Distributed Systems

6. Remote Method Invocation

Werner Nutt

Remote Method Invocation

6.1 Communication between Distributed Objects

1. **Communication between Distributed Objects**
2. Java RMI
3. Dynamic Code

Middleware

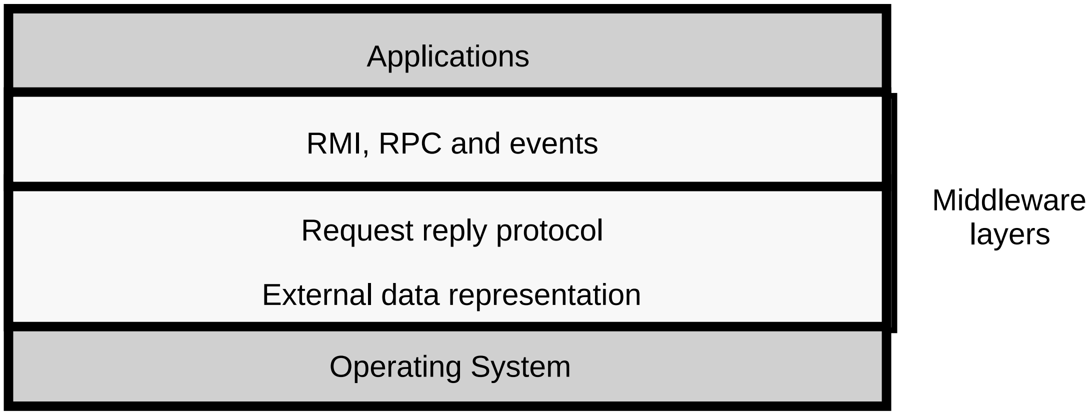
- Middleware offers an infrastructure that enables application processes to communicate with each other
- Processes issue requests to the transportation layer
  (i.e., the application takes the initiative, not the middleware)
- Applications access the middleware via APIs, e.g.,
  - creation and manipulation of sockets
- Integration into programming languages
  - remote procedure call (RPC)
  - remote method invocation (RMI)
- For higher level APIs, data has to be transformed before it can be shipped ("data marshalling")
- Protocols for Client/Server Interaction ("Request/Reply")

Why Middleware?

Distributed computing environments are heterogeneous:

- Networks
  - ATM, Ethernet, etc. have different protocols
- Computer hardware
  - data types (integers) can be represented differently
- Operating systems
  - e.g., TCP module can be part of OS (Unix/Linux) or not
- Programming languages
  - e.g., different paradigms (functional, OO, etc.)
  - e.g., data structures (arrays, records) can be represented differently
- Applications implemented by different developers

Middleware Hides Heterogeneity



Applications

RMI, RPC and events

Request reply protocol

External data representation

Operating System

Middleware layers

Middleware Characteristics

- Location transparency
  - client/server need not know their location
- Sits on top of OS, independent of
  - Communication protocols:
    use abstract request-reply protocols over UDP, TCP
  - Computer hardware:
    use external data representation e.g. CORBA CDR
  - Operating system:
    use e.g. socket abstraction available in most systems
  - Programming language:
    e.g. CORBA supports Java, C++

Middleware Programming Models

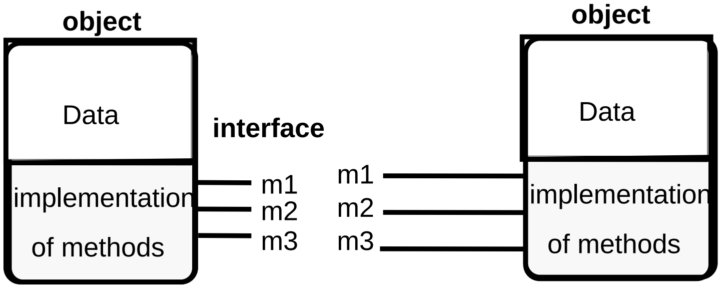
Commonly used models:

- Distributed objects and remote method invocation *(Java RMI, Corba)*
- Remote Procedure Call *(Web services)*
- Remote SQL access *(JDBC, ODBC)*
- Distributed transaction processing

CORBA (old):

- provides remote object invocation between
  - a client program written in one language and
  - a server program written in another language
- commonly used with C++

Objects

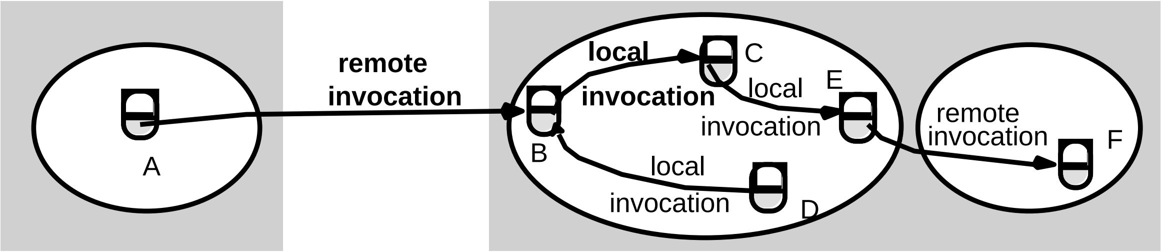


- Object = data + methods
  - logical and physical encapsulation
  - accessed by means of references
  - first class citizens, can be passed as arguments
- Interaction via interfaces
  - define types of arguments and exceptions of methods

