standard, extensibility, can be validated
Web Services and WSDL

- Web Service Description Language
- XML-based
- Describes location and protocol of the service
- Main elements:
  - **portType**: Operations of service (cf. RPC program)
  - **message**: Spezification of parameters
  - **types**: Data types (XML Schema)
  - **binding**: Message format and protocol
WSDL 2.0 Example (excerpt)

```xml
<types>
  <xs:element name="getTermRequest" type="xs:string"/>
</xs:element>

<xs:element name="getTermResponse" type="xs:string"/>
</types>

@interface name="glossaryTerms">
  <operation name="getTerm">
    <input messageLabel="In" element="tns:getTermRequest"/>
    <output messageLabel="Out" element="tns:getTermResponse"/>
  </operation>
</interface>
```

- `xs` is the namespace for XML Schema definitions
  `xmlns:xs=http://www.w3.org/2001/XMLSchema`

- `tns` is the targetnamespace for the type definitions
WSDL Example: One-Way Operation

```xml
<types>
  <xs:element name="newTermValues">
    <xs:attribute name="term" type="xs:string" use="required"/>
    <xs:attribute name="value" type="xs:string" use="required"/>
  </xs:element>
</types>

@interface name="glossaryTerms">
  <operation name="setGlossaryTerm">
    <input messageLabel="In" element="tns:newTermValues"/>
  </operation>
</interface>

▶ No return value ⇒ no answer message
```
Further Kinds of Operation

- output-only (no `<input>` params), Example:

  ```xml
  <types>
    <xs:element name="whatTimeValue"/>
    <xs:element name="theTimeValue" type="xs:date"/>
  </types>

  <interface name="Date">
    <operation name="currentTime">
      <input messageLabel="In" element="tns:whatTimeValue"/>
      <output messageLabel="Out" element="tns:theTimeValue"/>
    </operation>
  </interface>
  ```

- “Notification”: output with empty request
Automatic generation of WSDL code

- Translation from WDSL to a client API is tedious
- Parsing XML
- Verifying XML Schema
- Choice of data types
- Binding to HTTP and SOAP possible

⇒ Tools: WSDL2Java
Glimpse on Two Further Component Models
Distributed Component Object Model (DCOM)

- Proprietary format for communication between objects
- Binary standard (not language specific) for “components”
- COM object implements one or more interfaces
  - Described by IDL (Interface Definition Language); stubs etc. directly generated by tools
  - Immutable and persistent
  - May be queried dynamically
- COM services
  - Uniform data transfer IDataObject (clipboards, drag-n-drop, files, streams, etc)
  - Dispatch interfaces IDispatch combine all methods of a regular interface into one method (RTTI)
  - Outgoing interfaces (required interfaces, female connector)
Common Object Request Broker Architecture (CORBA)

- Open distributed object computing infrastructure
- Specified by OMG (Object Management Group)
- Manages common network programming tasks
  - Cross-Language: Normalizes the method-call semantics
  - Parameter marshalling and demarshalling
  - Object registration, location, and activation
  - Request demultiplexing
  - Framing and error-handling
- Extra services
  Component model reminiscent of EJB
Summary

- Distributed Systems Architecture
  - client/server
  - web
  - n-tier (Java EE 6)
- Middleware building blocks