

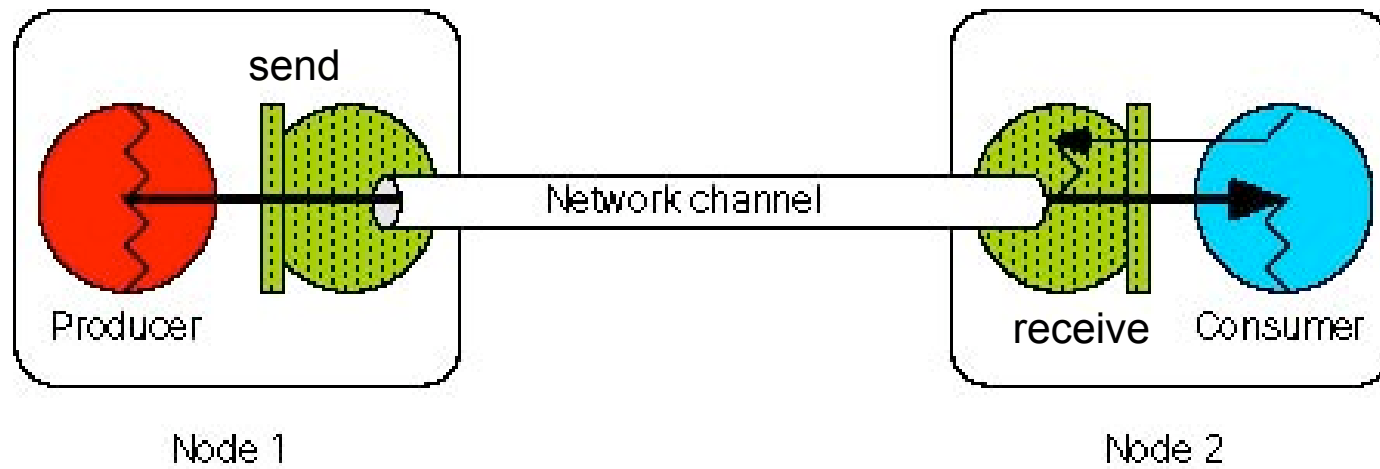
Department of Computer Science

PURDUE
UNIVERSITY

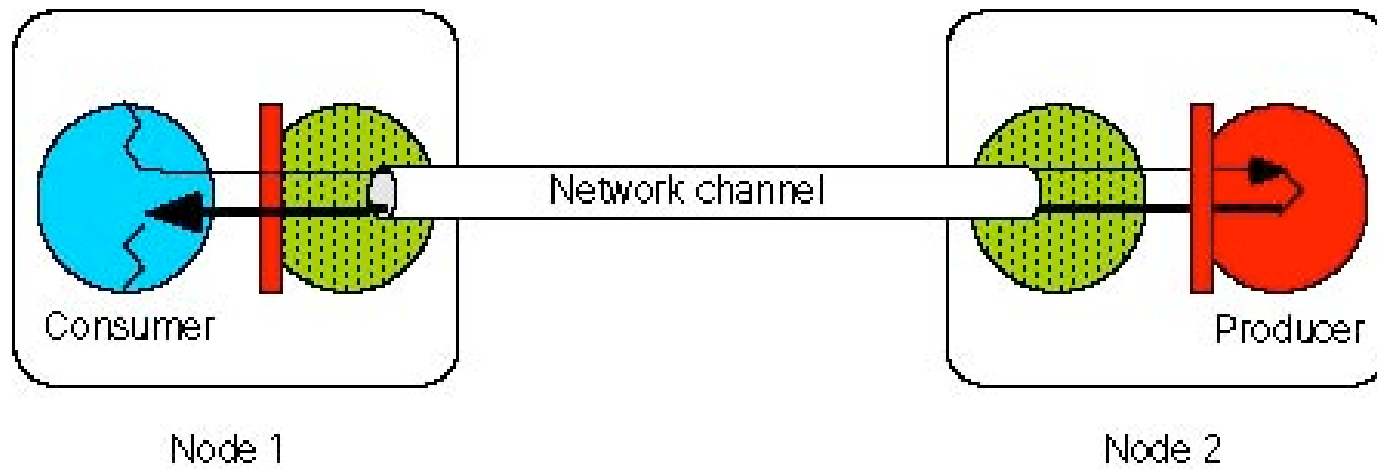
CS505: Distributed Systems

Lecture 9: Remote Method Invocations

Channel



Remote Method Invocation



Outline

- ▶ Using `send`, `receive`, object serialization tedious
- ▶ Higher-level abstraction, well-integrated with objects
 - E.g., Argus, CLU, Modula-3, Obliq, Java
- ▶ Main concept: *proxy (stub)* mimicks remote object
 - Type “related” to the type of the remote object (e.g., subtype)
 - Encapsulates
 - “Location” of the server object, e.g., globally unique object reference/name
 - “Connection”, most commonly a TCP socket
 - Transforms invocations to messages, and sends to the other side (“marshaling”)

