3. Network Programming and RMI in Java

3.1. Network Programming

- Based on the \textit{java.net} package

- Classes for \textit{socket} and for \textit{URL} (Uniform Resource Locator) management

- With the help of URLs applications can fetch e.g.
  - An entire file
  - A record from a database

- Sockets are the basic access mechanism to the network
Sockets

- Sockets (originated in *BSD Unix*) implement *Transport Service Access Points* (TSAPs)

- Java supports a simplified socket interface
• Connection-oriented protocols (based on class Socket)
  ➢ First, a connection is established
  ➢ Then good quality communication
  ➢ At the end, the connection is closed
  ➢ Similar to a telephone call
  ➢ Implementation could support theoretically any such protocol
  ➢ Actual implementations support TCP

• Connection-less protocols (based on class DatagramSocket)
  ➢ All data items are addressed and routed individually
  ➢ There is no guarantee for quality (especially, no guarantee for ordering)
  ➢ Cheap to implement, the missing features can be implemented by higher level protocols
  ➢ Similar to a telegram or surface mail
  ➢ Implementation could support theoretically any such protocol
  ➢ Actual implementations support UDP
• Example: Establishing Connection

```java
try {
    Socket sock = new Socket("lpc1.itec.uni-klu.ac.at", 1234)
    //port 1234
}
catch (UnknownHostException e) {
    System.out.println("Can't find host");
}
catch (IOException e) {
    System.out.println("Error connecting host");
}
```

• Example: Sending and receiving data over a connection

- Data bytes can be sent and received individually over sockets
- Sockets can also provide streams (java.io package)

```java
try {
    Socket server = new Socket("server...",1234)
    InputStream in = server.getInputStream();
    OutputStream out = server.getOutputStream();

    out.write(41);   //Send a byte to the server

    PrintStream pout = new PrintStream (out);
    pout.println("Hello!"); //String to server

    Byte back = in.read(); //Read from server

    DataInputStream din =
        new DataInputStream (in);
    String response = din.readLine();
    //Read the response (a line) from server

    server.close();   //Close connection
}
```
• Example: Server
  ➢ The server waits in a cycle for clients
  ➢ It is single-threaded → A realistic server forks a thread for every client-connection
  ➢ A realistic server must handle IOException properly → this is not easy – an “orphan” problem

```java
// Server “server...” on port 1234
try {
    ServerSocket listener = new ServerSocket(1234);

    while ( !finished ) {
        Socket aClient = listener.accept();
        // waits for a connection to any client
        InputStream in = aClient.getInputStream();
        OutputStream out = aClient.getOutputStream();

        Byte importantByte = in.read();
        // read a byte from client

        DataInputStream din = new DataInputStream(in);
        String request = din.readLine();
        //Read the request (a line) from client

        out.write(41); //Send a byte to the client
        PrintStream pout = new PrintStream (out);
        pout.println("Goodbye!");
        //String to the client

        aClient.Close(); // Close connection to client
    } // WHILE waits for a next client

    listener.close(); // Close the server socket
}```
catch (IOException e) {}
3.2. RMI  
(Remote Method Invocation)

- RMI is the state-of-the art Java technology to write distributed client/server systems
- RMI is more dynamic than CORBA in supporting dynamic loading of code between clients and servers
- Code and design pattern reuse is supported better due to this feature
- CORBA is more dynamic than RMI in supporting dynamic interfaces
- RMI architecture:
3.2.1. Remote vs. Non-remote objects

• Similarities

- A reference to a remote object can be passed as an argument or as a result in any method call (local or remote)
- A remote object can be cast to any of the set of remote interfaces actually supported with the usual syntax
- The built-in `instanceof` operator can be used as usual

• Differences

- Clients of remote objects interact with remote interfaces, never with implementation classes (would be quite desirable for all Java classes)
- Non-remote arguments and results are passed by value, i.e. in the case of an object, a deep copy is sent. The object must be `serializable`
- Remote object arguments and results are passed by reference
- The semantics of some methods of `java.lang.Object` must be specialized:
  - `equals`
  - `hashCode`
  - `toString`
- Cloning of remote objects is not possible
- Exception handling is more complex
3.2.2. RMI Interfaces and Classes

- A remote interface must at least (directly or indirectly) extend `java.rmi.Remote`
- A remote method declaration must satisfy:
  - It must throw `java.rmi.RemoteException` or a superclass of it (such as `io.IOException` or `lang.Exception`)
  - A remote object declared as parameter or return value must be declared as a remote interface
  - A remote interface may extend another, non-remote interface, if all of its methods satisfy the remote method declaration requirements
  - `java.rmi.Remote` is an empty interface:
    ```java
    public interface Remote {}
    ```
• Use of RMI
  ➢ Create a remote interface
  ➢ Create implementations of the remote interface
  ➢ Create clients of the remote interface
• Example for remote interfaces

