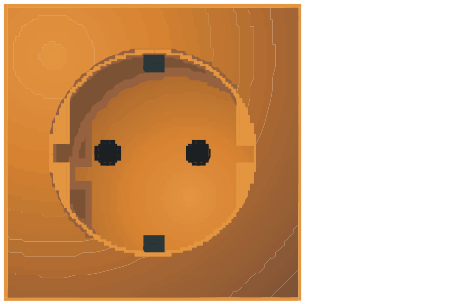
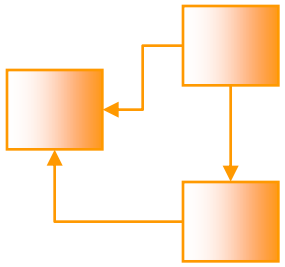
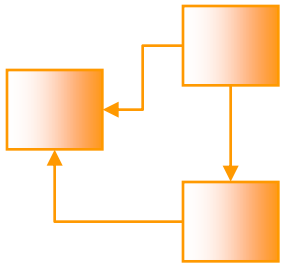
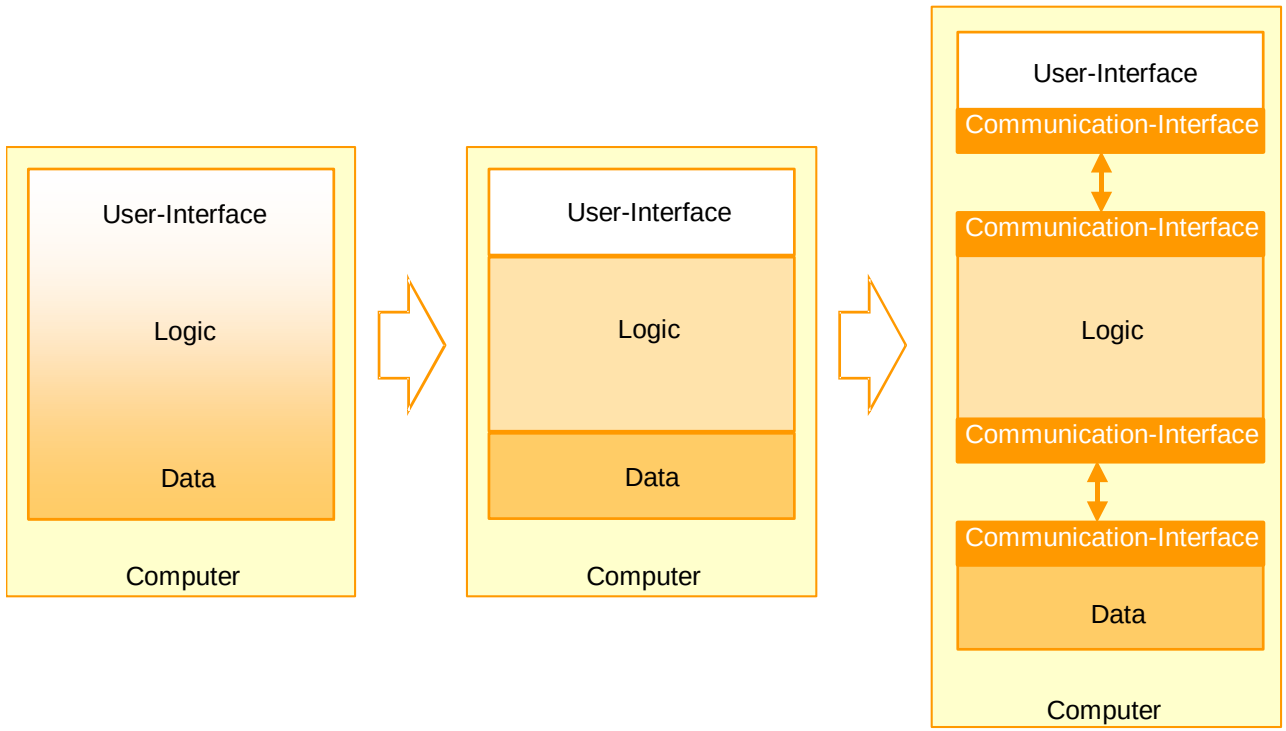
SOCKETS



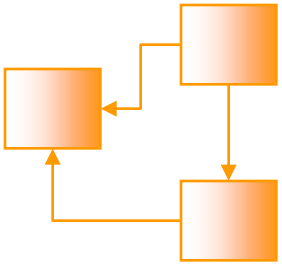


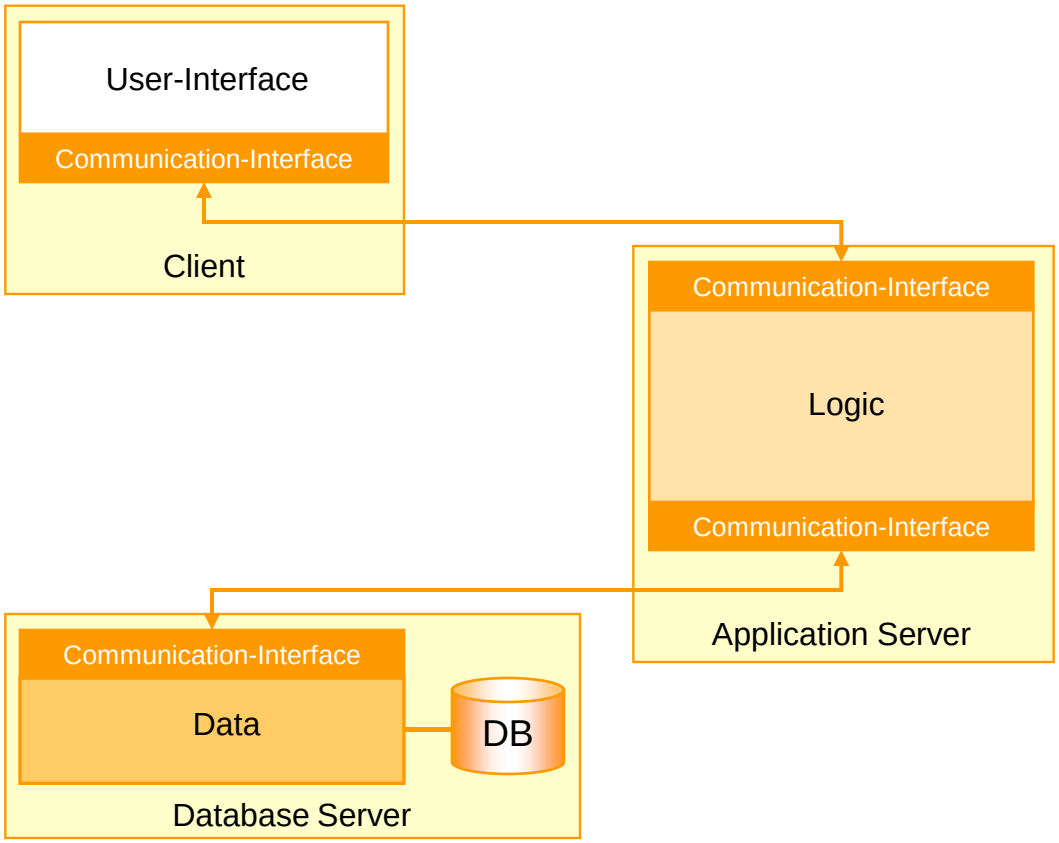
Evolution




User-Interface
Logic
Data
Computer
User-Interface
Logic
Data
Computer
User-Interface
Communication-Interface
Communication-Interface
Logic
Communication-Interface
Communication-Interface
Data
Computer

