Middleware

Till G. Bay, May 18th, 2006
Looking back…

- Input- and output streams
- Sockets
- Threads
- Client-Server
Same needs

- Communication of data over the network
- Calling methods on remote object
Middleware

- Abstraction
  - *Proxy (stub)* for the remote object, mimics it and redirects invocations to it
  - Globally unique object reference/name
  - Communication handling (most commonly a TCP socket)
Interaction Scheme

Network channel

client

server

consumer

stub

producer

skeleton
Invocation

• Transform messages and send to the «other side»
  – Marshalling

• The «other side»: skeleton
  – Serverside counterpart to stub
  – Extracts request arguments from message (*unmarshaling*) and invokes the server object
  – Marshals return value and sends it to the invoker side, where stub unmarshals it and returns the result to invoker
Stubs and Skeletons in Perspective

• Client side: Stub
  – Offers same interface than server object: mimics the server
  – Usually bound to a single server
  – Marshals the request into a stream of bytes
    • Method id (e.g., name)
    • Arguments
  – Additional features:
    • Caching of values
    • Load balancing
    • Statistics
    • ...

• Server side: Skeleton
  – Represents the server objects
  – Bound to a single server
  – Sometimes several proxies for a server
  – Unmarshals the request and calls the corresponding Method on the server object
  – Additional features:
    • Persistence
    • ...

Chair of Software Engineering
Distributed Objects in perspective

• Object has
  – Interface (abstract type)
  – Implementation (concrete type)
  – Local reference, e.g., value of a monotonically increased counter, memory address

• «Remote» object has
  – Interface for remote invocations
  – Implementation
  – Global reference, e.g., (host id, process id, obj id)
Preview: Further Concepts

• Repositories
  – Reference Repository
    • Find new remote objects (locate objects, i.e., bootstrapping)
  – Interface Repository
    • Discover new remote object types (browse remote types)

• Advanced concepts
  – Dynamic invocations
  – Threading
Java RMI Overview

• Allow distributed Java Objects to interact
  – Through (remote) method invocations
  – Invocations are synchronous (even if there is no reply)
  – Fully integrated into Java language
  – Remote interfaces described through Java interfaces

• Separate compilation
  – Generate Stubs and Skeletons according to interfaces
  – Compile application
Java RMI Architecture

Application

Client

Server

RMI System

Stubs  Skeletons

Remote Reference Layer

Transport
Stub Skeleton Layer

• **Stub**
  - Has same interface than remote object
  - Initializes call to remote object
  - Marshals arguments to stream
  - Passes stream to remote reference layer
  - Unmarshals the return value
  - Informs the remote reference layer that call is complete

• **Skeleton**
  - Unmarshals arguments from the stream
  - Makes up-call to the remote object implementation
  - Marshals the return value or an exception onto the stream
Example

1. Write the interfaces of the remote (i.e., remotely accessible) objects: coarse grained
2. Write the implementations of the remote objects
3. Write other classes involved: fine grained
4. Compile the application with `javac`
5. Generate stubs and skeletons with `rmic`
Example: Declaring a remote interface

- Objects are remotely accessible through their remote interface(s) only.
- Methods to be exported are declared in an interface that extends the `java.rmi.Remote` interface.
- Remote interfaces
  - Must be public
  - All methods must declare `java.rmi.RemoteException` in throws list: represent exceptions due to distribution.
A HelloWorld Remote Interface

```java
import java.rmi.*;

public interface Hello extends Remote {
    public void print() throws RemoteException;
}
```
Implementing a Remote Interface

• Implement the **Remote** interface
  
  – Abstract class
    
    `java.rmi.server.RemoteObject` implements **Remote**
    
    • Remote behavior for `hashCode()`, `equals()` and `toString()`
  
  – Abstract class
    
    `java.rmi.server.RemoteServer` extends `RemoteObject`
    
    • Functions to export remote objects
Implementing a Remote Interface

• Concrete class
    • Non-replicated remote object
    • Support for point-to-point active object references (invocations, parameters, and results) using TCP
    • Inheritance: subclass UnicastRemoteObject

• Note
  – Own exceptions must not subtype RemoteException
import java.rmi.*;
import java.rmi.server.*;

public class HelloImpl extends UnicastRemoteObject
    implements Hello {

    public HelloImpl() throws RemoteException
    { super(); } 
    public void print() throws RemoteException
    { System.out.println("Hello World"); } 
}
Constructing a Remote Object

