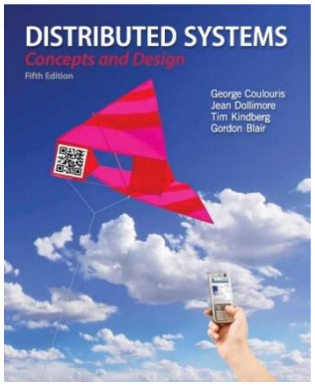
Distributed Objects Programming - Remote Invocation



Some concepts are drawn from Chapter 5

Rajkumar Buyya

Cloud Computing and **D**istributed **S**ystems (CLOUDS) Laboratory
School of Computing and Information Systems

The University of Melbourne, Australia
http://www.cloudbus.org/652

Co-contributors: Xingchen Chu, Rodrigo Calheiros, Maria Sossa, Shashikant Ilager

Sun Java online tutorials:

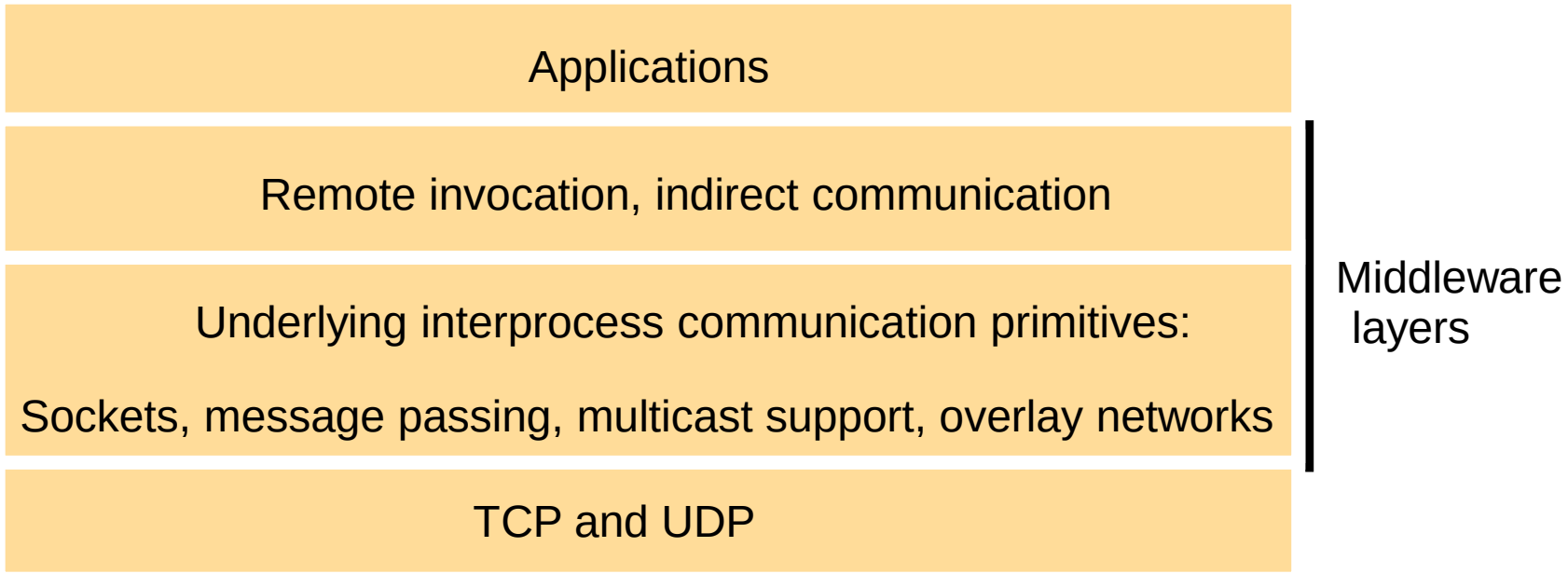
http://java.sun.com/docs/books/tutorial/rmi/