The Object Model

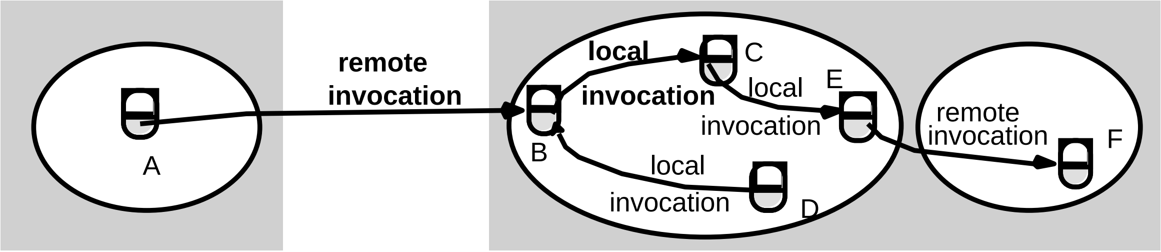
- Programs are *(logically and physically)* partitioned into objects
  ➔ distributing objects natural and easy
- Interfaces
  – the only means to access data
  ➔ make them remote
- Actions
  – via method invocation
  – interaction, chains of invocations
  – may lead to exceptions ➔ part of interface
- Garbage collection
  – reduces programming effort, error-free *(Java, not C++)*
  ➔ generalize to distributed garbage collection

The Distributed Object Model: Ideas



- Objects are distributed
  – client-server relationship at the object level
- Extended with
  – Remote interfaces
  – Remote Method Invocation (RMI)
  – Remote object references

The Distributed Object Model: Principles



- Each process contains objects, some of which can receive remote invocations, others only local invocations
- Objects that can receive remote invocations are called *remote objects*
- The *remote interface* specifies which methods can be invoked remotely
- Objects need to *know* the *remote object reference* of an object in another process in order to invoke its methods ➔ *How do they get it?*

Remote Object References

- Object references
  - used to access objects, which live in processes
  - can be passed as arguments and results
  - can be stored in variables
- Remote object references
  - object identifiers in a distributed system
  - must be unique in space and time
  - error returned if accessing a deleted object
  - can allow relocation (see CORBA)

Remote Object Reference

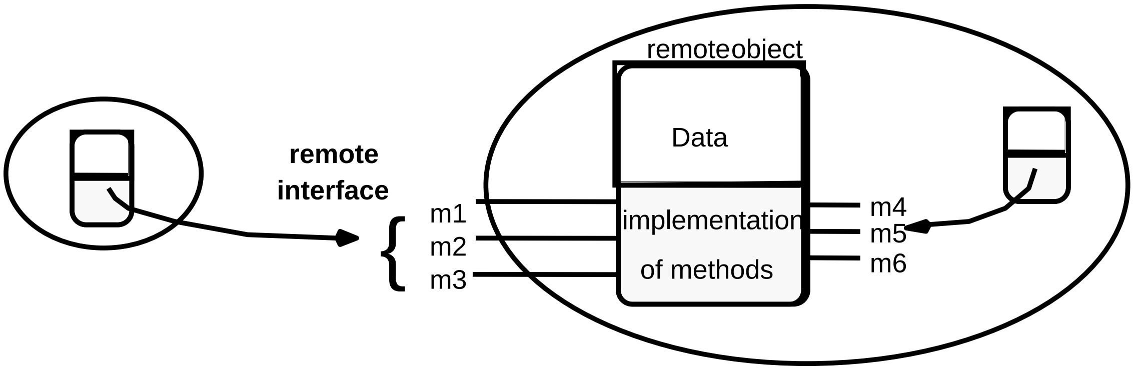
- Construct unique remote object reference
  - – IP address, port, interface name
  - – time of creation, local object number (new for each object)
- Use in the same way as local object references
- If used as address
  - ➔ cannot support relocation

32 bits	*32 bits*	*32 bits*	*32 bits*	
Internet address	port number	time	object number	interface of remote object

Remote Interfaces

- Specify externally accessible methods
  - – no direct references to variables *(no global memory)*
  - – local interface is separate
- Parameters
  - – input, output or both (no output parameters in Java ➔ *why?*)
  - – call by value/by copy and call by reference
- No pointers
  - – but references
- No constructors
  - – but factory methods

A Remote Object and its Interface



- CORBA: Interface Definition Language (IDL)
- Java RMI: like other interfaces, extends class ***`Remote`***

Handling Remote Objects

- Exceptions *(Java:* ***`RemoteException`****)*
  - – raised in remote invocation
  - – clients need to handle exceptions
  - – timeouts in case server crashed or too busy

