Remote invocation, part II
(case study: Java RMI)

Netzprogrammierung
(Algorithmen und Programmierung V)
Our topics last week

Descriptive models for distributed system design

Physical model

Architectural model

Architectural elements

Communicating entities
- Processes
- Objects
- Components
- Web Services

Inter-process communication
- UDP sockets
- TCP sockets
- Multicast

Indirect communication

Roles and responsibilities
- Architectural styles
  - Client-server
  - Peer-to-peer

Placement
- Multiple server
- Proxy/Cache
- Mobile code

Architectural patterns

Vertical distribution
- Multi-tier
- Thin/Fat Client

Horizontal distribution

Failed model

Security model
Our topics today

Java RMI architecture and its layers

Stub and Skeleton layer
• Proxy design pattern
• Reflections

Remote Reference Layer

Transport Layer
Java RMI architecture
Components of the RMI architecture

- **Client**
  - Object A
  - Proxy for B
  - Remote reference module
  - Communication module

- **Server**
  - Remote object B
  - Skeleton & dispatcher for B’s class
  - Remote reference module
  - Servant
Separation of concerns

RMI architecture is based on one important principle: *the definition of behavior and the implementation of that behavior are separate concerns.*

RMI allows the code that defines the behavior and the code that implements the behavior to remain separate and to run on separate JVMs.
Implementation of the interface

Client
Service Proxy

Server
Service Implementation

«Interface»
Service

RMI
“Magic”
Java RMI architecture

RMI Architecture Layers
Three abstraction layers in the RMI implementation

1. **Stub and Skeleton layer**
   Intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service

2. **Remote Reference Layer**
   Interpret and manage references made from clients to the remote service objects

3. **Transport layer**
   Is based on TCP/IP connections between machines in a network
   Provides basic connectivity, as well as some firewall penetration strategies
RMI Architecture Layers

**Stub and Skeleton Layer**
Stub and Skeleton Layer

RMI uses the **Proxy design pattern**
- Stub class is the *proxy*
- Remote service implementation class is the *RealSubject*

Skeleton is a helper class
- Carries on a conversation with the stub
- Reads the parameters for the method call → makes the call to the remote service implementation object → accepts the return value → writes the return value back to the stub

- (Please note: Since the Java 2 SDK implementation of RMI, the new wire protocol has made skeleton classes obsolete. RMI uses *reflection* to make the connection to the remote service object.)
Proxy design pattern

Motivation
Provide a surrogate or placeholder for another object to control access to it

Implementation
Proxy design pattern: Applications

Virtual Proxies: delaying the creation and initialization of expensive objects until needed, where the objects are created on demand.

Remote Proxies: providing a local representation for an object that is in a different address space. A common example is Java RMI stub objects. The stub object acts as a proxy where invoking methods on the stub would cause the stub to communicate and invoke methods on a remote object (called skeleton) found on a different machine.

Protection Proxies: where a proxy controls access to RealSubject methods, by giving access to some objects while denying access to others.

Smart References: providing a sophisticated access to certain objects such as tracking the number of references to an object and denying access if a certain number is reached, as well as loading an object from database into memory on demand.
RMI with Reflection

Reflection enables Java code

1. to dynamically discover information about the fields, methods and constructors of loaded classes \((\text{at runtime})\), and

2. to use \textit{reflected} fields, methods, and constructors to operate on their underlying counterparts on objects \((\text{within security restrictions})\).

More information:
http://download.oracle.com/javase/tutorial/reflect/
Method Invocation with Java Reflection API

java.lang.reflect.Method class
- provides methods for obtaining the type information for parameters and return value.
- can be used to invoke methods on a given object: java.lang.reflect.Method.invoke()
  - First argument is the object instance on which this particular method is to be invoked
  - Second argument are the method parameters packed in an array
  - Return value is the return object of the method
Using Reflection in RMI

1. Proxy has to marshal information about a method and its arguments into a request.
2. For a method it marshals an object of class `Method` into the request. It then adds an array of `objects` for the method’s arguments.
3. The dispatcher unmarshals the `Method` object and its arguments from request message.
4. The local object reference is obtained with the remote object reference from remote reference module.
5. The dispatcher then calls the `Method` object’s “invoke” method, supplying the target object reference and the array of argument values.
6. After the method execution, the dispatcher marshals the result or any exceptions into the reply message.

- dispatcher is generic and can be used for all classes of remote objects
Remote Reference Layer

RMI Architecture Layers

Remote Reference Layer
Remote Reference Layer

Defines and supports the invocation semantics of the RMI connection

Provides a **RemoteRef object** that represents the link to the remote service implementation object

**JDK 1.1 implementation of RMI**

- Provides a **unicast, point-to-point connection**
- Before a client can use a remote service, the remote service must be instantiated on the server and exported to the RMI system

**Additional since Java 2 SDK implementation of RMI**

- When a method call is made to the proxy for an **activatable object**, RMI determines if the remote service implementation object is dormant
- If yes, RMI will instantiate the object and restore its state from a disk file
RMI Architecture Layers

**Transport Layer**
Transport Layer

The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.

On top of TCP/IP, RMI uses a wire level protocol called **Java Remote Method Protocol (JRMP)**. JRMP is a proprietary, stream-based protocol that is only partially specified in two versions:

- First version was released with the JDK 1.1 version of RMI and required the use of Skeleton classes on the server.
- Second version was released with the Java 2 SDK. It has been optimized for performance and does not require skeleton classes.
Naming Remote Objects

Java RMI architecture

Naming Remote Objects
Naming Remote Objects

How does a client find a RMI remote service?

RMI can use different naming and directory services such as Java Naming and Directory Interface (JNDI)

RMI includes a simple service called the **RMI Registry**, *rmiregistry*.