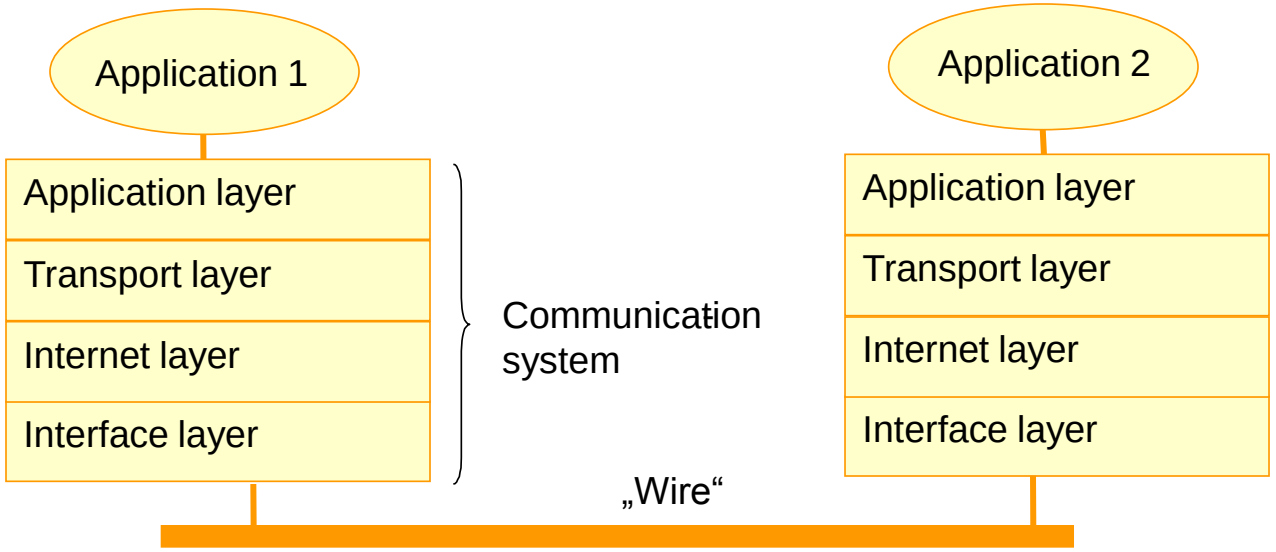
Distributed Systems

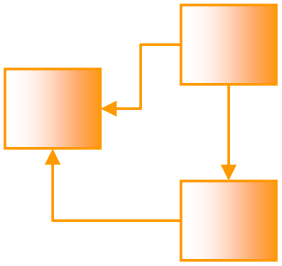




A **communication system** is the connection between program and „wire“. It allows the sending and receiving of messages over a network.

Communication





The Communication system of the TCP/IP architecture contains 4 layer. The interface layer represents the physical connection to the network. The internet layer is used to build a connection between computers. The transport layer is the transport service on a computer. The application layer connects the application with the communication system.

InetAddress

The class InetAddress encapsulates the name server functionality. This class is never instanziated directly. To get an instance their are several static methods:

```
static InetAddress getByName (String host)
```

returns the of the host which is named in the paramter.

```
static InetAddress[] getAllByName (String host)
```

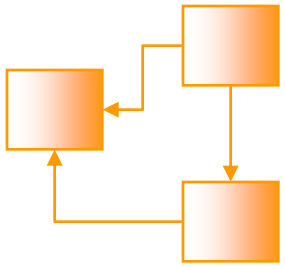
returns an array of all addresses of the host which is named in the parameter.

```
static InetAddress getLocalHost()
```

returns the adress of the local computer.

`getHostAddress()` returns the IP address of the host

`getHostName()` returns the hostname.



URL

To access resources in a network a URL (Uniform Resource Locator) can be used. A URL contains several parts:

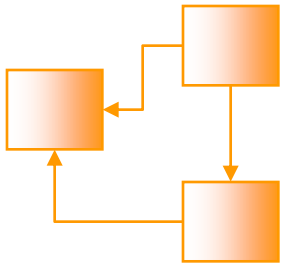
```
Protocoll://[Login[:Password]@]Hostname.Domain[:Port]
/Directory/Resource
```

Example:

```
http://www.hs-esslingen.de/index.html
```

To use URLs in Java two classes can be used: URL and URLConnection. Both classes can be found in the package `java.net`.

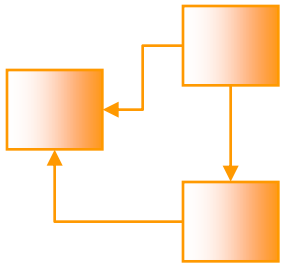
```
try {
   URL url = new URL (
      "http://www.hs-esslingen.de/index.html");
   System.out.println ("Protokoll: " +
      url.getProtocol());
   System.out.println ("Rechner:   " +
      url.getHost());
   System.out.println ("Datei:     " +
      url.getFile());
} catch (MalformedURLException e) {
   e.printStackTrace();
}
```



URL

The class URL also gives direct access to the resource:

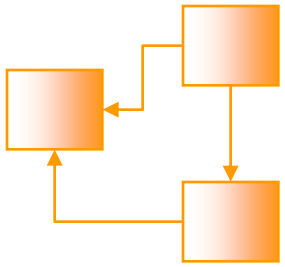
```
// Create a buffer
byte[] b = new byte[1024];
// Connect the resource
URL url = new URL (
   "http://www.hs-esslingen.de/index.html");

// Open a stream to read data
InputStream stream = url.openStream();

// Read the first part of the data
stream.read (b);
System.out.println (new String (b));
```



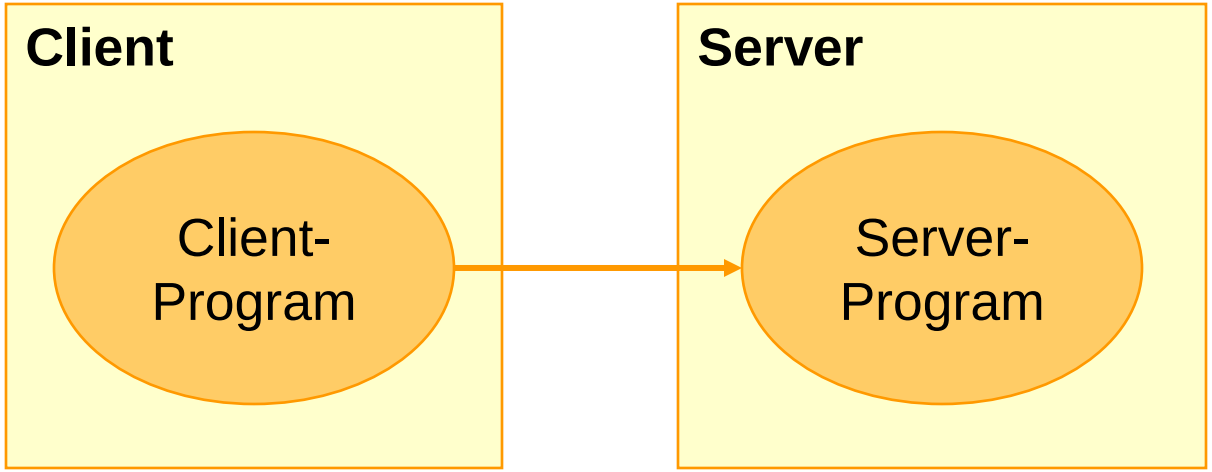
URLConnection

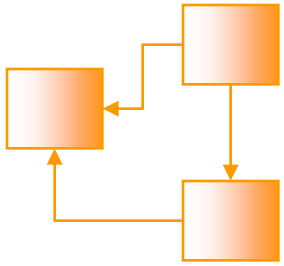
The abstract class URLConnection is the base class of all classes which want to connect a resource over an URL. Instances of these classes can access the resource for reading and writing:

```
// Create a URL
URL url = new URL (
   "http://www.hs-esslingen.de/index.html");

// Open the connection to the resource
URLConnection connection = url.openConnection();

// Connect the resource
connection.connect();

// Show the content of the HTTP-Header-Information
System.out.println (connection.getHeaderField (0));
```



Client and Server





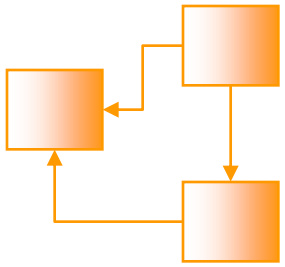
Client

A client sends transactions to an server application.