```java
public interface BankAccount extends java.rmi.Remote {
    public void deposit (float amount)
    throws java.rmi.RemoteException;
    public void withdraw (float amount)
    throws Overdrawn, java.rmi.RemoteException;
    public float getBalance ()
    throws java.rmi.RemoteException;
} // BankAccount
```

```java
public interface Alpha {
    public final String const = "constants are o.k. as well"
    public Object x (Object obj)
    throws java.rmi.RemoteException;
    public void y ()
    throws java.rmi.RemoteException;
} // Alpha
```

// Beta extends Alpha, which is non-remote, but conform
```java
public interface Beta extends Alpha, java.rmi.Remote {
    public void ping ()
    throws java.rmi.RemoteException;
} // Beta
```
• Example for the implementation of a remote interface

```java
public class BankAccountImpl extends UnicastRemoteObject implements BankAccount {

    private float balance = 0.0;

    public BankAccountImpl(float initialBalance) throws java.rmi.RemoteException {
        balance = initialBalance;
    }

    public void deposit(float amount) throws java.rmi.RemoteException {
        // Implementation
    }

    public void withdraw(float amount) throws Overdrawn, java.rmi.RemoteException {
        // Implementation
    }

    public float getBalance() throws java.rmi.RemoteException {
        // Implementation
    }

} // BankAccountImpl
```
3.2.3. RMI Registration

- Servers must register their services by name
  - The registration must be done on the same machine where the service is running

- Clients may lookup for services by name on any computer they can reach
  - The client must know where the service is located
  - To find a service on the network needs an additional directory service

- Basic interface: Registry in the java.rmi.registry package

- The Naming class provides a comfortable set of the most important methods for registration

  - Naming.bind, Naming.rebind
    Binds a service to an URL (if not yet exists resp. anyway)
    TimeServerImpl TSI = new TimeServerImpl();
    Naming.rebind("TimeServer", TSI);

  - Naming.lookup
    Looks for the required service
    TimeServer TS = null;
    try { TS = (TimeServer)
    Naming.lookup("rmi://lpc1/TimeServer"); . . . }

  - Naming.list
    Returns an array of the names of the registered services
    String [] remObjs = Naming.list("rmi://lpc1/");
**Full Example: Time Server**

1. Remote Interface *TimeServer*

   ```java
   import java.util.*;

   public interface TimeServer extends java.rmi.Remote {
       public Date getTime ()
           throws java.rmi.RemoteException;
   } // TimeServer
   ```

2. Policy file (preferably .java.policy in the home directory)

   ```java
   grant {
       // Allow everything for now
       permission java.security.AllPermission;
   };
   ```

3. Compilation + stub/skeleton generation

   ```bash
   javac *.java
   rmic TimeServerImpl
   ```

4. Starting on the server site

   ```bash
   rmiregistry &    (except the registry is started by the program)
   java TimeServerImpl &

   (java -Djava.security.policy=java.policy TimeServerImpl &
    if policy file is not stored as .java.policy in the home directory)
   ```

5. Starting on the client sites

   ```bash
   java Time lpc5
   ```
6. Implementation of the *TimeServer*

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

public class TimeServerImpl extends UnicastRemoteObject
    implements TimeServer {

    public Date getTime() throws RemoteException {
        return new java.util.Date();
    } // Date

    public TimeServerImpl() throws RemoteException {
        System.out.println("Initializing Timeserver");
    } // TimeServerImpl

    public static void main(String arg[]) {
        System.setSecurityManager(
            new RMISecurityManager());

        try {
            TimeServerImpl TSI = new TimeServerImpl();
            Naming.rebind("TimeServer", TSI);
            System.out.println("Registered with registry");
        } catch (RemoteException e) {
            System.out.println("Error: " + e);
        } catch (java.net.MalformedURLException e) {
            System.out.println("URL Error: " + e);
        } // catch
        } // main
    } // TimeServerImpl
```
7. The Client

```java
import java.rmi.*;
import java.util.*;
static final String SERVER = "lpc4", ServiceName = "TimeServer";
public class Time {

public static void main(String arg[]) {
    String ServerName = (argv.length == 1 ? argv[0] : SERVER);
    TimeServer TS = null;
    try { // The client knows the server site
        TS = (TimeServer) Naming.lookup("rmi://" + ServerName + "/" + ServiceName);
    } catch (NotBoundException e) {
        System.out.println("TimeServer was not found in registry");
        System.exit(0);
    } catch (java.net.MalformedURLException e) {
        System.out.println("URL error: "+ e);
        System.exit(0);
    } catch (RemoteException e) {
        System.out.println("Time error: "+ e);
        System.exit(0);
    } // try
    Date remoteTime = null, localTime = null;
    while (true) {
        try {
            remoteTime = TS.getTime();
            localTime = new java.util.Date();
        } catch (RemoteException e) {
            System.out.println("Time error: "+ e);
            System.exit(0);
        } // try
        if(remoteTime != null) {
            System.out.println("remote time: "+ remoteTime);
            System.out.println("local time: "+ localTime);
            try {
                Thread.sleep(1000);
            } catch (java.lang.InterruptedException e) {
                /* do nothing */
            } // try
        } // if
    } // while
} // main
} // Time
```
8. Starting the registry from the server

```java
public class TimeServerImpl extends UnicastRemoteObject
    implements TimeServer {
    static final int DefaultPort = 1099;

    public Calendar getTime() throws RemoteException {
        return Calendar.getInstance();
    } // Calendar

    public static void main(String arg[]) {
        System.setSecurityManager(new RMISecurityManager());
        try {
            Registry reg =
                LocateRegistry.createRegistry(DefaultPort);
            catch (RemoteException e) {
                try {
                    reg = LocateRegistry.getRegistry();
                } catch (RemoteException e2) {
                    System.out.println("Registry not established" + e);
                    System.exit(0);
                } // try-catch-e2
            } // try-catch-e
            System.out.println("Registry established");
            String [] list = null;
            try {
                TimeServerImpl TSI = new TimeServerImpl();
                reg.rebind("TimeServer", TSI);
                list = reg.list();
            } catch (RemoteException e) {
                System.out.println("Error: " + e);
            }
            if (reg != null) { // Lists all registered services
                System.out.println("List of registered services:");
                for (int i = 0; i < list.length; i++)
                    System.out.println(list[i]);
            } // if
        } // main
    } // TimeServerImpl
```
3.2.3. Passing Behavior