- Garbage collection
  - – distributed garbage collection may be necessary
  - – combined local and distributed collector
  - – cf. Java reference counting
    (remote object knows in which processes live proxies, extra communication to inform server about creation and deletion of proxies)

RMI Issues

- Local invocations
  - executed exactly once

- Remote invocations
  - via Request-Reply
  - may suffer from communication failures!
  - ➔retransmission of request/reply
  - ➔message duplication, duplication filtering
  - ➔no unique semantics…

Invocation Semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Maybe Invocation

- Remote method
  - may execute once or not at all, invoker cannot tell
  - useful only if failures are rare
- Invocation message lost...
  - method not executed
- Result not received...
  - was method executed or not?
- Server crash...
  - before or after method executed?
  - if timeout, result could be received after timeout …

At-least-once Invocation

- Remote method
  - invoker receives result (executed at least once) or exception (no result received)
  - retransmission of request messages
- Invocation message retransmitted …
  - method may be executed more than once
  - arbitrary failure (wrong result possible)
  - method must be idempotent (repeated execution has the same effect as a single execution) to be acceptable
- Server crash...
  - dealt with by timeouts, exceptions

At-most-once Invocation

- Remote method
  - invoker receives result (executed once) or exception (no result)
  - retransmission of reply and request messages
  - receiver keeps history with results *(how long?)*
  - duplicate filtering
- Best fault-tolerance ...
  - arbitrary failures prevented if method called at most once
- Used by CORBA and Java RMI

  (however, based on TCP)

Transparency of RMI

- Should remote method invocation be same as local?
  - same syntax, see Java RMI (keyword *Remote*)
  - need to hide:
    - data marshalling
    - IPC calls
    - locating/contacting remote objects
- Problems
  - different RMI semantics? susceptibility to failures?
  - protection against interference in concurrent scenario?
- Approaches (Java RMI)
  - transparent, but express differences in interfaces
  - provide recovery features (IPC over TCP)

Remote Method Invocation

6.2 Java RMI

1. Communication between Distributed Objects
2. **Java RMI**
3. Dynamic Code

Hello World: Remote Interface

```
import java.rmi.*;

public interface HelloInterface extends Remote {
  /*
   * Remotely invocable method,
   * returns the message of the remote object,
   *                  such as "Hello, world!"
   * throws a RemoteException
   *                  if the remote invocation fails
   */
  public String say() throws RemoteException;
}
```

Hello World: Remote Object

```
import java.rmi.*;
import java.rmi.server.*;

public class Hello extends UnicastRemoteObject
                   implements HelloInterface {
  private String message;
  /* Constructor for a remote object
   * Throws a RemoteException if exporting the object fails
   */
  public Hello (String msg) throws RemoteException {
    message = msg;
  }
  /* Implementation of the remotely invocable method
   */
  public String say() throws RemoteException {
    return message;
  }
}
```

Hello World: Server

```
import java.io.*;
import java.rmi.*;

public class HelloServer{
  /*
   * Server program for the "Hello, world!" example.
   */
  public static void main (String[] args) {
    try {
      Naming.rebind ("SHello",
             new Hello ("Hello, world!"));
      System.out.println ("HelloServer is ready.");
    } catch (Exception e) {
      System.out.println ("HelloServer failed: " + e);
    }
  }
}
```

Hello World: Client

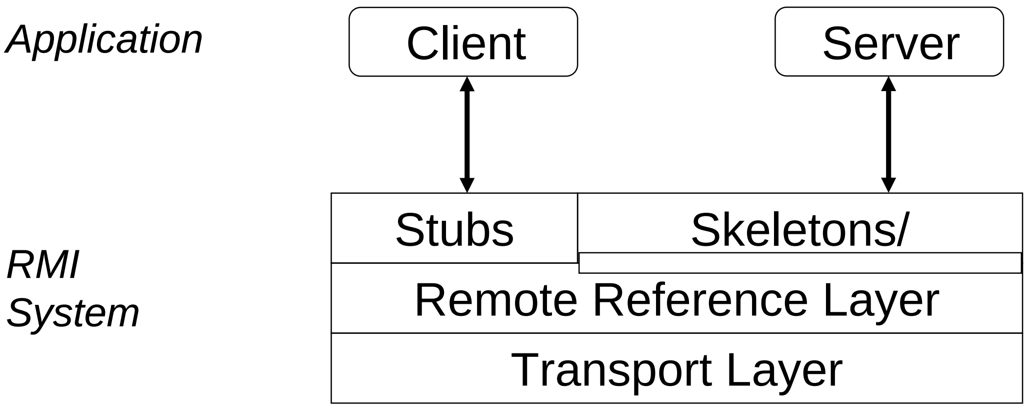
```
import java.io.*;
import java.rmi.*;

public class HelloClient{
  /*
   * Client program for the "Hello, world!" example
   */
  public static void main (String[] args) {
    try {
      HelloInterface hello = (HelloInterface)
         Naming.lookup ("//russel.inf.unibz.it/SHello");
      System.out.println (hello.say());
    } catch (Exception e) {
      System.out.println ("HelloClient failed: " + e);
    }
  }
}
```