Server Side

▶ ***Skeleton* (server-side stub)**

- Server-side counterpart to the 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

▶ **Connection**

- **Delegation:** skeleton is separate object
 - Associated with the effective server object (*binding* is usually explicit in application)
- **Inheritance:** an instance of a supertype of the server object
 - Developer subclasses a skeleton class generated for the server object interface

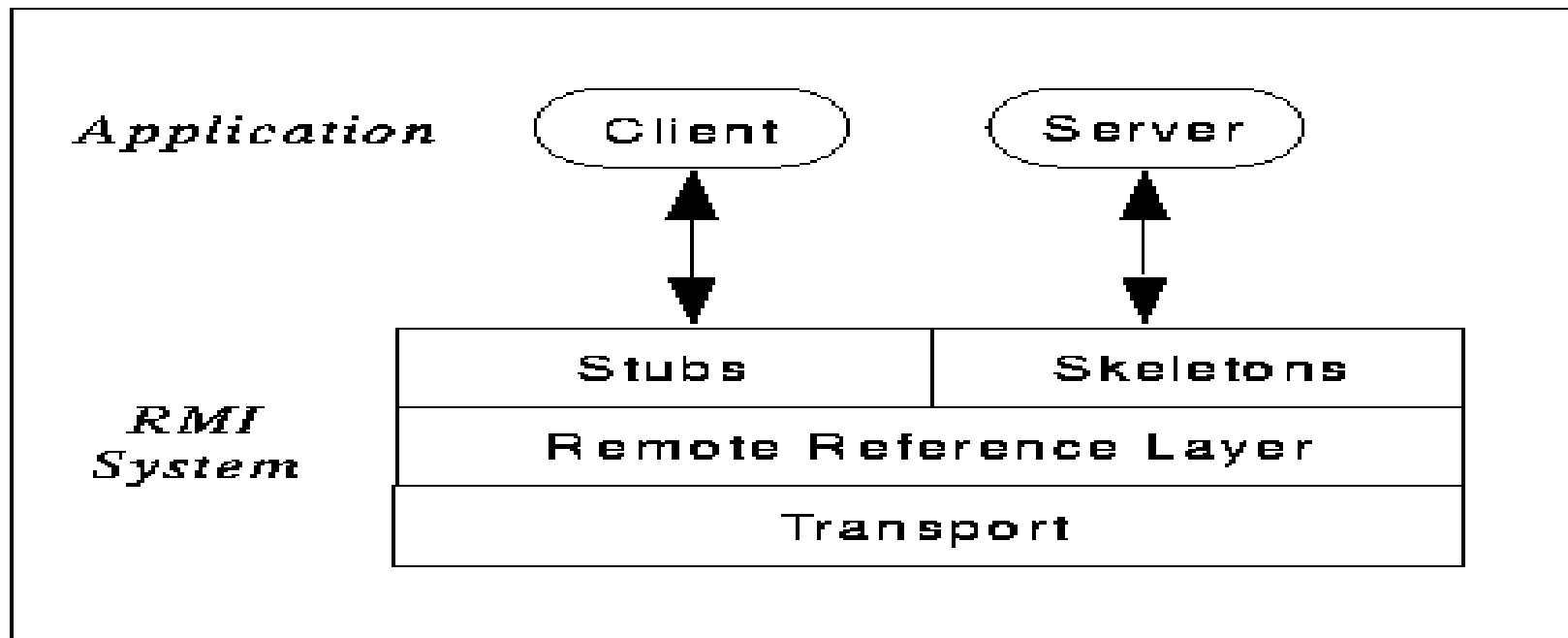
Java RMI

▶ 1st class RPC package

- Fully integrated with Java language
- Remote interfaces are described through Java interfaces
- Stubs (and skeletons) generated according to interfaces