Example, Server-Defined Policy

- Server communicates with the clients via RMI and with the relational database via JDBC
- Clients create expense records
- Clients check the validity of the expense records
- Clients send the proper reports to the server
- Server stores them in the database

- The validation policy may change
- The computation for validation the expense record can be offloaded from the server to the client
- The policy for this check can be changed without any change in the client code
• The client may use the following interface remotely:

```java
public interface ExpenseServer extends Remote {
    Policy getPolicy() throws RemoteException;
    void submitReport(ExpenseReport report)
        throws RemoteException, InvalidReportException;
} // ExpenseServer
```

• The policy interface itself is non-remote, a policy object will be copied to the client, who may use the `checkValid` method locally

```java
public interface Policy {
    void checkValid(ExpenseEntry entry)
        throws PolicyViolationException;
} // Policy
```

• A typical client looks like

```java
Policy curPolicy = server.getPolicy();
// start a new expense report
// show the GUI to the user
while (user keeps adding entries) {
    try {
        curPolicy.checkValid(entry);
        // add the entry to the expense report
    } catch (PolicyViolationException e) {
        // show the error to the user
    } // try
} // while
server.submitReport(report);
...```
• Implementation of the server:

class ExpenseServerImpl
    extends UnicastRemoteObject
    implements ExpenseServer
{
    ExpenseServerImpl() throws RemoteException {
        // ...set up server state...
    }
    public Policy getPolicy() {
        return new TodaysPolicy();
    }
    public void submitReport(ExpenseReport report) {
        // ...write the report into the db...
    }
} // ExpenseServerImpl

• The actual policy is defined by the following class

public class TodaysPolicy implements Policy {
    public void checkValid(ExpenseEntry entry)
        throws PolicyViolationException
    {
        if (entry.dollars() < 20) {
            return; // no receipt required
        } else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException;
        } // if
    } // checkValid
} // TodaysPolicy
• To change the policy
  
  ➢ Provide a new implementation of the interface `Policy`
  ➢ Install it on the server
  ➢ Tell the server to start handing out `TomorrowsPolicy` instead of `TodaysPolicy` objects.

```java
public class TomorrowsPolicy implements Policy {
    public void checkValid(ExpenseEntry entry) throws PolicyViolationException {
        if (entry.isMeal() && entry.dollars() < 20) {
            return;  // no receipt required
        } else if (entry.haveReceipt() == false) {
            throw new PolicyViolationException;
        } // if
    } // checkValid
} // TomorrowsPolicy
```
Example, Compute Server

- We want to delegate some computations to a special compute server (e.g. a very fast processor)