Hello World: Compilation

- On the server side
  - start the RMI registry: **`rmiregistry &`**
    (Standard port number 1099)
  - compile with Java compiler: **`HelloInterface.java, Hello.java, HelloServer.java`**
  - compile with RMI compiler: **`Hello`**
    - command: **`rmic Hello`**

    ➔ produces class **`Hello_Stub.class`**
    (previously **`Hello_Stub`** and **`Hello_Skel`**)
- On the client side
  - compile **`HelloClient`**
    - class **`HelloInterface.class`** needs to be accessible

RMI Architecture



Application

Client

Server

RMI System

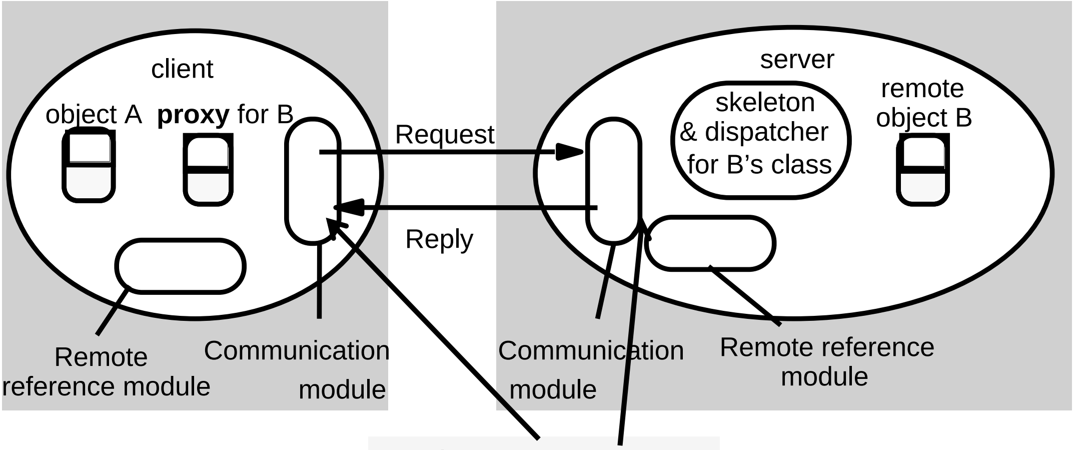
Stubs

Skeletons/

Remote Reference Layer

Transport Layer

Implementation of RMI



client

object A **proxy** for B

Request

Reply