▶ Since Java 1.1

Java RMI Architecture



Stub/Skeleton Layer

▶ Stub

- Has same interface as 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

Remote Reference Layer

- ▶ **Carries out remote reference protocol**
 - Independent of stubs/skeletons
- ▶ **Remote object implementation chooses invocation protocol**
 - Unicast point-to-point
 - Replicated object group
 - Support for specific replication strategy
 - Support for persistent reference to remote object (automatic activation of remote object)
 - Reconnection strategies

Transport Layer

► Responsibilities

- Connection set-up to remote address space
- Managing connections, monitoring connection “liveness”
- Listening for incoming calls
- Maintaining a table of remote objects
- Connection set-up for incoming call
- Locating the dispatcher for the target of the remote call

► Abstractions

- Endpoint: denotes an address space or JVM
- Channel: conduit between two address spaces, manages connections
- Connection: data transfer (input/output)

Designing a Java RMI Application

1. **Write the interfaces of the remote (i.e., remotely accessible) objects: coarse grain**
2. **Write the implementations of the remote objects**
3. **Write other classes involved: fine grain**
4. **Compile the application with `javac`**
5. **Originally: generate stubs and skeletons with `rmic`**

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 Hello World Remote Interface

```
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**

► Concrete class

- `java.rmi.server.UnicastRemoteObject` **extends** `RemoteServer`
 - **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`

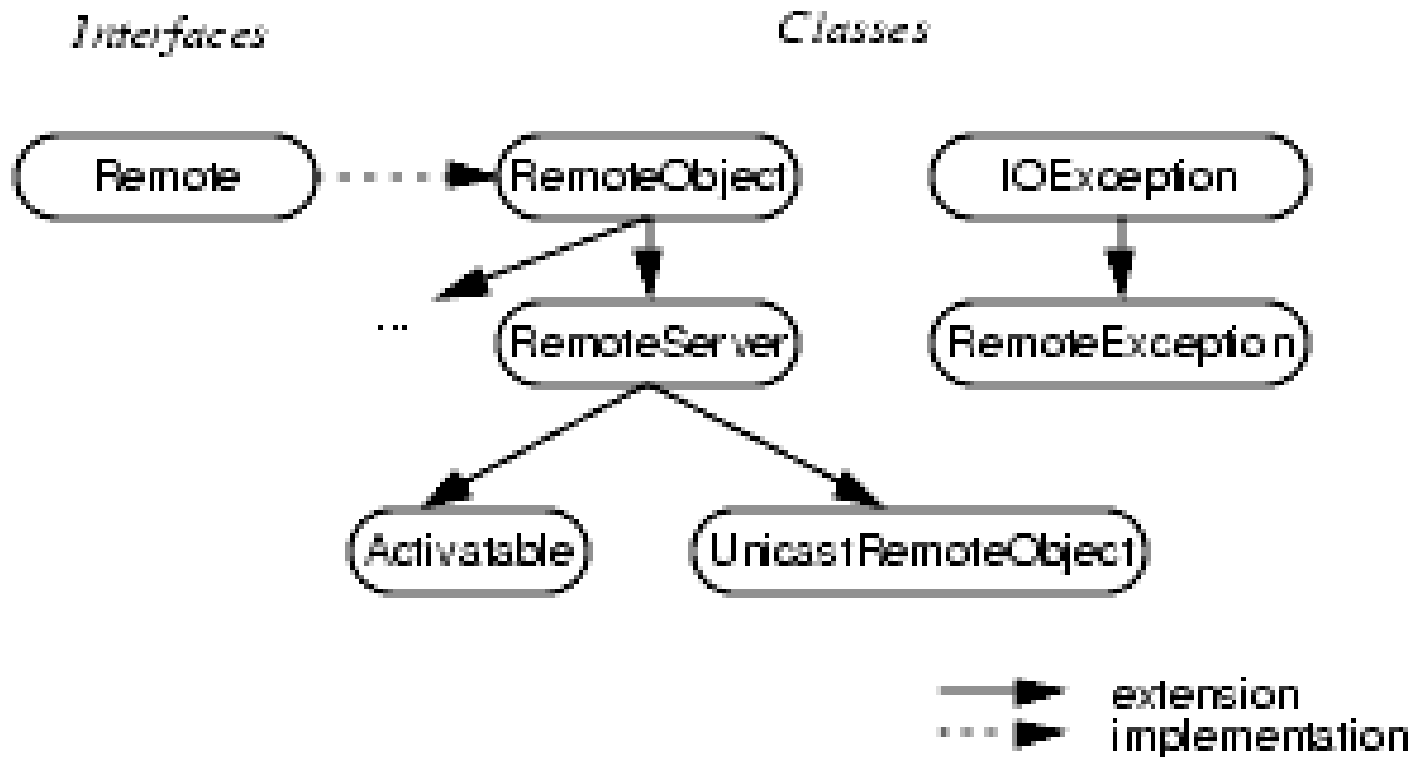
A Hello World Implementation