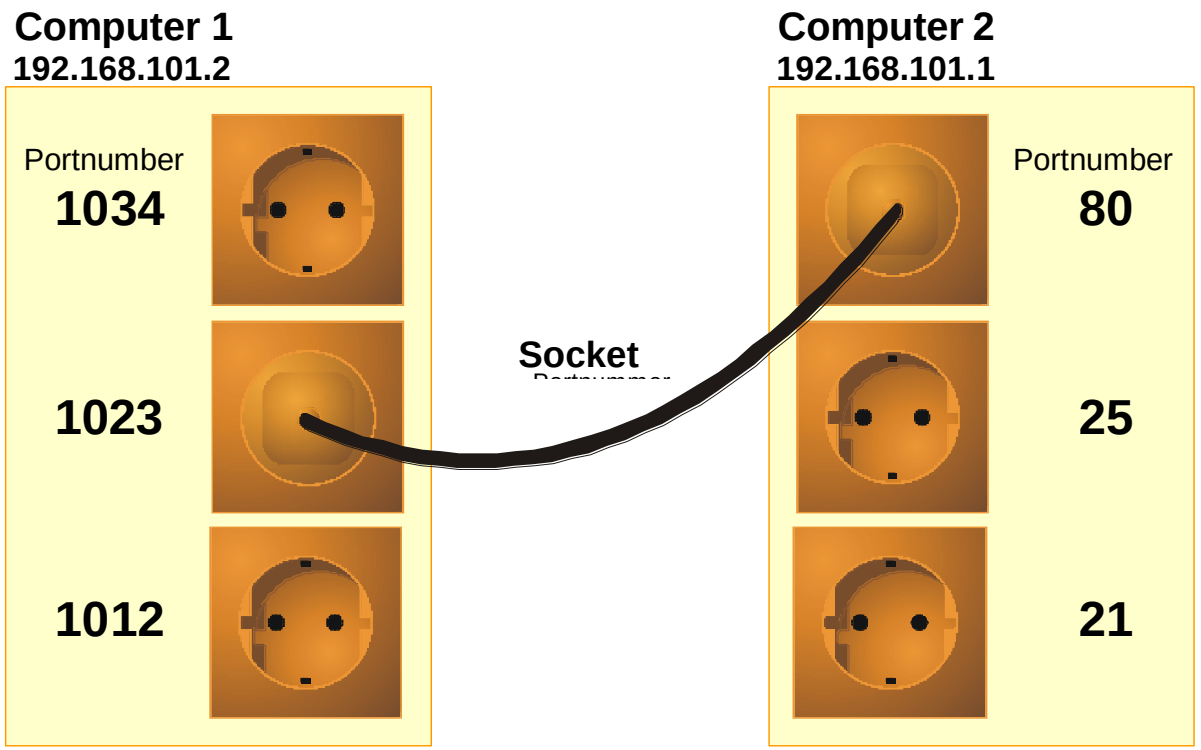
Server

A server application process the transactions from one or more clients.

Sockets



Address and Port

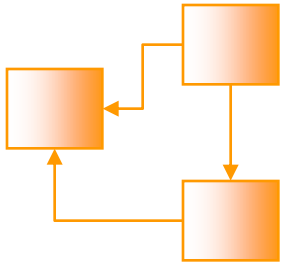


Socket

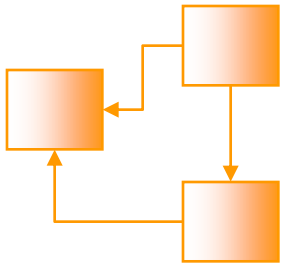
Endpoint of a communication connection between two Applications.

Transport Control Protocol
Connection-oriented protocol

Connection oriented protocols contain three essential steps:
1. Building a connection
2. Exchange of data
3. Closing the connection



TCP
Server

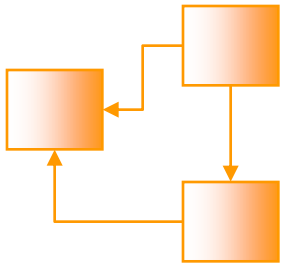


Java code (Server)

```
ServerSocket server = new ServerSocket (port);

while (true) {
    System.out.println ("Waiting for connection");
    Socket client = server.accept ();
    System.out.println ("Connection: " +
        client.getInetAddress ());

    DataOutputStream output = new DataOutputStream
        (client.getOutputStream ());

    output.writeBytes ("Hello Client");

    output.close ();
    client.close ();
}
```

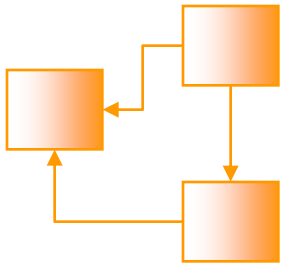
TCP:
- Connection oriented
- Using streams for exchanging data
- reliable

TCP
Client

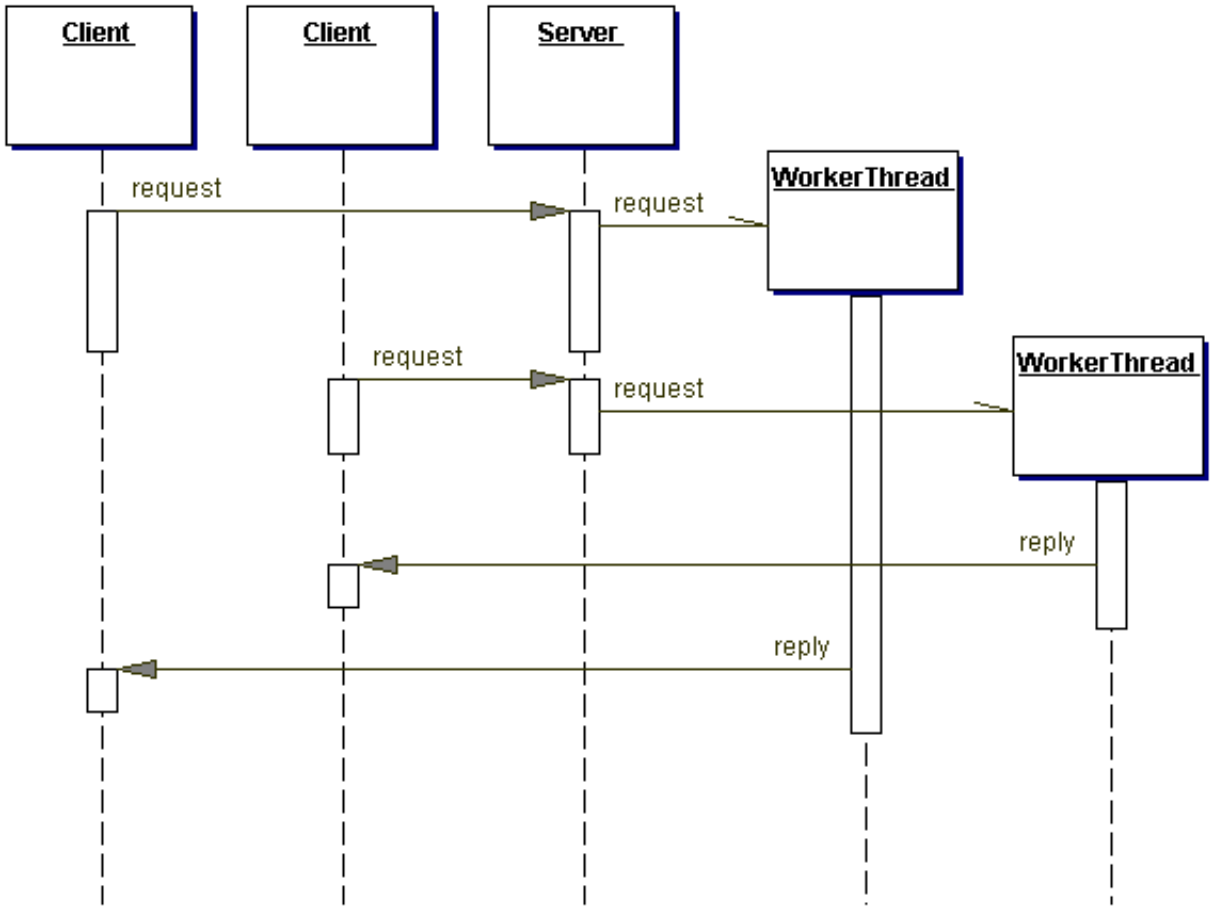


Java code (Client)

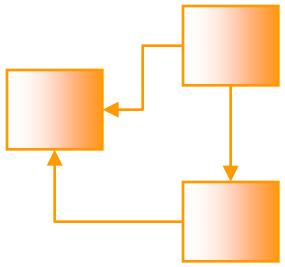
```
Socket socket = new Socket (hostname, port);

BufferedReader reader =
    new BufferedReader (new InputStreamReader
        (socket.getInputStream ()));

String answer = reader.readLine ();
System.out.println (answer);
socket.close ();
```



Example: Server with Worker-Thread



UDP
Communication



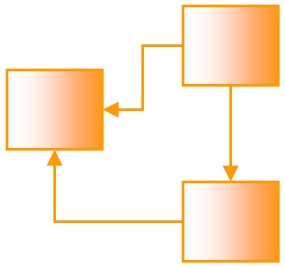
User Datagram Protocol
Connection-less protocol

When using UDP sockets every message (Datagram) must contain IP address and port of the recipient.