Remote reference module

Communication module

server

skeleton & dispatcher for B's class

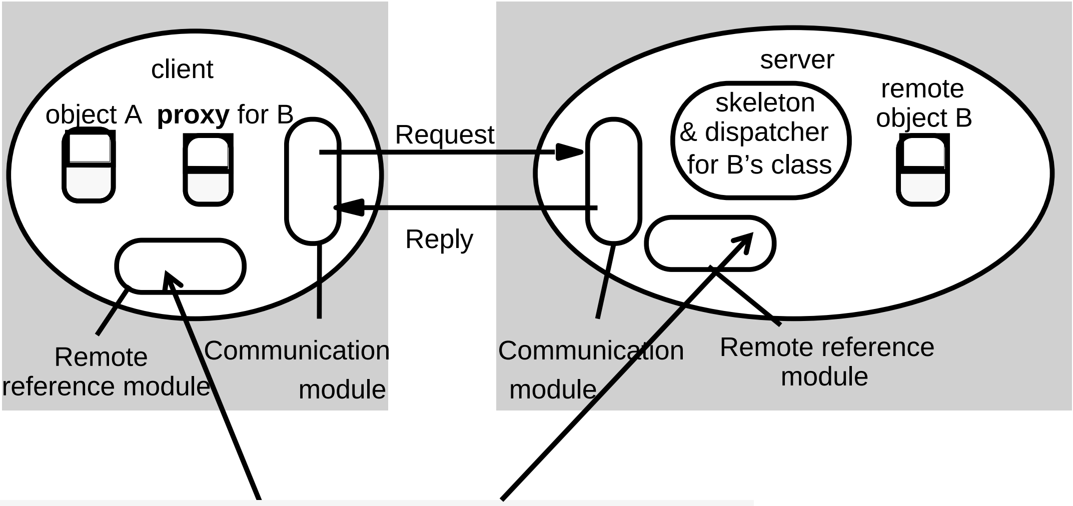
remote object B

Communication module

Remote reference module

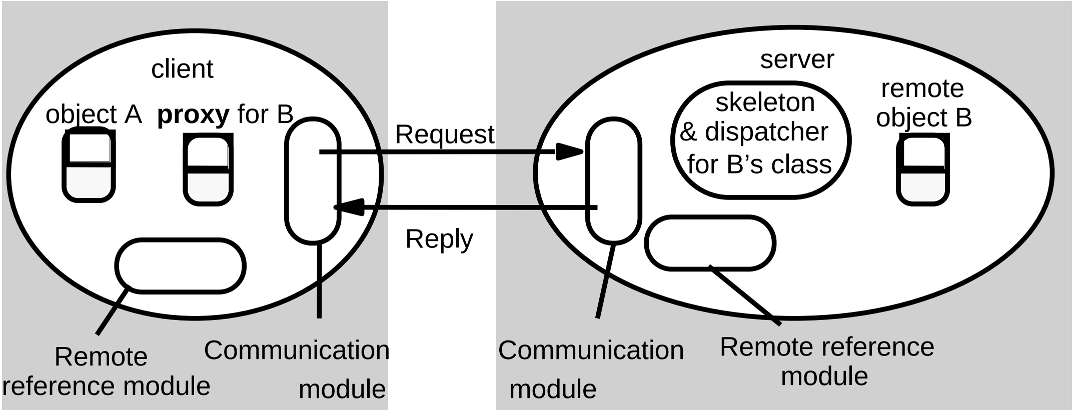
Carries out Request-reply protocol, responsible for semantics.
In Java RMI realized by RemoteRef

Implementation of RMI



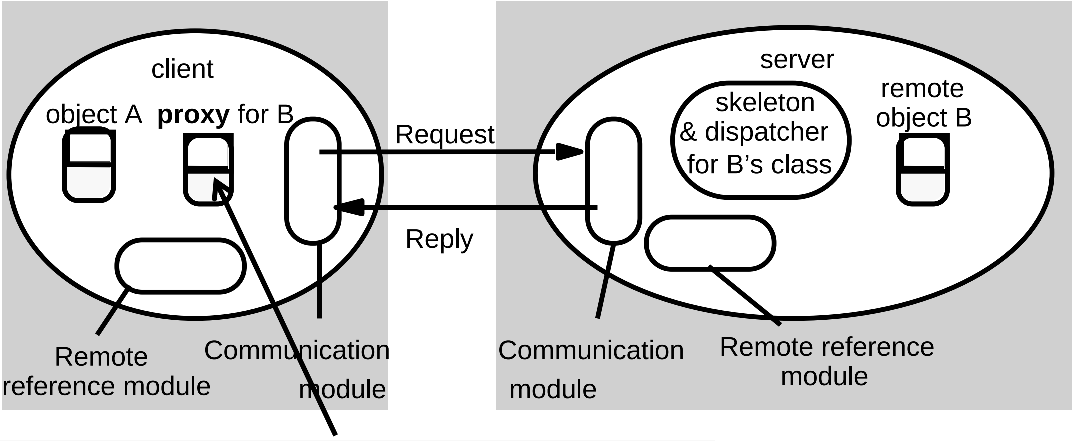
Translates between local and remote object references,
creates remote object references.
Uses remote object table
(relating remote and local object references, plus proxies)

Implementation of RMI



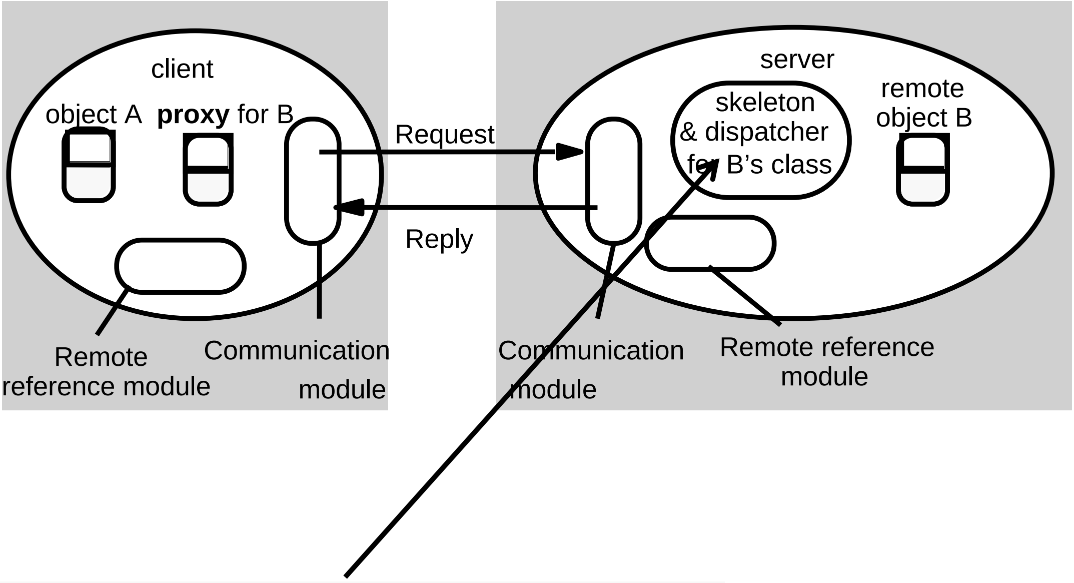
RMI software - between application level objects and communication and remote reference modules
(according to JRMP v1.1)