```
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"); }
}
```

Simple Classes and Interfaces



Constructing a Remote Object

▶ The constructor

- Calls the no-argument constructor of the `UnicastRemoteObject` class (implicitly or explicitly)
- Which *exports* a `UnicastRemoteObject`, meaning that it is available to accept incoming requests by listening to calls from clients on an anonymous port
- Throws `RemoteException`, since the constructor of `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 `UnicastRemoteObject.exportObject()`

Starting a Server

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

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...  
    }  
}
```

Lookup

▶ How does a client object find a server object?

- Registry

▶ Interface

```
interface Registry {  
    void bind(String name, Remote obj);  
        // Binds remote reference to specified name in registry.  
    String[] list();  
        // Returns an array of the names bound in this registry.  
    Remote lookup(String name);  
        // Returns reference bound to specified name in registry  
}
```

▶ Registry “creates” proxies

Parameters and Return Values

▶ Can be

- Remote objects
 - Passed by reference
- Value objects
 - Primitive types (e.g., `int`, `boolean`, ...)
 - Serializable, i.e. implementing `java.io.Serializable`: fields are copied (except `static` or `transient`) and serialized
- Remote *and* serializable objects
 - Former takes precedence

▶ Serialization

- Can be overridden
- A copy of the object is created at the client side

Exceptions

▶ **All RMI exceptions subclass** `RemoteException`

▶ **Examples**

- `StubNotFoundException` **or**
`SkeletonNotFoundException`: **forgot `rmic`?**
- `ServerError`: **error while server executes remote method**
- `ServerException`: **remote exception in server's remote method call, received by 1st invoker**

Distributed Garbage Collection

- ▶ **(Local) object garbage collected if**
 - No (local) reference to it exists
- ▶ **Remote object garbage collected if**
 - No local reference to it exists
 - No remote reference to it exists
- ▶ **JVM keeps track of the # of refs to an object**
 - First reference triggers message send to JVM hosting the object
 - Discarding last reference triggers message send as well
 - When no remote JVM references object anymore, reference becomes “weak”

Improvements

▶ Java 1.2

- No more skeletons needed. `UnicastRemoteObject` deals with everything

▶ Java 1.3

- Introduction of dynamic proxies. Proxies can be generated at runtime as byte-code

```
public class Proxy {  
    public static Object newProxyInstance(  
        ClassLoader cl, Class[] intfs, InvocationHandler ih) ...  
}
```

▶ Java 1.5

- No need to call `rmic` anymore. Typing constraints are verified directly by `javac`

Research

▶ **Asynchrony, e.g.**

- Future method invocations
- Parameter passing (partial transfer, transfer-by-need, lazy transfer)

▶ **Efficiency, e.g.**

- Batching
- One-to-many invocations

▶ **Safety, e.g.**

- Distributed subtyping
- Session types
- Type state

More Information

- ▶ **Java site:** <http://java.sun.com>
- ▶ **Java RMI site:** <http://java.sun.com/javase/technologies/core/basic/rmi/>
- ▶ **Tutorial:** <http://java.sun.com/docs/books/tutorial/rmi/>
- ▶ **Specification:** <http://java.sun.com/j2se/1.5.0/docs/guide/rmi/spec/rmiTOC.html>
- ▶ **White paper:** <http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/>
- ▶ **Dynamic proxies:** <http://java.sun.com/j2se/1.5.0/docs/guide/reflection/proxy.html>