Outline

- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

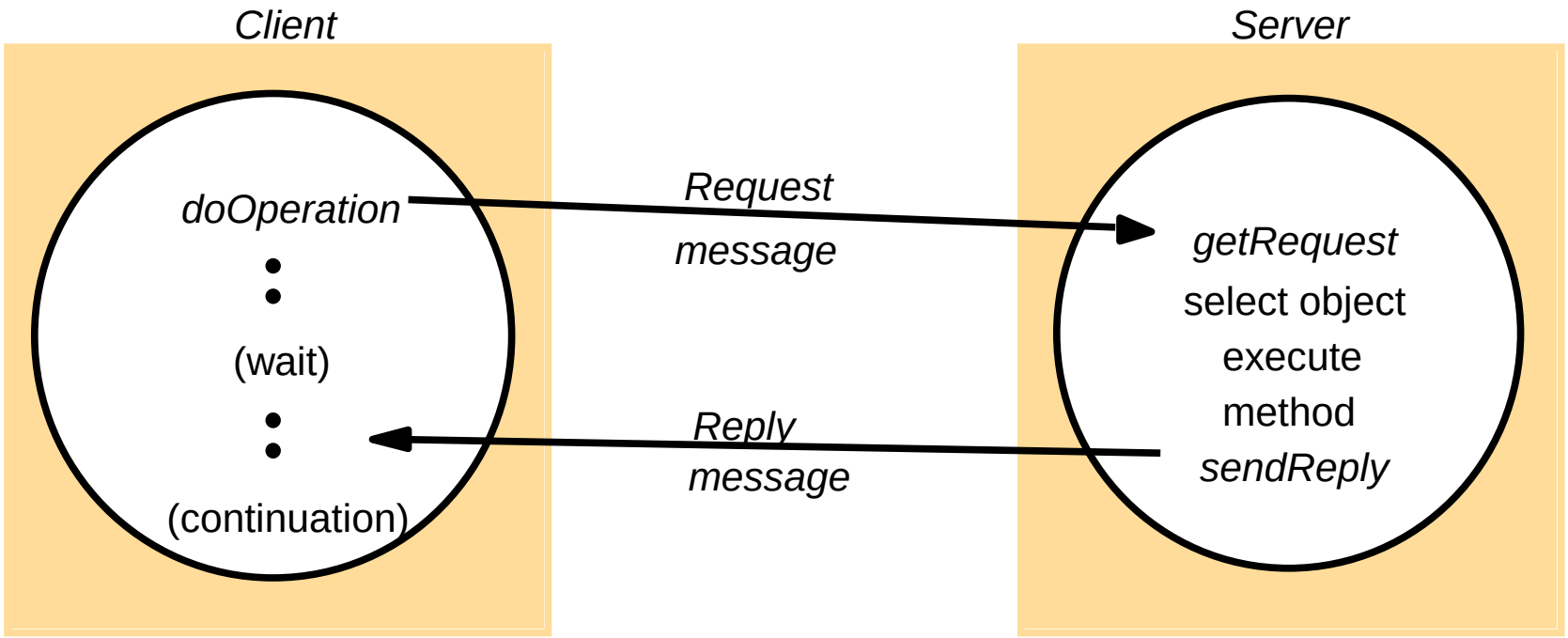
Introduction

- We cover high-level programming models for distributed systems. Two widely used models are:
  - *Remote Procedure Call (RPC)* - an extension of the conventional procedure call model
  - *Remote Method Invocation (RMI)* - an extension of the object-oriented programming model.



Request-Reply Protocol

- Exchange protocol for the implementation of remote invocation in a distributed system.
- We discuss the protocol based on three abstract operations: doOperation, getRequest and sendReply



Request-Reply Operations

- **public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)**
  - Sends a request message to the remote server and returns the reply
  - The arguments specify the remote server, the operation to be invoked and the arguments of that operation
- **public byte[] getRequest ()**
  - Acquires a client request via the server port
- **public void sendReply (byte[] reply, InetAddress clientHost, int clientPort)**
  - Sends the reply message reply to the client at its Internet address and port

Invocation Semantics

- Middleware that implements remote invocation generally provides a certain level of semantics:
  - **Maybe**: The remote procedure call may be executed once or not at all. Unless the caller receives a result, it is unknown as to whether the remote procedure was called.
  - **At-least-once**: Either the remote procedure was executed at least once, and the caller received a response, or the caller received an exception to indicate the remote procedure was not executed at all.
  - **At-most-once**: The remote procedure call was either executed exactly once, in which case the caller received a response, or it was not executed at all and the caller receives an exception.
- Java RMI (Remote Method Invocation) supports at-most-once invocation.
  - It is supported in various editions including J2EE.
- Sun RPC (Remote Procedure Call) supports at-least-once semantics.
  - Popularly used in Unix/C programming environments

Distributed Objects

- A programming model based on Object-Oriented principles for distributed programming.
- Enables reuse of well-known programming abstractions (Objects, Interfaces, methods…), familiar languages (Java, C++, C#...), and design principles and tools (design patterns, UML…)
- Each process contains a collection of objects, some of which can receive both remote and local invocations:
  - Method invocations between objects in *different processes* are known as **remote method invocation,** *regardless the processes run in the same or different machines*.
- Distributed objects may adopt a client-server architecture, but other architectural models can be applied as well.