Implementation of RMI



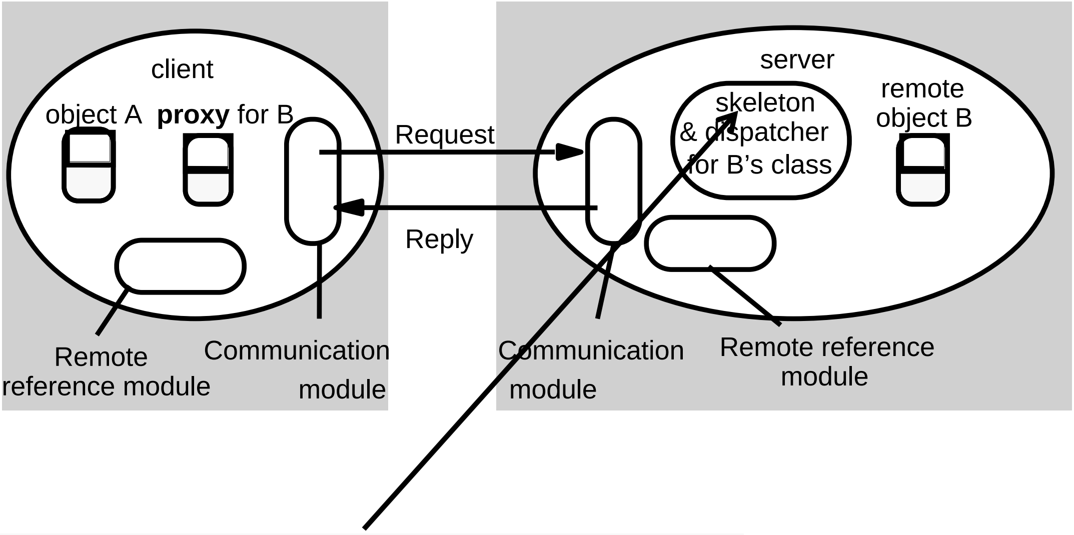
Proxy - makes RMI transparent to client. Class implements Remote interface. Marshals requests and unmarshals results. Forwards request.

Implementation of RMI



Dispatcher - gets request from communication module and invokes method in skeleton (using *methodID* in message).

Implementation of RMI



Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.

Communication Modules

- Reside in client and server virtual machine
- Carry out Request-Reply jointly
  - implement given RMI semantics (at least once, at most once, exactly once)
- Server's communication module
  - calls Remote Reference Module to convert remote object reference to local

Remote Reference Module

- Creates remote object references and proxies
- Translates remote to local references (object table):
  - correspondence between remote and local object references (proxies)
- Called by RMI software
  - when marshalling/unmarshalling

RMI Software Architecture

- Proxy/Stub
  - behaves like local object to client
  - forwards requests to remote object
- Dispatcher
  - receives request
  - selects method and passes on request to skeleton
- Skeleton
  - implements methods in remote interface
    - unmarshals data, invokes remote object
    - waits for result, marshals it and returns reply

Hello Skeleton/1

```
// Skeleton class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class Hello_Skel
    implements java.rmi.server.Skeleton
{
    private static final java.rmi.server.Operation[] operations = {
      new java.rmi.server.Operation("java.lang.String say()")
    };

    private static final long interfaceHash = -7469971880086108926L;

    public java.rmi.server.Operation[] getOperations() {
      return (java.rmi.server.Operation[]) operations.clone();
    }
```

Hello Skeleton/2

```
public void dispatch(java.rmi.Remote obj, java.rmi.server.RemoteCall call, int opnum, long hash)
     throws java.lang.Exception
   {
     if (hash != interfaceHash)
         throw new java.rmi.server.SkeletonMismatchException("interface hash mismatch");

     Hello server = (Hello) obj;
     switch (opnum) {
     case 0: // say()
     {
         call.releaseInputStream();
         java.lang.String $result = server.say();
         try {
               java.io.ObjectOutput out = call.getResultStream(true);
               out.writeObject($result);
         } catch (java.io.IOException e) {
               throw new java.rmi.MarshalException("error marshalling return", e);
         }
         break;
     }
}}}
```

Hello Stub/1

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class Hello_Stub
    extends java.rmi.server.RemoteStub
    implements HelloInterface, java.rmi.Remote
{
    private static final java.rmi.server.Operation[] operations = {
      new java.rmi.server.Operation("java.lang.String say()")
    };

    private static final long interfaceHash = -7469971880086108926L;

    // constructors
    public Hello_Stub() {
      super();
    }
    public Hello_Stub(java.rmi.server.RemoteRef ref) {
      super(ref);
    }
```

Hello Stub/2

```
 // methods from remote interfaces

 // implementation of say()
    public java.lang.String say()
      throws java.rmi.RemoteException
    {
      try {
          java.rmi.server.RemoteCall call = ref.newCall((java.rmi.server.RemoteObject) this,
      operations, 0, interfaceHash);
          ref.invoke(call);
          java.lang.String $result;
          try {
                java.io.ObjectInput in = call.getInputStream();
                $result = (java.lang.String) in.readObject();
          } catch (java.io.IOException e) {
                throw new java.rmi.UnmarshalException("error unmarshalling return", e);
          } catch (java.lang.ClassNotFoundException e) {
                throw new java.rmi.UnmarshalException("error unmarshalling return", e);
          } finally {
                ref.done(call);
          }
          return $result;
```

Hello Stub/3