UDP
Server

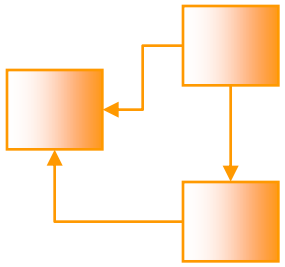
Java code (Server)

```
byte[] buffer = new byte[1024];
DatagramSocket server = new DatagramSocket (port);
DatagramPacket packet =
    new DatagramPacket (buffer, buffer.length);

while (true) {
    System.out.println ("Waiting for message ...");
    server.receive (packet);
    System.out.println ("Connection by: " +
        packet.getAddress ().toString ());
    System.out.println (new String (packet.getData ()));
}
```



UDP
- Connection less
- Uses messages for communication
- Is **not** reliable

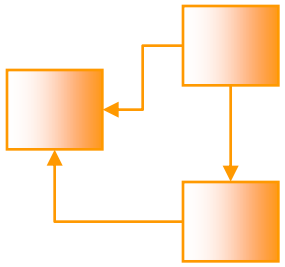
UDP
Client



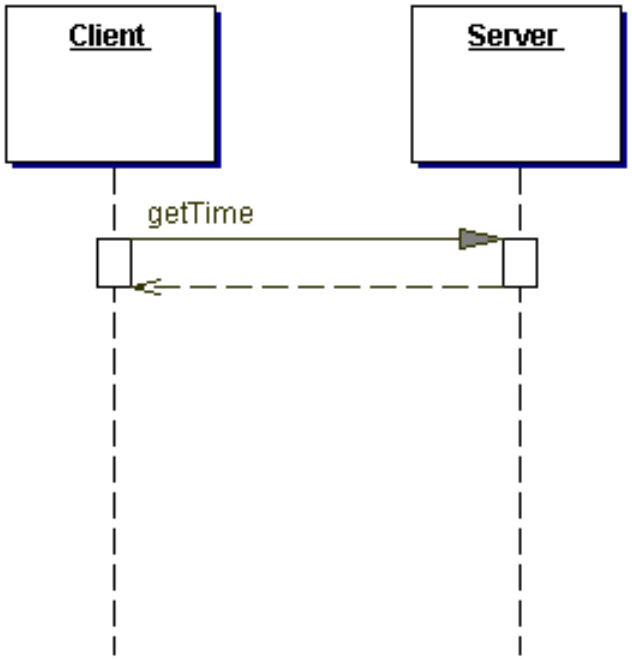
Java code (Client)

```
byte[] buffer = "Hello server".getBytes ();
DatagramSocket client = new DatagramSocket ();
InetAddress address = InetAddress.getByName (hostname);
DatagramPacket packet =
    new DatagramPacket (buffer, buffer.length,
                        address, port);
client.send (packet);
client.close ();
```

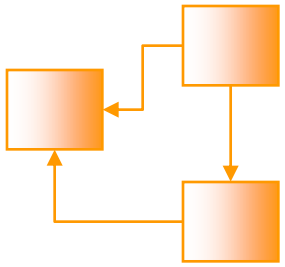
UDP
Application



Example: Time server

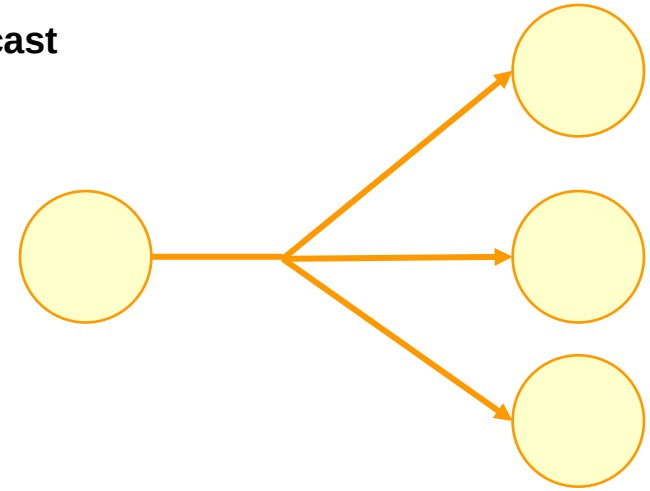


Multicast



Unicast

Multicast



Broadcast means that every sent message is received by all computers. Multicast only sends messages to a specified group of recipients.

Multicast
Groups

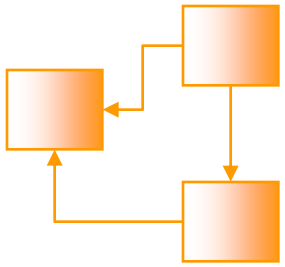
The groups are named by a specified range of IP addresses (class D address)

Class-D addresses: 224.0.0.0 to 239.255.255.255

224.0.0.0 to 224.0.0.255 is reserved for multicast routing information

Exmamples:

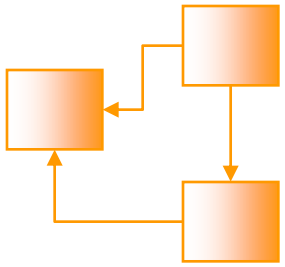
JINI:	The „magic“ of JINI
CORBA:	To connect the ORBs on different computers



Every IP multicast group has a *group address*.

IP multicast provides only *open groups*: That is, it is not necessary to be a member of a group in order to send datagrams to the group. [Keon00]

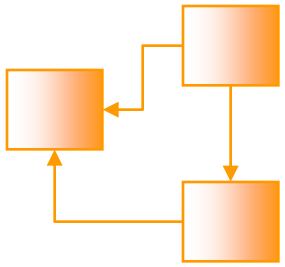
Multicast
Server



Java code (Sniffer)

```
byte[] buffer = new byte[1024];
DatagramPacket packet =
    new DatagramPacket (buffer, buffer.length);

InetAddress address = InetAddress.getByName ("224.3.4.5");

System.out.println ("Wait for message...");
MulticastSocket ms = new MulticastSocket (6789);
ms.joinGroup (address);

while (true) {
    ms.receive (packet);
    System.out.println (new String (packet.getData ()));
}

ms.leaveGroup (address);
```

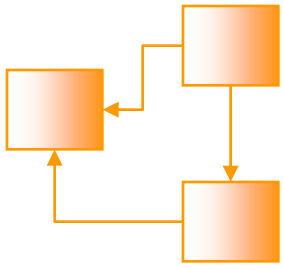
Multicast
Client



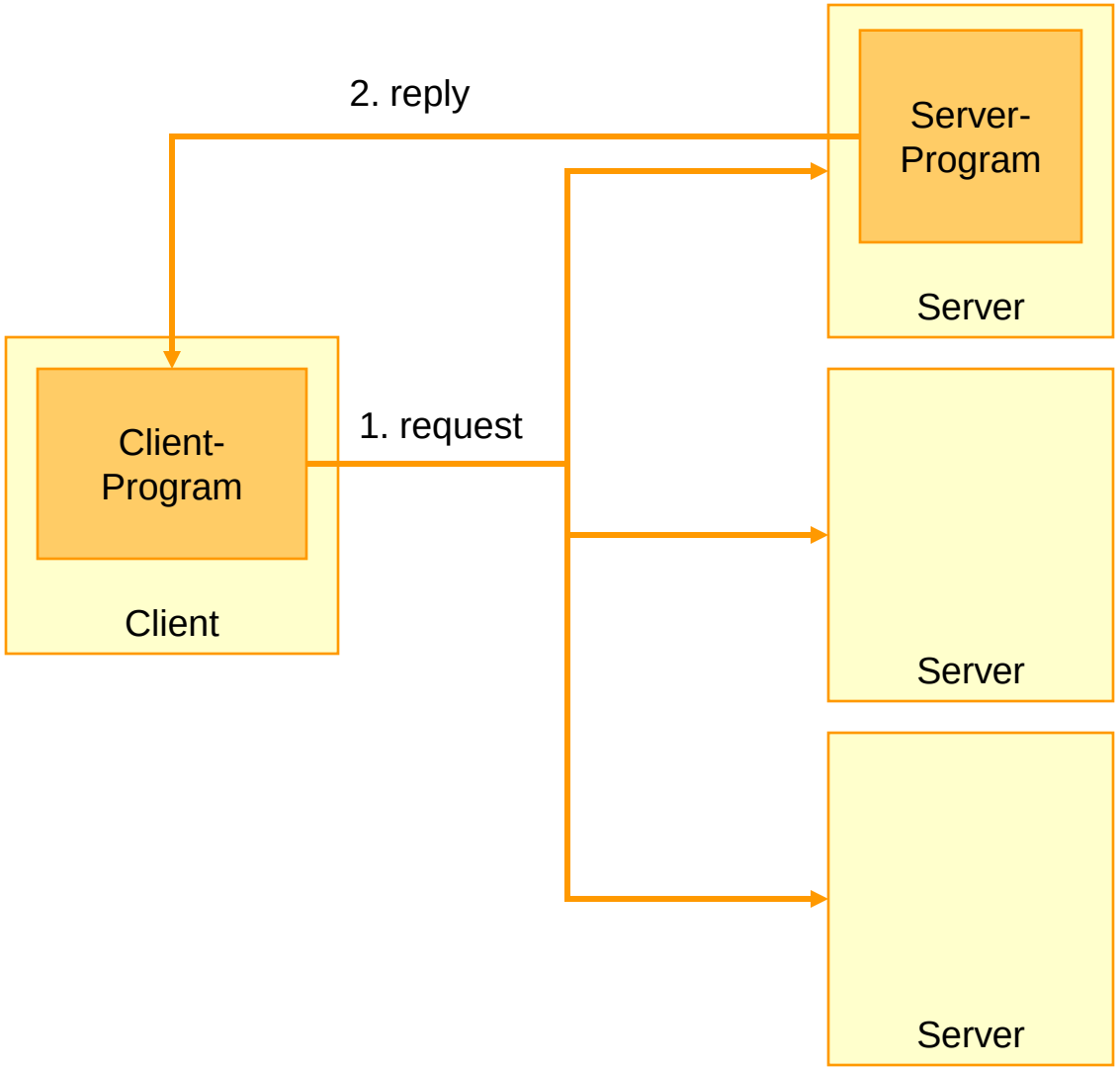
Java code (Sender)

```
byte[] buffer = "Hello server".getBytes ();
InetAddress address = InetAddress.getByName ("224.3.4.5");
DatagramPacket packet =
    new DatagramPacket (buffer, buffer.length,
                        address, 6789);

MulticastSocket ms = new MulticastSocket ();
ms.send (packet);
ms.close ();
```