Outline

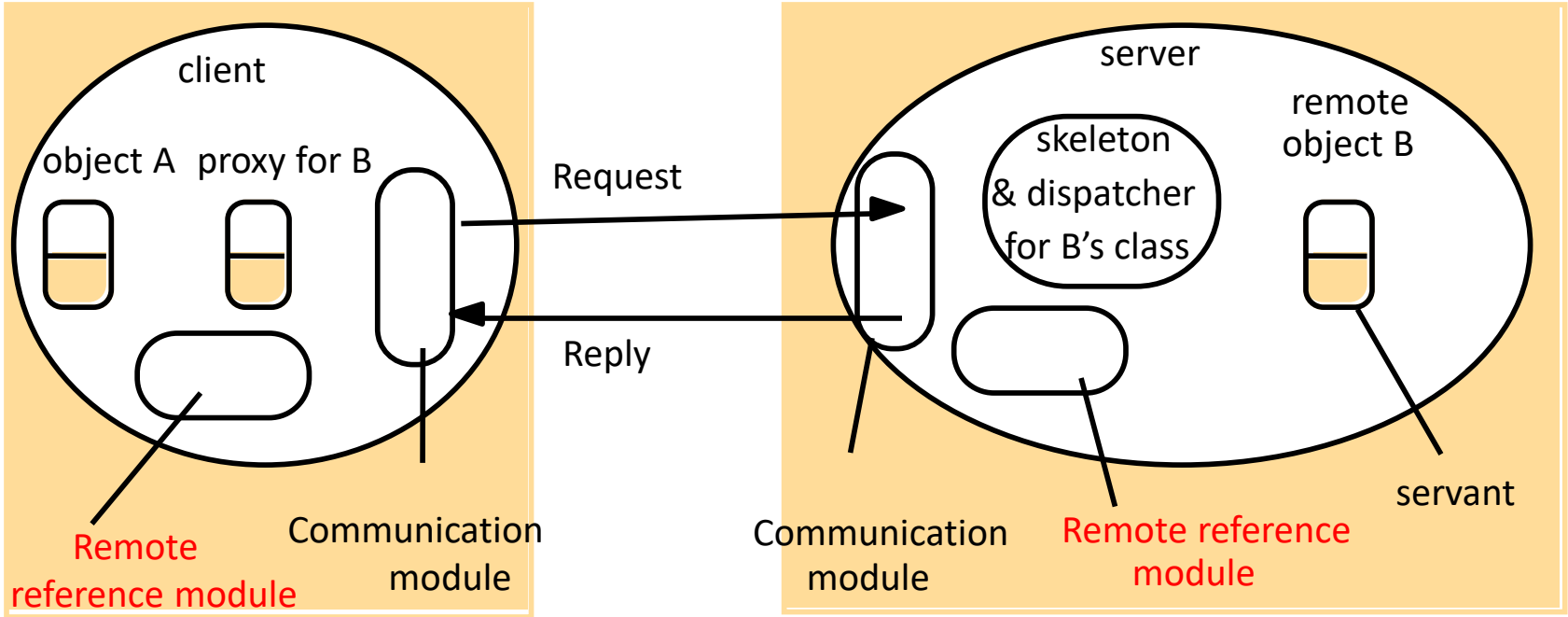
- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