```
      } catch (java.lang.RuntimeException e) {
          throw e;
      } catch (java.rmi.RemoteException e) {
          throw e;
      } catch (java.lang.Exception e) {
          throw new java.rmi.UnexpectedException("undeclared checked exception", e);
      }
    }
}
```

Hello Stub/1

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

public final class Hello_Stub
    extends java.rmi.server.RemoteStub
    implements HelloInterface, java.rmi.Remote
{
    private static final long serialVersionUID = 2;

    private static java.lang.reflect.Method $method_say_0;

    static {
      try {
          $method_say_0 = HelloInterface.class.getMethod("say", new java.lang.Class[] {});
      } catch (java.lang.NoSuchMethodException e) {
          throw new java.lang.NoSuchMethodError(
                "stub class initialization failed");
      }
    }
```

Hello Stub/2

```
// constructors
   public Hello_Stub(java.rmi.server.RemoteRef ref) {
     super(ref);
   }
```

HelloStub/3

```
// methods from remote interfaces

   // implementation of say()
   public java.lang.String say()
     throws java.rmi.RemoteException
   {
     try {
         Object $result = ref.invoke(this, $method_say_0, null, -3164833839299227514L);
         return ((java.lang.String) $result);
     } catch (java.lang.RuntimeException e) {
         throw e;
     } catch (java.rmi.RemoteException e) {
         throw e;
     } catch (java.lang.Exception e) {
         throw new java.rmi.UnexpectedException("undeclared checked exception", e);
     }
   }
}
```

The Methods of the Naming Class

- **`void rebind (String name, Remote obj)`**
  - This method is used by a server to register the identifier of a remote object by name
- **`void bind (String name, Remote obj)`**
  - This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.
- **`void unbind (String name, Remote obj)`**
  - This method removes a binding.
- **`Remote lookup (String name)`**
  - This method is used by clients to look up a remote object by name. A remote object reference is returned.
- **`String [] list()`**
  - This method returns an array of Strings containing the names bound in the registry.

Exercise: Callback

Write a chat version where

- the server has
  - a **`Multicaster`** object with method **`send(String)`**
- each client has
  - a **`Display`** object with method **`show(String)`**
- both classes and methods are remote.

Clients invoke **`send`** and the server invokes **`show`**.

Sending a string means showing it on all displays.

How can one implement this?

Remote Method Invocation

6.3 Dynamic Code

1. Communication between Distributed Objects
2. RMI
3. **Dynamic Code**

Parameter Passing

Remote methods can have arguments and return results

- arguments: client → server
- results: server → client

Local case

- Parameters are passed by value (if atomic) or by reference

Remote case

- Atomic values: by value
- Remote objects: by remote reference (represented by stub/proxy)
- Other objects: must be **`Serializable`**! Then by copy. Exception if not serializable (cannot be "marshalled")

Dynamic Code Downloading

A client

- holds a remote reference to an instance of a remote interface
- needs stub class for the referenced remote object
- needs classes for arguments and return values of remote methods

Where should these classes come from?

- client stores all possible classes locally (bad because ...)
- client retrieves classes when needed from server host

Example: Generic Echo Server

Server: exports generic method

```
public <T> T doEcho(T input) throws RemoteException;
```

that is, for any type T, echo an object of the same type as the input

Client: invokes **doEcho** with a type unknown to the server

Shows same problem as *compute server*,
which accepts tasks to compute results of arbitrary types

```
public <T> T execute(Task<T> task) ...
```

Echo Interface

```
/* Similar in spirit to HelloWorld */

import java.rmi.*;

public interface EchoInterface extends Remote
  {

    public <T> T doEcho(T input)
                    throws RemoteException;

}
```

Echo Remote Object

```
import java.rmi.*;
import java.rmi.server.*;

public class Echo extends UnicastRemoteObject
                  implements EchoInterface {

  public Echo () throws RemoteException {
      super();
  }

  public <T> T doEcho(T input)
                  throws RemoteException {
    return input;
  }
}
```

Constructor

echoes its input

Echo Server

```
import java.io.*;
import java.rmi.*;

public class EchoServer{

  public static void main (String[] argv) {

    if (System.getSecurityManager() == null) {
    System.setSecurityManager(new SecurityManager());
      }

    try {
      Naming.rebind ("//localhost/Echo", new Echo());
      System.out.println ("Echo Server is ready.");
    } catch (Exception e) {
      System.out.println ("Echo Server failed: " + e);
    }
  }
}
```

The security manager is is new!