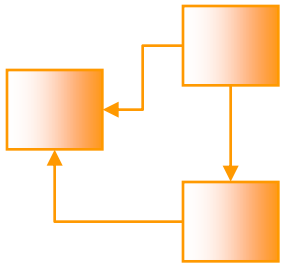
Multicast
Application



Example: Lookup

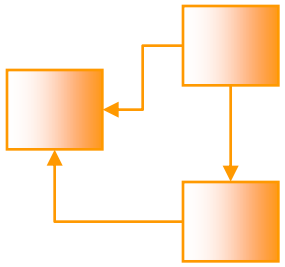


Protocolls



A protocol is a standard set of rules that governs how computers communicate with each other. Protocols describe both the format that a message must take and the way in which messages are exchanged between computers. [NeSo96]

Example HTTP



HTTP-Client

GET /index.html HTTP/1.0
Pragma: no-cache
Host: localhost
...

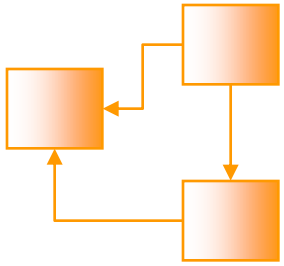
HTTP-Server

HTTP/1.0 200 OK
Content-Length: 485
Content-Type: text/html
<html> ...

Own protocolls

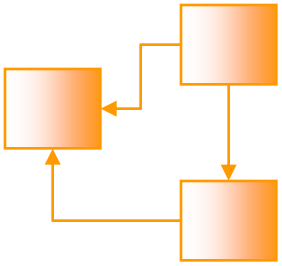
Method: setValue
Param1: key
Param2: value

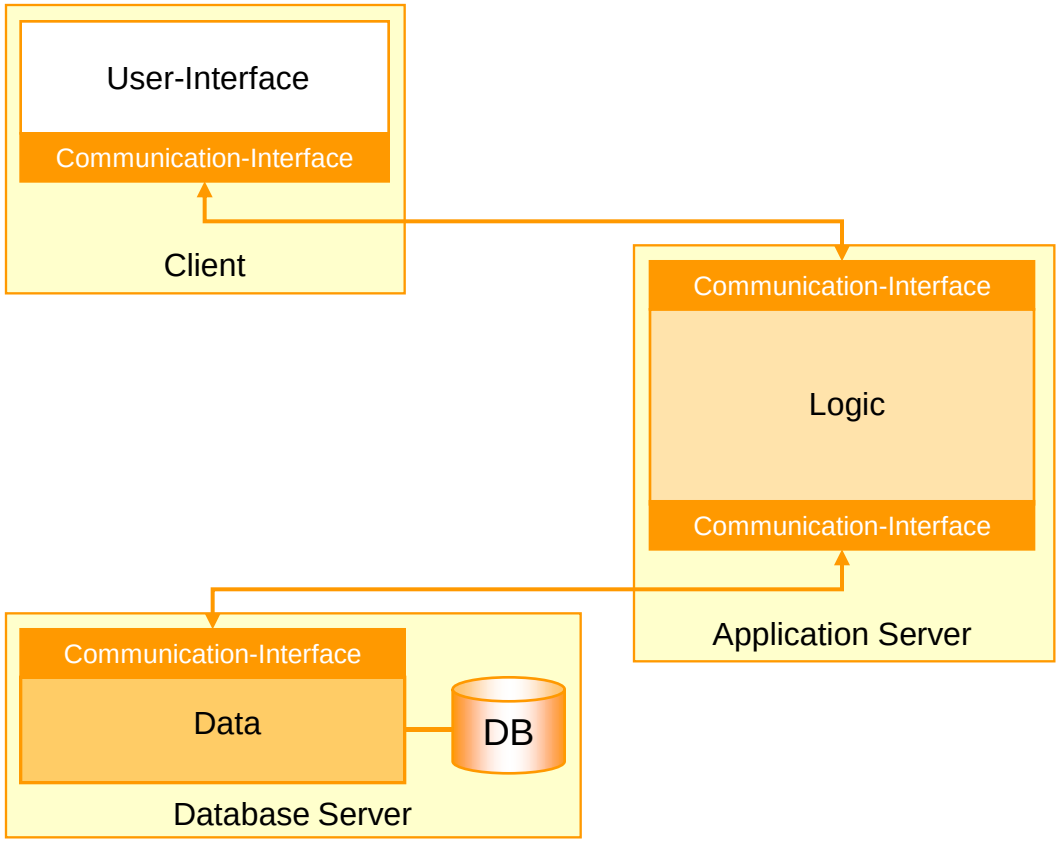
Return value

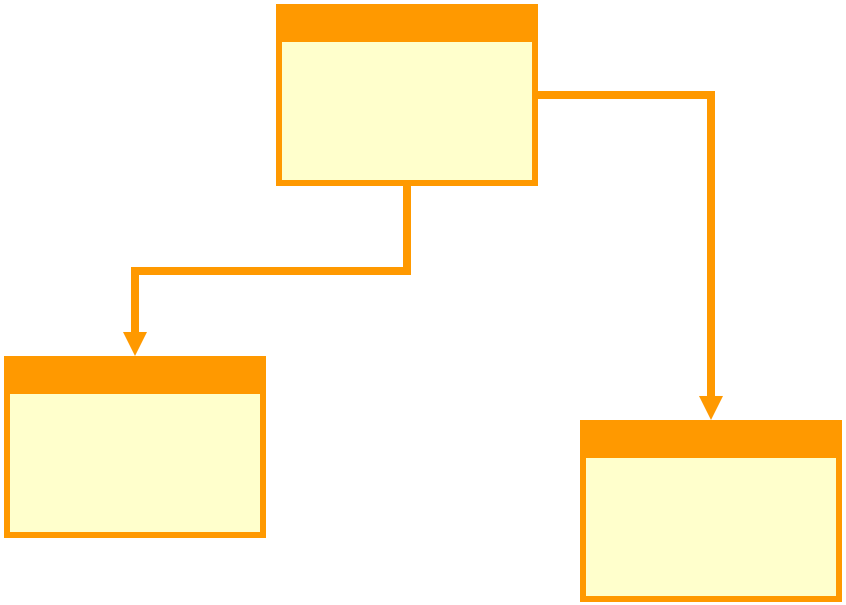


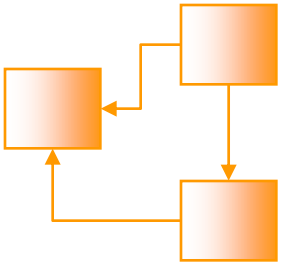
In a protocol we could also describe the methods which should be called in a specified server object (CORBA, RMI). It could also be possible to use XML to mark the different elements in the protocol (SOAP)