The RMI Registry runs on each machine that hosts remote service objects and accepts queries for services, by default on port 1099.
Naming Remote Objects (cont.)

Server program

1. Creates a remote service by creating a local object
2. Export object to RMI
3. Register object in the RMI Registry

Client program

Queries RMI Registry by method lookup()

\[ \text{rmi://<host_name>:<name_service_port>}/<service_name> \]
Java RMI architecture

**Factory Design Pattern**
Examples

1. http://www.flickr.com/photos/23065375@N05/

Factory Design Pattern

1. Registration
2. Client requests Factory reference
3. Registry returns FactoryImpl reference
4. Client invokes method on FactoryImpl
5. FactoryImpl returns a ProductImpl reference
6. Client invokes method on ProductImpl
How could the examples be implemented in Java RMI?

1. `public interface AccountManager extends Remote {
   public Account createAccount() throws RemoteException;
}

   `public interface Account extends Serializable {
   public depositMoney();
   ...
   }

2. `public interface Librarian extends Remote {
   public LibraryCard createLibraryCard() throws RemoteException;
   public Loanable getLoanable() throws RemoteException;
}

   `public interface LibraryCard extends Serializable {...
   public interface Loanable extends Serializable {...

Barry Linnert, linnert@inf.fu-berlin.de, Netzprogrammierung WS 2015/16
Java RMI architecture

Dynamic code downloading using RMI
Dynamic code loading

The Java VM **interprets bytecode or compiles it on the fly and can load and run code dynamically.**

Bytecode loading is encapsulated in a **ClassLoader**
- Developers can write custom ClassLoaders
- Can load bytecode from anywhere
- Specifically from the network

**URLClassLoader** ("out of the box")
- Loads from a Uniform Resource Locator (URL) (per file://, ftp://, http://)

Well-known example are Java Applets.

```xml
<applet height=100 width=100 codebase="myclasses/" code="My.class">
<param name="ticker">
</applet>
```
What is the codebase

A codebase can be defined as a source, or a place, from which to load classes into a Java virtual machine.

CLASSPATH is your "local codebase", because it is the list of places on disk from which you load local classes. When loading classes from a local disk-based source, your CLASSPATH variable is consulted.

So just as CLASSPATH is a kind of "local codebase", the codebase used by e.g. applets and remote objects can be thought of as a "remote codebase". But other codebases must be supported by a ClassLoader.

ClassLoaders form a hierarchy

- Asks parent ClassLoader for class first, if not found then ask other
- Loads the class itself
How codebase is used in applets

1. Initialize ClassLoader for Applet
2. ClassLoader first looks into CLASSPATH
3. If class not found then load from URL
4. Same for other classes the Applet needs
Codebases for Java RMI

1. Server registers a remote object, bound to a name

2. Client makes a Naming lookup call

3. The registry returns an instance of the remote object’s stub

4. Client requests the stub class from the codebase

5. The HTTP server returns the remote object’s stub class
RMI client making a remote method call
Reminder: RMI Parameter Passing

In remote method invocations:

• If formal parameter is of type `Remote`: Pass a stub

• Else, if formal parameter is of type `Serializable`: Pass a serialized object

• Else, throw a MarshalException
Using codebase in RMI for more than just stub downloading

The client may pass to the remote object:

1. Primitive data types
2. Types that are found in the remote CLASSPATH
3. Types unknown to the remote object
   - Implementation of known interface
   - Subclass of known type

How can the latter case be handled?
Using codebase in RMI for more than just stub downloading (cont.)

6. Client makes a remote method call, passing a subtype unknown to the remote object as a remote method parameter

Remote object instance

7. Remote object downloads the subtype's class definition from the client-specified codebase

URL location (file, ftp, or http)
Remote method invocation

Summary
Summary: Steps to develop an RMI application

1. Design and implement the components of a distributed application
   • Define the remote interface(s)
   • Implement the remote object(s)
   • Implement the client(s)

2. Compile sources and generate stubs (and skeletons)

3. Make required classes network accessible

4. Run the application

(for details see last lecture with whiteboard example)
We have we learned?

- Core concepts of the RMI infrastructure
- The three RMI architecture layers
- Proxy design pattern
- Reflections
- Process how a client finds a RMI remote service
- Factory design pattern
- Dynamic code loading in RMI
Questions

• How does RMI achieve that the definition of behavior and the implementation of that behavior are separate concerns?
• Explain the proxy design pattern and how RMI implements it? Name four typical applications of this pattern.
• Explain how dynamic (Java) reflection with a generic dispatcher can be used in RMI instead of static skeletons.
• How can a client find a named remote object in RMI?
• What is a codebase and how does dynamic code loading work with it?
Our topics next week

Descriptive models for distributed system design

Physical model

Architectural model

Architectural elements

Communicating entities
- Processes
- Objects
- Components
- Web Services

Communication paradigm
- Inter-process communication
  - UDP sockets
  - TCP sockets
  - Multicast
- Indirect communication
  - Remote invocation

Roles and responsibilities
- Architectural styles
  - Client-server
  - Peer-to-peer

Placement
- Multiple server
- Proxy/Cache
- Mobile code

Architectural patterns
- Vertical distribution
  - Multi-tier
  - Thin/Fat Client
- Horizontal distribution

Interaction model

Interaction model

Failure model

Security model
Next class

Indirect Communication
References

Main resource for this lecture:

Oracle Remote Method Invocation (RMI):
http://java.sun.com/developer/onlineTraining/rmi/RMI.html#BooksAndArticles

Dynamic Code Loading:
http://download.oracle.com/javase/1.4.2/docs/guide/rmi/codebase.html

Security in Java RMI
http://www.eg.bucknell.edu/~cs379/DistributedSystems/rmi_tut.html#secure