Java RMI

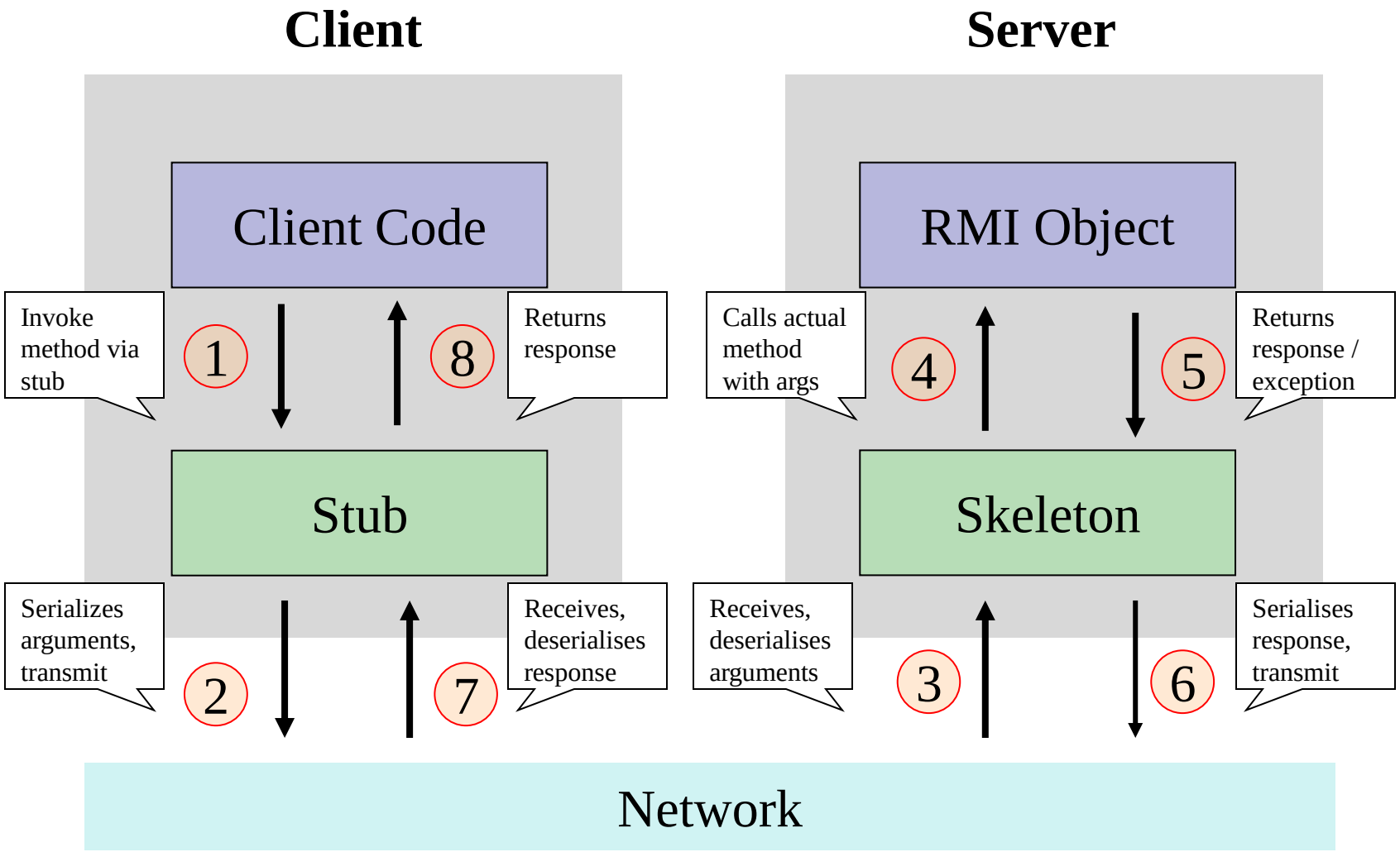
- Java Remote Method Invocation (Java RMI) is an extension of the Java object model to support distributed objects
  - methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts
- Single-language system with a proprietary transport protocol (JRMP, java remote method protocol)
  - Also supports IIOP (Internet Inter-Orb Protocol) from CORBA
- RMI uses object serialization to marshal and unmarshal
  - Any serializable object can be used as parameter or method return
- **Releases of Java RMI**
  - Java RMI is available for Java Standard Edition (JSE), Java Micro Edition (JME), and Java Enterprise Edition (Java EE)

RMI Architecture and Components

- Remote reference module (at client & server) is responsible for providing addressing to the proxy (stub) object
- Proxy is used to implement a stub and provide transparency to the client. It is invoked directly by the client (as if the proxy itself was the remote object), and then marshal the invocation into a request
- Communication module is responsible for networking
- Dispatcher selects the proper skeleton and forward message to it
- Skeleton un-marshals the request and calls the remote object