• The Constructor
  – Calls the no-argument constructor of the \texttt{UnicastRemoteObject} class (implicitly or explicitly)
  – Which exports a \texttt{UnicastRemoteObject}, meaning that it is available to accept incoming requests by listening to calls from clients on an anonymous port
  – Throws \texttt{RemoteException}, since the constructor of \texttt{UnicastRemoteObject} might do so, if the object cannot be exported
    • Communication resources are unavailable
    • Stub class cannot be found, …

• Alternative: Delegation
  – Explicitly export the object
    \texttt{UnicastRemoteObject.exportObject()}

public class HelloServer {

    public static void main(String[] args) {
        
        Hello hello = new HelloImpl();
        // Register object (e.g., naming service)
        // What’s up doc?
        
    }

}
Starting a Client

```
public class HelloClient {

    public static void main(String[] args) {
        ...
        // Lookup object (e.g., naming service)
        Hello hello = ...;
        // Invoke the remote object
        hello.print();
        // That's all folks...
    }
}
```
CORBA Overview

• Object model (with calling convention etc.)
• IDL with generators and compilers
• Object Request Broker (ORB)
• System functions as Object Services
• Application support through Common Facilities / Application Domains
• Conventions (for interfaces and protocols etc.)

• http://www.omg.org
Exercise 1: Mini Discussion

• Discuss the following 2 questions each for 3 minutes with your neighbor:
  1. Which features of RMI are Java specific?
  2. What should be changed to make RMI programming language independant?
RMI vs. CORBA

RMI
- Java only
- Platform independence due to Java
- Easy to use

Using RMI
1. Define java interfaces for remote classes
2. Create and compile implementation of the remote classes
3. Create stub and skeleton classes using the `rmic`
4. Create and compile server application
5. Create and compile client to access remote objects
6. Start RMI registry and server app.
7. Test client

CORBA
- Heterogeneous Systems
- Platform independence due to language independance
- More elaborate architecture

Using CORBA
1. Define IDL interfaces of remote classes
2. Create stub and skeleton classes using `idl`
3. Create and compile implementation of the remote classes
4. Create and compile server application
5. Create and compile client to access remote objects
6. Start server
7. Test client
Exercise 2: Mini Discussion

Discuss the following question for 3 minutes with your neighbor:

What are the advantages of a technology independent component model for distributed applications?
Object Model Architecture: OMA

- OMG’s reference architecture: **Object Management Architecture**

![Diagram of Object Model Architecture](image-url)
OMA

Application Interfaces
Developed for specific application not part of CORBA infrastructure

Object Services
Domain independent interfaces used by many distributed applications. Examples: Naming Service, Trading Service

Common Facilities
Commonly used facilities used in end-user applications. Examples: GUI Library, Internationalization framework

Domain Interfaces
Like Object Services and Common Facilities but targeted to a specific application domain. Examples: Telecommunication, Medical, Financial

ORB
Infrastructure propagating method calls, relating objects to each other.
Object Services

- Base Services as system-wide infrastructure (not all implemented and not all fully specified)
- **COSS** (Common Object Services Specification) (CORBA conforming products must provide these)
- Eventhandling, Persistence,
- Naming, Lifecycle,
- Transactions, Time,
- Security, Licensing,
- Trading, Replication,
- Concurrency, Externalization
Communication of Objects

- ORB Core

Client ▷ Object Request Broker (ORB) ▷ Server

request
ORB: Object Request Broker

Client
- Dynamic Invocation Interface
- Static IDL Stubs
- ORB Interface

Server
- ORB Interface
- Static IDL Skeletons
- Dynamic Skeleton Interfaces
- Object Adapters

Network
- ORB-Core Client side
- ORB-Core Server side
- Interface Repository
- Implementation Repository
Other initial Services

- Collection service
- Concurrency service
- Event service
- Externalization service
- Licensing service
- Life cycle service
- Notification service
- Persistent state service
- Property service
- Query service
- Relationship service
- Security service
- Telecoms log service
- Time service
- Trading object service
- Transaction service
Need a Semester or Master Thesis?

SEmasters: May 30. 2006, IFW E42, 16.00

The Chair of Software Engineering presents the Thesis topics that are available
Article to read

- TSpaces
  