```
public interface Task {
    Object run();
}
```

- Task is a non-remote interface, with a run method that can be overridden by any computation. The signature of run is as “generic” as possible.

- The remote server takes a task from a client, executes the task and returns the result.
• Interface

public interface ComputeServer extends Remote {
    Object compute(Task task)
    throws RemoteException;
} // ComputeServer

• Implementation

public class ComputeServerImpl
    extends UnicastRemoteObject
    implements ComputeServer
{
    public ComputeServerImpl()
        throws RemoteException { . . . }

    public Object compute(Task task) {
        return task.run(); // executes task, returns result
    } // compute

    public static void main(String[] args)
        throws Exception {
        // use the default, restrictive security manager
        System.setSecurityManager(new
            RMISecurityManager());
        ComputeServerImpl server =
            new ComputeServerImpl();
        Naming.rebind("ComputeServer", server);
        System.out.println("Ready to receive tasks");
        return;
    } // main
} // ComputeServerImpl
Agents

- Java provides good tools for “traveling agents”

```java
public interface AgentServer extends Remote {
    void accept(Agent agent)
        throws RemoteException, InvalidAgentException;
}
```

```java
public interface Agent extends java.io.Serializable {
    void run();
}
```

- To start an agent, create a class that implements the `Agent` interface, finding a server, and invoking `accept` with that agent object

- The implementation for the agent would be downloaded to the server and run there

- The accept method would start up a new thread for the agent, invoke its `run` method, and then return, letting the agent continue execution after the method returned

- The agent could migrate to another host by invoking accept on a server running on that host, passing itself as the agent to be accepted, and then killing its thread on the original host
3.2.4. Connecting to an Existing Server

- Java provides access to existing databases by JDBC
- RMI can be used to connect two- and three-tier systems even if both sides are not written in Java.

Connection via JDBC

- The package `java.sql` contains the JDBC package
- Any remote method can be implemented by the server using JDBC calls onto the actual database or by native methods that use other database access mechanisms
- Let us suppose that we had an existing server that stored information on customer orders in a relational database
- The methods return a vector (list) of `Order` objects
- A corresponding remote interface:

```java
public interface OrderServer extends Remote {
    Vector getUnpaid() throws RemoteException, SQLException;
    void shutDown() throws RemoteException;
    // ... other methods (getOrderNumber, getShipped, ...
}
```
• A corresponding implementation

public class OrderServerImpl  
   extends UnicastRemoteObject  
   implements OrderServer
   {
      
      Connection db;          // connection to the db 
      PreparedStatement unpaidQuery;  
                      // unpaid order query 
      OrderServerImpl()  
         throws RemoteException, SQLException {
         db = DriverManager.getConnection  
                      ("jdbc:odbc:orders");  
         unpaidQuery = db.prepareStatement("...");
      } // OrderServerImpl

      public Vector getUnpaid() throws SQLException {
      ResultSet results =  
                     unpaidQuery.executeQuery();  
                      // executes the pre-compiled query 
      Vector list = new Vector();  
      while (results.next())  
         list.addElement(new Order(results));  
      return list;
      } // getUnpaid

      public native void shutDown();
   }

• OrderServerImpl runs on the same system as the database
• Order is a class to hold a customer’s order
3.2.5 Security

- Security Needs
  - E.g. for a compute server inside a secure corporate network, a user identification may be enough
  - For a commercial compute server more security is needed

- Secure channels between client and server and the
  - socket factory that can create sockets of any type
  - Starting with JDK 1.2, the requirements on the services provided for a server's sockets can be specified
  - encrypted sockets
  - compressed sockets
  - ...

- Isolation of downloaded implementations inside a "sandbox"
  - SecurityManager object passes judgement on all security-sensitive actions, such as opening files and network connections
  - RMI provides an \textit{RMISecurityManager} type that is as restrictive as those used for applets (no file access, only connections to the originating host etc.)
  - This prevents downloaded implementations from reading or writing data from the computer, or connecting to other systems behind your firewall
  - Own security manager object may enforce different constraints
3.2.6. Firewalls

- RMI provides a means for clients behind firewalls to communicate with remote servers
- This allows to use RMI to deploy clients on the Internet, such as in applets available on the World Wide Web
- Traversing the client's firewall can slow down communication, so RMI uses the fastest successful technique it can detect
- The technique is discovered by the reference for `UnicastRemoteObject` on the first attempt of the client
- Each of three possibilities are tried in turn
  - Communicate directly to the server's port using sockets
  - If this fails, build a URL to the server's host and port and use an HTTP POST request on that URL, sending the information to the skeleton as the body of the POST. If successful, the results of the post are the skeleton's response to the stub
  - If this also fails, build a URL to the server's host using the standard HTTP port (80), using a CGI script that will forward the posted RMI request to the server
  - Use WebServices 😊
- Whichever of these three techniques succeeds first is used for all future communication with the server
- If none of these succeeds, the remote method invocation fails
- On systems with no firewall, or with communication inside an enterprise behind a firewall, the client will directly connect to the server using sockets
- The secondary communication techniques are significantly slower than direct communication, but their use enables us to write clients that can be used broadly across the Internet and Web.