Distributed Systems





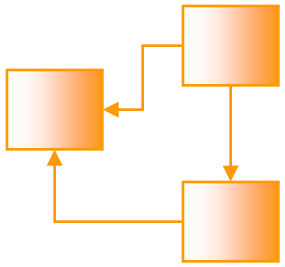




RMI

Remote Method Invocation

RMI



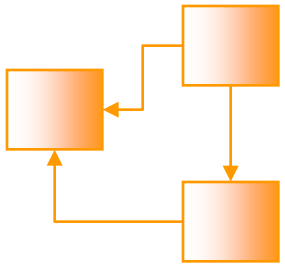
Usage

- Method calls between virtual machines (JVM)
- Method calls between JVMs on different computers
- Distributed applications:
  - Sharing resources
  - Load-Balancing

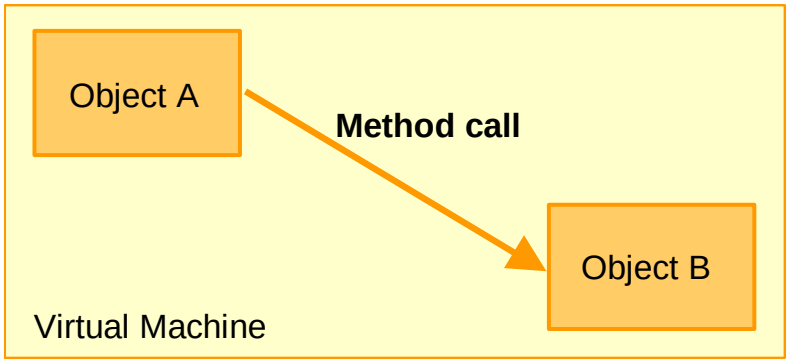
Properties

- Method calls
- Transparency
- Object by reference
- Object by value
- Dynamic loading of classes
- Java specific
- Since JDK 1.1