Invocation Lifecycle



Outline

- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

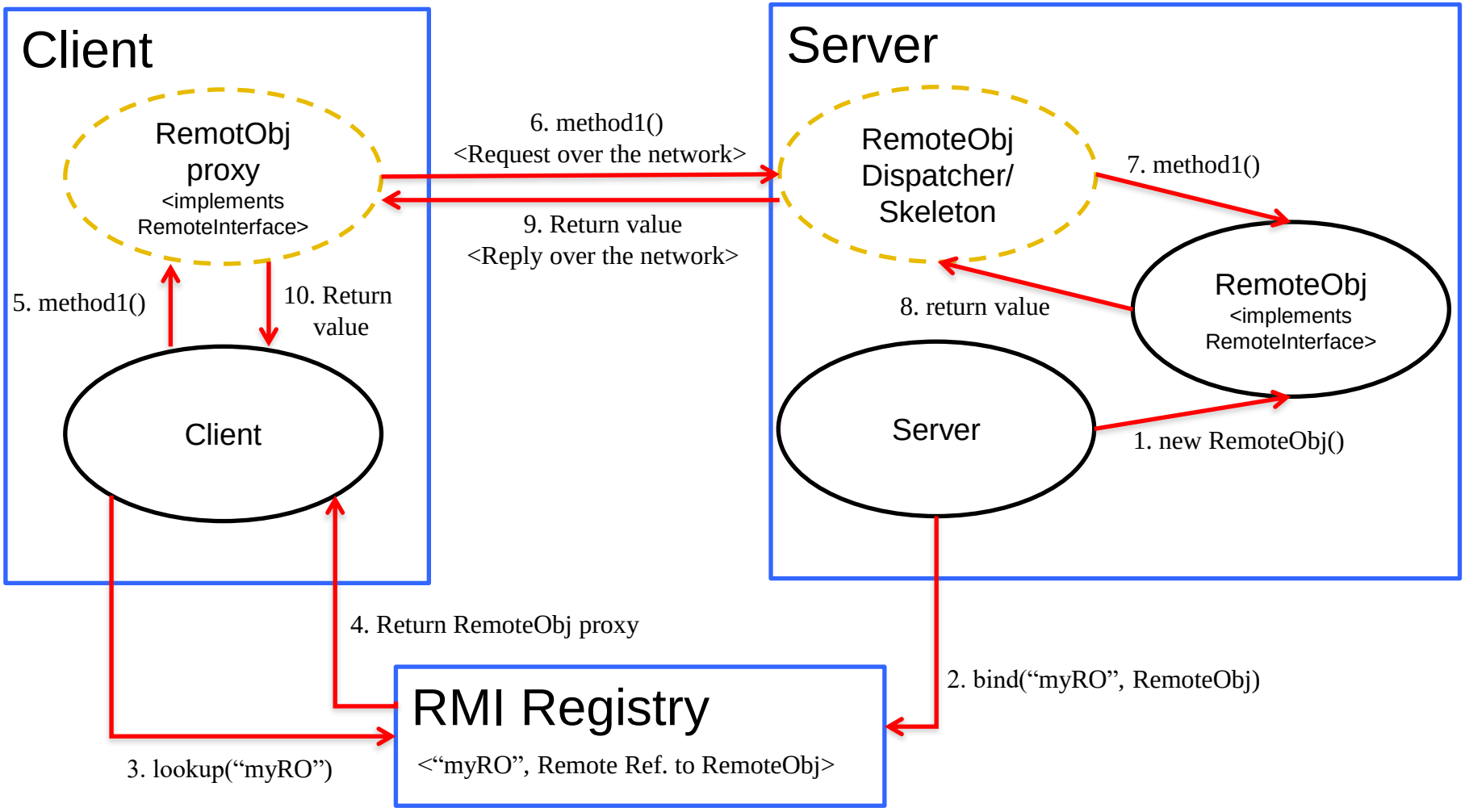
Steps for implementing an RMI application

- Design and implement the components of your distributed application
  - Remote interface
  - Servant program
  - Server program
  - Client program
- Compile source code and generate stubs
  - Client proxy stub
  - Server dispatcher and skeleton
- Make classes network accessible
  - Distribute the application on server side
- Start the application

RMI Programming and Examples

- Application Design
  - Remote Interface
    - Exposes the set of methods and properties available
    - Defines the contract between the client and the server
    - Constitutes the root for both stub and skeleton
  - Servant component
    - Represents the remote object (skeleton)
    - Implements the remote interface
  - Server component
    - Main driver that makes available the servant
    - It usually registers with the naming service
  - Client component

Java RMI


Client
RemotObj
proxy
<implements
RemoteInterface>
5. method1()
10. Return
value
Client
6. method1()
<Request over the network>
9. Return value
<Reply over the network>
Server
RemoteObj
Dispatcher/
Skeleton
7. method1()
8. return value
RemoteObj
<implements
RemoteInterface>
Server
1. new RemoteObj()
4. Return RemoteObj proxy
2. bind("myRO", RemoteObj)
3. lookup("myRO")
RMI Registry
<"myRO", Remote Ref. to RemoteObj>