Server Classes

A client using the **Echo** object needs two server classes

EchoInterface.class: at compile time

- must be known by developers and made available, e.g., at URL,

Echo_Stub.class: at runtime

- depends on *implementation*, e.g., Echo could implement > 1 interfaces
- developers on server side may create *new classes* that implement **EchoInterface**
- best *downloaded automatically* for a remote reference

⇒ remote reference should contain info about *stub location*

Codebases

Locations where server and client can make available classes for each other

Described by URLs, e.g.,

- codebase=
  http://www.inf.unibz.it/~nutt/classes/EchoServerCode/
- codebase=
  file:/home/nutt/public_html/classes/EchoServerCode/

Classes from a codebase are retrieved

- by contacting a web server
- by accessing them on a common file system

Codebase Annotations

If a Java application finds a class in a codebase,
then it annotates
 - references to
 - copies of
instances that class with the codebase.

For example,

- the RMI registry annotates references
- a client annotates serialized copies

A codebase is defined as the value of the property

- **`java.rmi.server.codebase`**,

Usage

- **`java ... -Djava.rmi.server.codebase=<codebase> ...`**

Security

Code downloaded from other sites can be harmful

In Java one can:

- define security policies
- set up a security manager in an application
- let the manager check whether operations satisfy the policies

Security Policies: Examples

```
grant {
    permission java.security.AllPermission;};
```

Allow anyone to do anything

```
grant
  codeBase "http://www.foo.net/nice/classes/" {
    permission java.security.AllPermission;};
```

Allow code from a specific codebase to do anything

Policy Files and Properties

Policies

- are stored in files, e.g. clientPolicy.pol

- are assigned to properties, e.g.,

```
java ... -Djava.security.policy
             = clientPolicy.pol
       ...
```

Echo Client Sending a String

```
import java.rmi.*;
import java.io.*;

public class EchoClientString {

  public static void main (String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());}

    try {
      EchoInterface echo =
        (EchoInterface) Naming.lookup
                   ("//localhost/Echo");
      System.out.println (echo.doEcho(args[0]));
    } catch (Exception e) {
      System.out.println ("EchoClientString exception: " + e);
    }
  }
}
```

Starting the Server

```
java -Djava.rmi.server.codebase
      =file:/Users/nutt/Java/classes/EchoServerCode/
     -Djava.security.policy=serverPolicy.pol
     EchoServer
```

Note that here

- the interface class and the Echo stub class are in the directory **`EchoServerCode`**
- the security policy of the server is defined in the file **`serverPolicy.pol`**

Don't forget the backslash at the end of ***`.../EchoServerCode/`*** *!*

Compiling and Starting the Client

```
javac –cp .:/Users/nutt/Java/classes/EchoInterface/
      EchoClientString.java
```

Note here

- the interface class is in **`EchoInterface`**
- the class path contains two directories

Starting the Client

```
java -cp .:/Users/nutt/Java/classes/EchoInterface/
     -Djava.security.policy=clientPolicy.pol
     EchoClientString
     'Hello!'
```

Note here

- we use the same class path for the interface
- the stub is downloaded from the server codebase ...
- ... if the security policy allows this
- the string 'Hello!' is echoed

Summary So Far

The client can download classes from the server side

- from the common file system
- from a web server

The client's security policy has to allow this

We have not seen yet

- the server downloading from the client

A Wrapper Class for Strings (Just for the Example)

```
import java.io.*;

public class MyString implements Serializable{

    String myString;

    public MyString(String string) {
      myString = string; }

    public String getString() {
      return myString; }
}
```

If a client sends this, the server needs more info …

Server Receiving and Sending MyStrings

```
import java.rmi.*;
import java.io.*;

public class EchoClientMyString {

  public static void main (String[] args) {

            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new SecurityManager());
            }
    try {
      EchoInterface echo =
        (EchoInterface) Naming.lookup("//localhost/Echo");
      MyString input = new MyString(args[0]);
      MyString output = echo.doEcho(input);
      System.out.println (output.getString());
    } catch (Exception e) {
      System.out.println ("EchoClientString exception: " + e);
    }
  }
}
```

The server
- receives a MyString object

The server
- returns a MyString

Starting the MyString Client

```
java -cp .:/Users/nutt/Java/classes/EchoInterface/
     -Djava.rmi.server.codebase
        =file:/Users/nutt/Java/classes/EchoClientCode/
     -Djava.security.policy
        =clientPolicy.pol
     EchoClientString
     'Hello!‘
```

Note:

- the client classes that the server needs are in
  .../EchoClientCode/
- the client has a codebase property
- every MyString copy will be annotated with the codebase
- the server can download the classes of the client

Summary

Java RMI

- implements a remote object model
- provides a much more abstract view of interoperating processes than socket communication
- is based on TCP, but hides this
- allows code to be downloaded at runtime, using the Web mechanism (URLs and Web servers)
- is powerful on intranets, but is often stopped by firewalls
- can tunnel through firewalls, but at a significant cost

References

In preparing the lectures I have used several sources.

The main ones are the following:

Books:

- Coulouris, Dollimore, Kindberg. Distributed Systems – Concepts and Design (CDK)

Slides:

- Marco Aiello, course on Distributed Systems at the Free University of Bozen-Bolzano
- Andrew Tanenbaum, Slides from his website
- CDK Website
- Marta Kwiatkowska, U Birmingham, slides of course on DS
- Ken Baclawski, Northeastern University