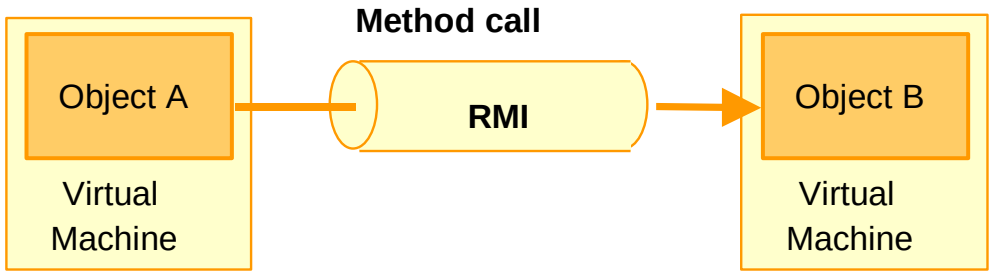
Extends the local method call



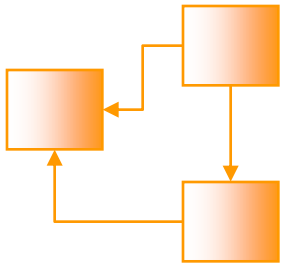
Local



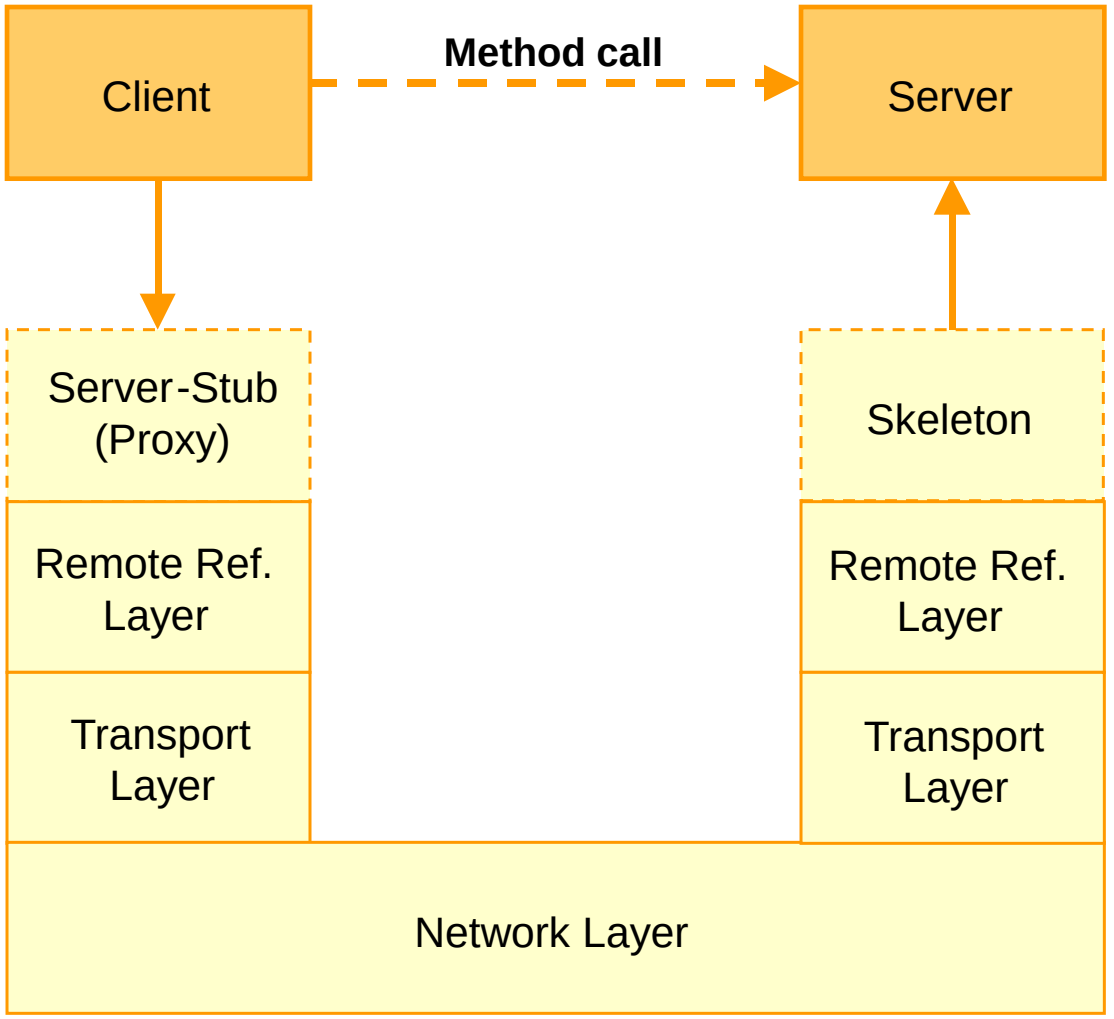
Remote



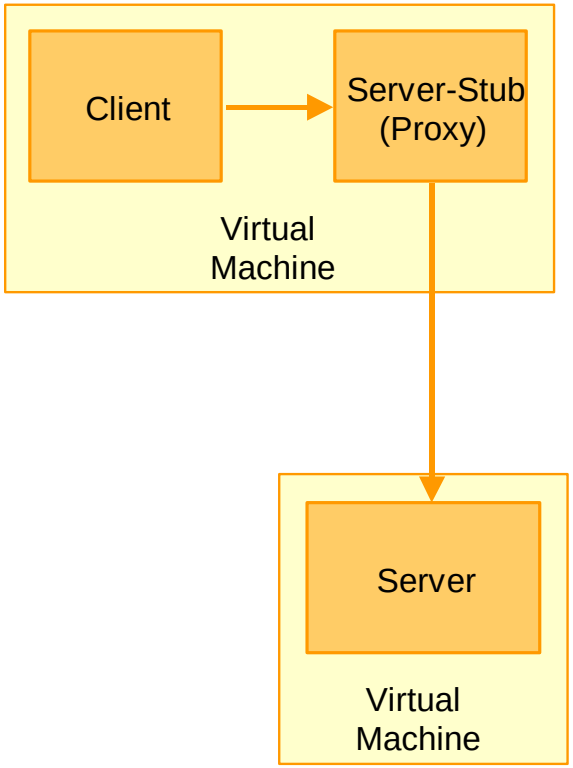
Transparency

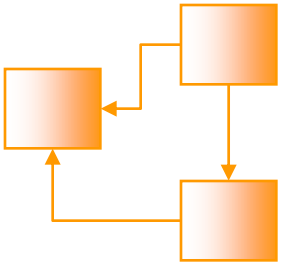


Architecture of RMI



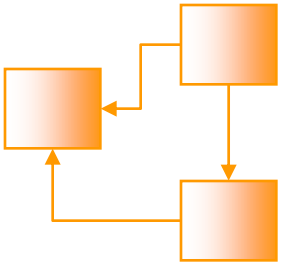
Proxy, Stub

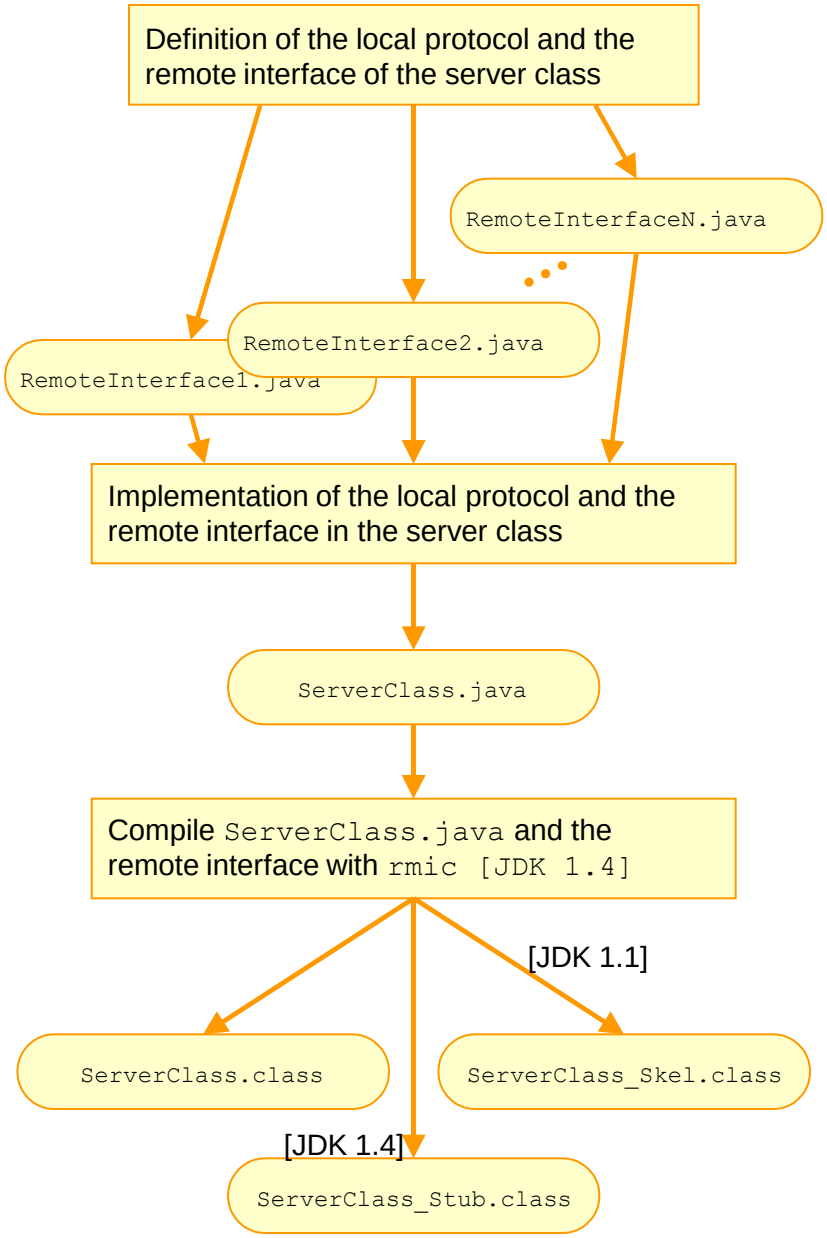




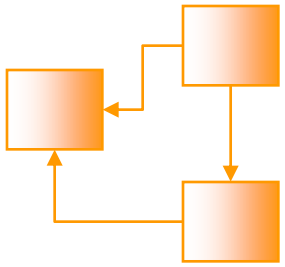
On account of calling the methods in the local proxy (Stub) of the server, the developer does not have to think about the network communication. For this, the stub class must implement the same interfaces like the server class. These interfaces are called remote interfaces and contain all methods the server wants to offer to the clients. The stub is generated automatically by the rmi compiler rmic.

Server-Implementation



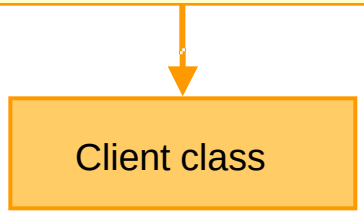


Client implementation ans startup



Implementation of client

1.	Implementation of the Client class
2.	Compile the Client class with `javac`

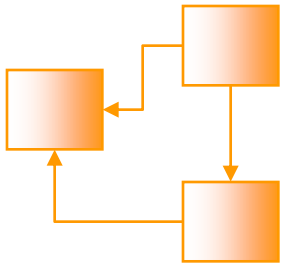


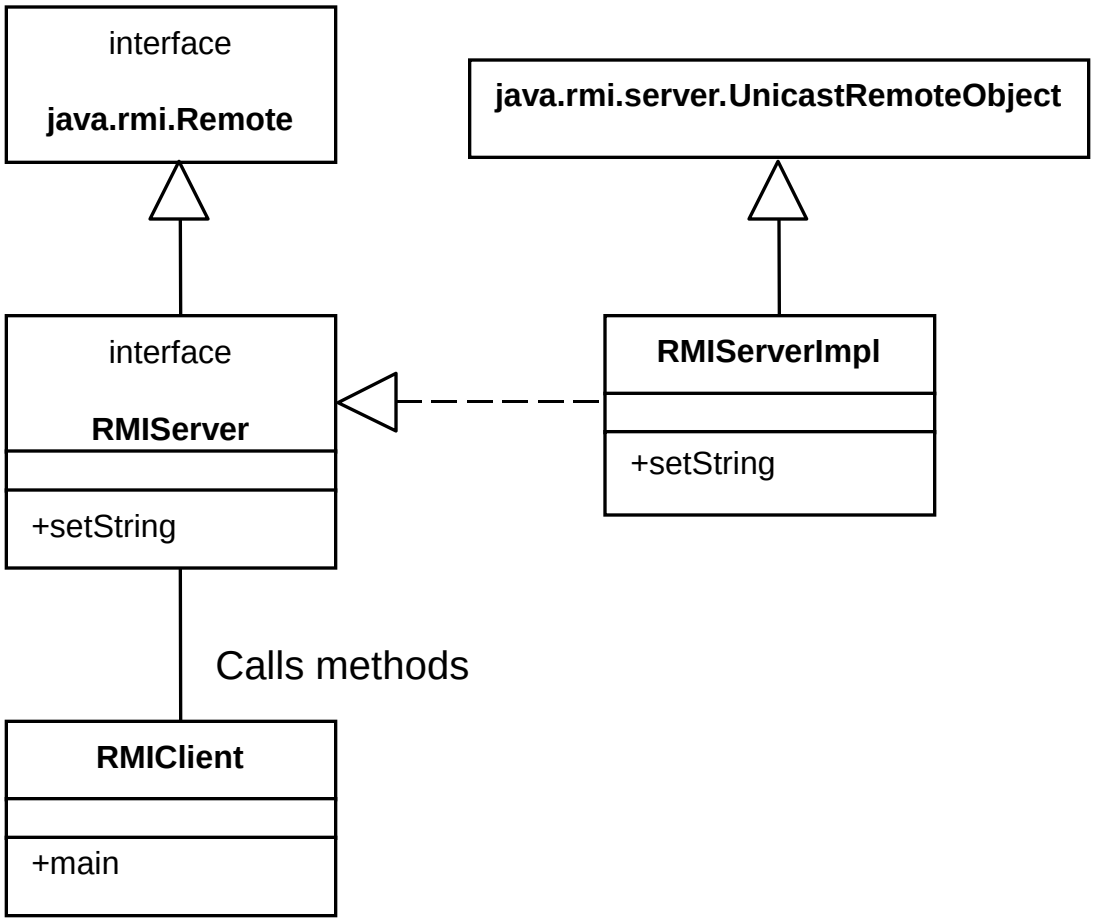
Communicationon

1.	Start the rmi registry
2.	Start the server
3.	Bind the server object
4.	Start the client
5.	Lookup
6.	Call methods of the server object