Example application – Hello World

- Server side
  - Create a HelloWorld interface
  - Implement HelloWorld interface with methods
  - Create a main method to register the HelloWorld service in the RMI Name Registry
  - Generate Stubs and Start RMI registry
  - Start Server
- Client side
  - Write a simple Client with main to lookup HelloWorld Service and invoke the methods

1. Define Interface of remote method

```
//file: HelloWorld.java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloWorld extends Remote {
   public String sayHello(String who) throws RemoteException;
}
```

2. Define RMI Server Program

```
// file: HelloWorldServer.java
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloWorldServer extends UnicastRemoteObject  implements HelloWorld {
        public HelloWorldServer() throws RemoteException {
                super();
        }
        public String sayHello(String who) throws RemoteException {
                return "Hello "+who+" from your friend RMI 433-652 :-)";
        }
        public static void main(String[] args) {
                String hostName = "localhost";
                String serviceName = "HelloWorldService";
                if(args.length == 2){
                        hostName = args[0];
                        serviceName = args[1];
                }
                try{
                        HelloWorld hello = new HelloWorldServer();
                        Naming.rebind("rmi://"+hostName+"/"+serviceName, hello);
                        System.out.println("HelloWorld RMI Server is running...");
                }catch(Exception e){
                        e.printStackTrace();
                }
        }
}
```

3. Define Client Program

```
// file:  RMIClient.java
import java.rmi.Naming;
public class RMIClient {
   public static void main(String[] args) {
      String hostName = "localhost";
      String serviceName = "HelloWorldService";
      String who = "Raj";
      if(args.length == 3){
         hostName = args[0];
         serviceName = args[1];
          who = args[2];
       }
       else if(args.length == 1){
           who = args[0];
       }
       try{
        HelloWorld hello = (HelloWorld)Naming.lookup("rmi://"+hostName+"/"+serviceName);
        System.out.println(hello.sayHello(who));
       }catch(Exception e){
           e.printStackTrace();
       }
   }
}
```

Define Access Policy

- Example: File HelloPolicy to contain

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

Java RMI Example

- Running the Server and Client
  - Compile Client and Server classes
  - Develop a security policy file (e.g., HelloPolicy)
    - grant { permission java.security.AllPermission "", ""; };
  - Start RMI registry
    - rmiregistry &
  - Start server
    - java -Djava.security.policy=HelloPolicy HelloWorldServer
  - Run a client program
    - java -Djava.security.policy=HelloPolicy RMIClient
    - java -Djava.security.policy=HelloPolicy RMIClient Pascal

Outline

- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

Security Manager

- Java's security framework
  - *java.security.-*
  - Permissions, Principle, Domain etc.
  - Security manager, for access control (file, socket, class load, remote code etc)
  - *$JAVA_HOME/jre/lib/security/java.policy*
- Use security manager in RMI
  - RMI recommends to install a security manager, or RMI may not work properly while encountering security constraints.
  - A security manager ensures that the operations performed by downloaded code go through a set of security checks.
    - Eg. Connect and accept ports for RMI socket and allowing code downloading

Security Manager (cont.)

- Two ways to declare security manager
  - Use System property java.security.manager

    java –Djava.security.manager HelloWorldImpl
  - Explicit declare in the source code

```
public static void main(String[]args){
    //check current security manager
    if(System.getSecurityManager()==null){
            System.setSecurityManager(new SecurityManager ());
    }
    …
    //lookup remote object and invoke methods.
}
```

- Use customized policy file instead of java.policy
  - Usage

    java -Djava.security.manager -Djava.security.policy=local.policy HelloWorldImpl

File: “local.policy” contents

Specific permissions:

```
grant {
    permission java.net.SocketPermission   "*:1024-65535","connect,accept";
    permission java.io.FilePermission       "/home/globus/RMITutorial/-", "read";
};
```

Grant all permissions:

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

Exceptions

- The only exception that could be thrown out is *RemoteException*
- All RMI remote methods **have** to throw this exception
- The embedded exceptions could be:
  - java.net.UnknownHostException or java.net.ConnectException: if the client can't connect to the server using the given hostname. Server may not be running at the moment
  - java.rmi.UnmarshalException: if some classes not found. This may because the codebase has not been properly set
  - Java.security.AccessControlException: if the security policy file java.policy has not been properly configured

Passing objects

- Restrictions on exchanging objects
  - Implementing *java.io.serializable*
  - All the fields in a serializable object must be also serializable
  - Primitives are serializable
  - System related features (e.g. *Thread*, *File*) are non-serializable
- How about the socket programming issues?
  - Where are sockets and corresponding input, output streams?
  - How to handle object passing?
  - Who does all the magic?

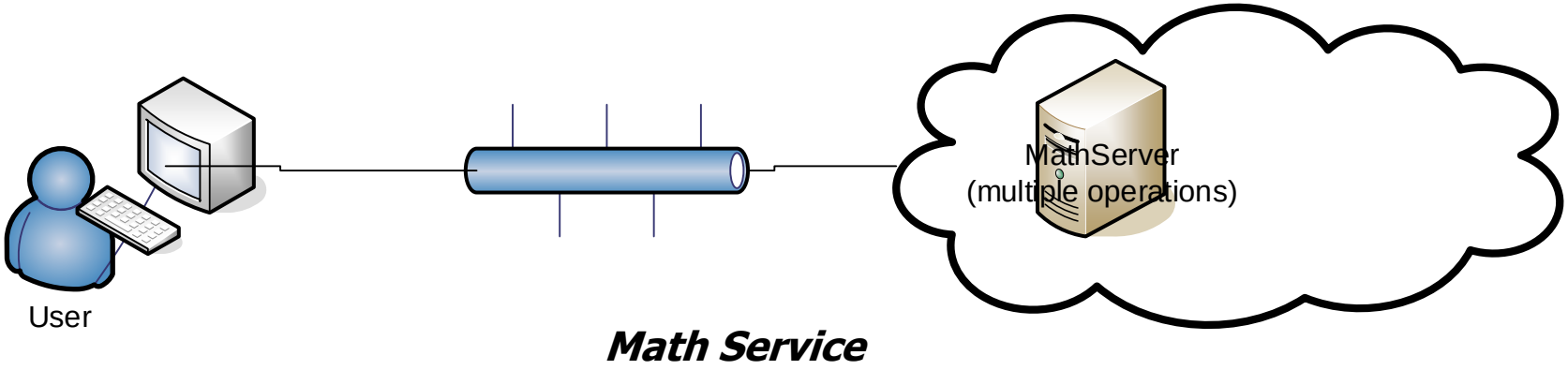
RMI Dynamic Class Loading

- Ability to download bytecode (classes) from Remote JVM
- New types can be introduced into a remote virtual machine without informing the client
  - Extend the behavior of an application dynamically
  - Removes the need to deploy stubs manually
- Explicit set property to support dynamic class load
  - Specify system property *java.rmi.server.codebase* to tell the program where to download classes

Outline

- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

A Simple Math Server in RMI



Java RMI Example

- Specify the Remote Interface

```
public interface IRemoteMath extends Remote {
        double add(double i, double j) throws RemoteException;
        double subtract(double i, double j) throws
RemoteException;
}
```

Java RMI Example

- Implement the Servant Class

```
public class RemoteMathServant extends UnicastRemoteObject implements IRemoteMath {
    public double add ( double i, double j ) throws RemoteException {
        return (i+j);
    }

    public double subtract ( double i, double j ) throws RemoteException {
        return (i-j);
    }
}
```

Java RMI Example

- Implement the server

```
public class MathServer {
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
        try{
            IRemoteMath remoteMath = new RemoteMathServant();
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Compute", remoteMath );
            System.out.println("Math server ready");
        }catch(Exception e) {
            e.printStackTrace();
        }
    }
}
```

Java RMI Example

- Implement the client program

```
public class MathClient {
    public static void main(String[] args) {
        try {
            if(System.getSecurityManager() == null)
                System.setSecurityManager( new RMISecurityManager() );

            LocateRegistry.getRegistry("localhost");
            IRemoteMath remoteMath = (IRemoteMath) registry.lookup("Compute");

            System.out.println( "1.7 + 2.8 = ” + math.add(1.7, 2.8) );
            System.out.println( "6.7 - 2.3 = ” + math.subtract(6.7, 2.3) );
        }
        catch( Exception e ) {
            System.out.println( e );
        }
    }
}
```

Java RMI Example

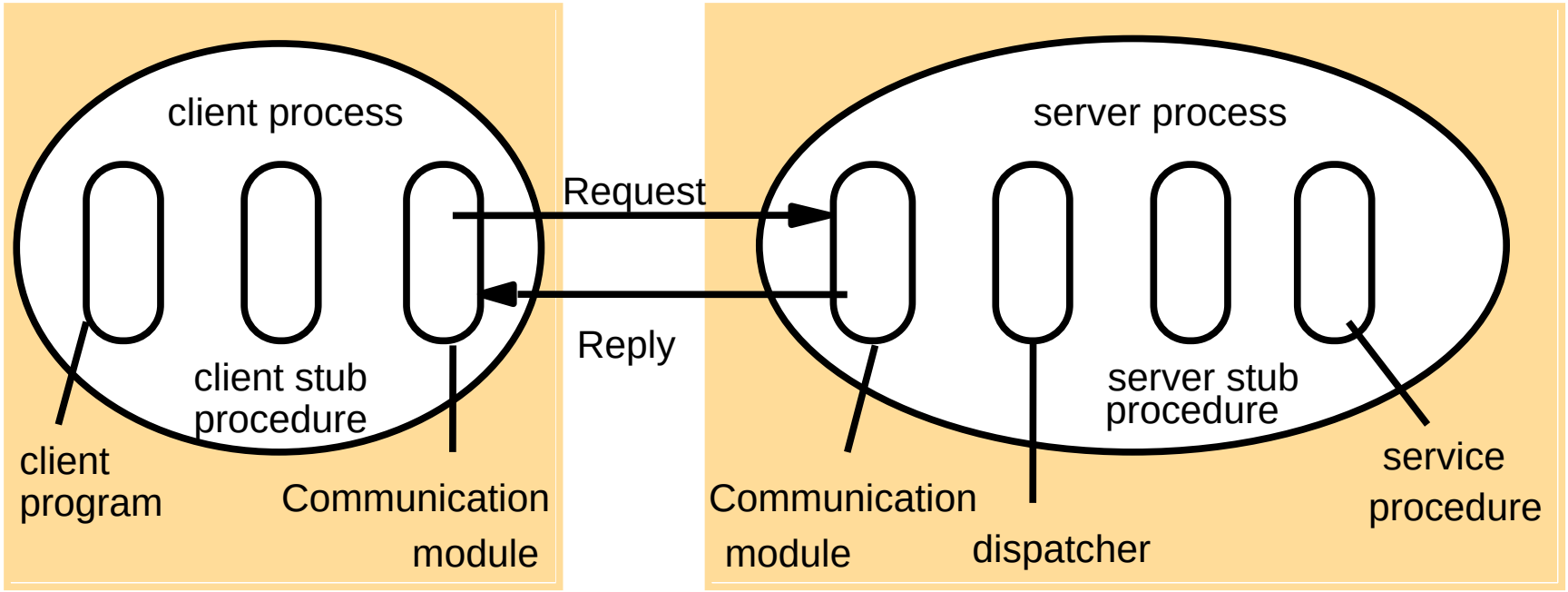
- Running the Server and Client
  - Compile Client and Server classes
  - Develop a security policy file
    - grant { permission java.security.AllPermission "", ""; };
  - Start RMI registry
    - rmiregistry &
  - Start server
    - java -Djava.security.policy=policyfile MathServer
  - Start client
    - java -Djava.security.policy=policyfile MathClient

Outline

- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

Remote Procedure Call (RPC) – used in C

- RPCs enable clients to execute procedures in server processes based on a defined service interface.



Remote Procedure Call (RPC)

- Communication Module
  - Implements the desired design choices in terms of retransmission of requests, dealing with duplicates and retransmission of results
- Client Stub Procedure
  - Behaves like a local procedure to the client. Marshals the procedure identifiers and arguments which is handed to the communication module
  - Unmarshalls the results in the reply
- Dispatcher
  - Selects the server stub based on the procedure identifier and forwards the request to the server stub
- Server stub procedure
  - Unmarshalls the arguments in the request message and forwards it to the service procedure
  - Marshalls the arguments in the result message and returns it to the client

Summary: RMI Programming

- RMI greatly simplifies creation of distributed applications (e.g., compare RMI code with socket-based apps)
- Server Side
  - Define interface that extend java.rmi.Remote
  - Servant class both implements the interface and extends java.rmi.server.UnicastRemoteObject
  - Register the remote object into RMI registry
  - Ensure both rmiregistry and the server is running
- Client Side
  - No restriction on client implementation, both thin and rich client can be used. (Console, Swing, or Web client such as servlet and JSP)