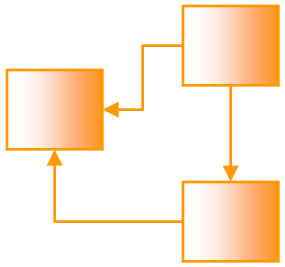
For the client implementation only the remote interface is used for calling methods of the server, never the server class itself

Class diagram





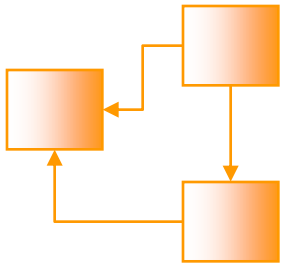
Implementation RemoteInterface



Java code (RemoteInterface)

```
// File: RMIServer.java

import java.rmi.*;

public interface RMIServer extends Remote
{
   // Method of the server which can be called
   // via remote
   void setString (String str) throws RemoteException;
}
```

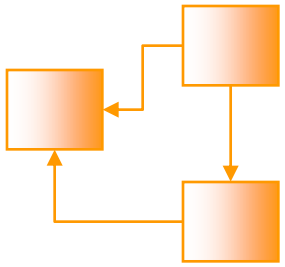
Implementation Server



Java code (Server)

```
// File: RMIServerImpl.java

import java.rmi.*;
import java.net.*;
import java.rmi.server.*;

public class RMIServerImpl extends UnicastRemoteObject
                           implements RMIServer {
   public RMIServerImpl () throws RemoteException {
      try {
         Naming.rebind ("rmi:///Server", this);
      } catch (Exception e) {
         System.out.println ("Error:\n" + e.getMessage ());
      }
   }

   public void setString (String str) {
      // Shows the string sent by the client
      System.out.println (str);
   }

   public static void main (String[] args) {
      try {
         RMIServerImpl server = new RMIServerImpl();
      } catch (RemoteException e) {
         System.out.println ("Error");
      }
   }
}
```

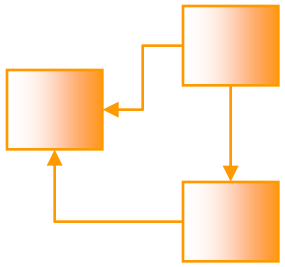
Implementation Client



Java code (Client)

```
// File: RMIClient.java

import java.rmi.*;
import java.net.*;

public class RMIClient {
   public static void main (String[] args) {
      try {
         // Search for the Server in RMIRegistry
         RMIServer server =
               (RMIServer)Naming.lookup ("rmi://localhost/Server");
         // Call a method from the server
         server.setString ("Hello Server");
      } catch (NotBoundException e) {
         System.out.println („Server not bound:\n" +
                            e.getMessage ());
      } catch (MalformedURLException e) {
         System.out.println ("URL wrong:\n" + e.getMessage ());
      } catch (RemoteException e) {
         System.out.println („Error in communication:\n" +
                            e.getMessage ());
      }
   }
}
```

Start the application



To start the program several steps have to be done. At first all classes must be compiled:

```
javac *.java
```

Following files are generated:

```
RMIClient.class, RMIServer.class, RMIServerImpl.class
```

Then the stub class must be generated (JDK 1.4):

```
rmic RMIServerImpl
```

This creates the file `RMIServerImpl_Stub.class`

To generate the stub, the name of the server class must be passed to the compiler. This must be done because the server class can implement more than one remote interface. The skeleton will not be created because it is not needed since the JDK 1.2

To start the rmiregistry the following program must be executed:

```
rmiregistry
```

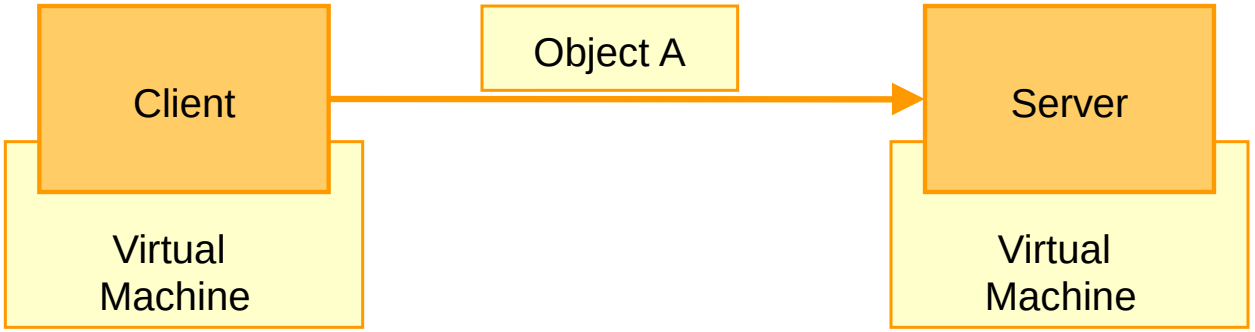
After this the server can be started:

```
java RMIServerImpl
```

The next line starts the client:

```
java RMIClient
```

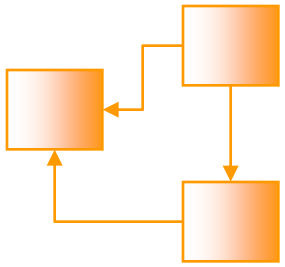
Passing objects to the server



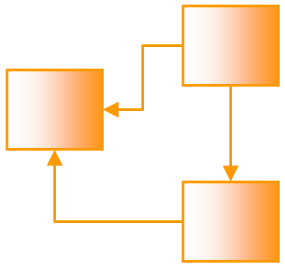
Object A is serialisable
-> The Server gets a clone of object A
Object by value

Object A is a remote object:
-> The server gets a clone of the stub of object A
Object by reference

Object A is not a remote object and not serializable
-> Exception



Object by value



Object by Value means that the transferred object will be sent as a clone. Changes in this object, which are done by the server will not influence the instanciated object in the client. So that it is possible, the transferred object must be serializable. This means it must implement the interface Serializable from the package java.io. This interface does not contain any methods, it is just a „marker interface“. The following code example shows a serializable class.

```
// File: Data.java

import java.io.*;

public class Data implements Serializable {
   public int i, n;

   public Data (int i, int n) {
      this.i = i;
      this.n = n;
   }
}
```

Object by value

In the remote interface must be defined a method for transfering the data object to the server:

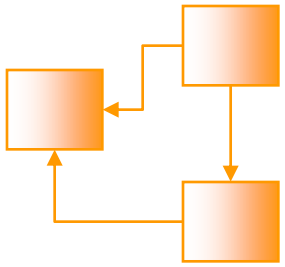
```
void setData (Data data) throws RemoteException;
```

The method in the server class:

```
public void setData (Data data) {
   System.out.println ("Received data: " + data.i +
      ", " + data.n);
}
```

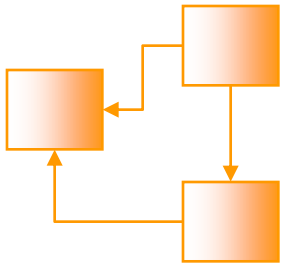
The client instanziates the data object and gives it to the server:

```
// Create the object
Data data = new Data (1, 2);
// Daten senden
server.setData (data);
```



Changes made in the transferred object do not influence the state of the original object at the client

Object by reference



The following example shows how the client connects to the server so that it can be called by the server. For this the client must also be a remote object, so it has to implement a remote interface:

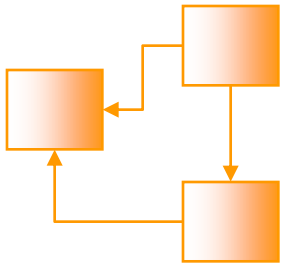
```
// File: RMIClient.java

import java.rmi.*;

public interface RMIClient extends Remote {
void setString (String str) throws RemoteException;
}
```

So that the client can connect to the server, the remote interface of the server must contain the following method:

```
void setClient (RMIClient client)
```

Not the client object itself will be transferred to the server but an instance of the stub. For this a stub-class has to be generated for the client (JDK 1.4)

Object by reference



The server class implements the following method:

```
public void setClient (RMIClient client) {
   System.out.println ("Client angemeldet");
   try {
      // Methode im Client ausführen
      client.setString ("Hallo Client");
   } catch (Exception e) {
      e.printStackTrace ();
   }
}
```

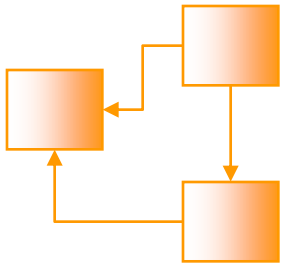
The client connects to the server by sending its own reference:

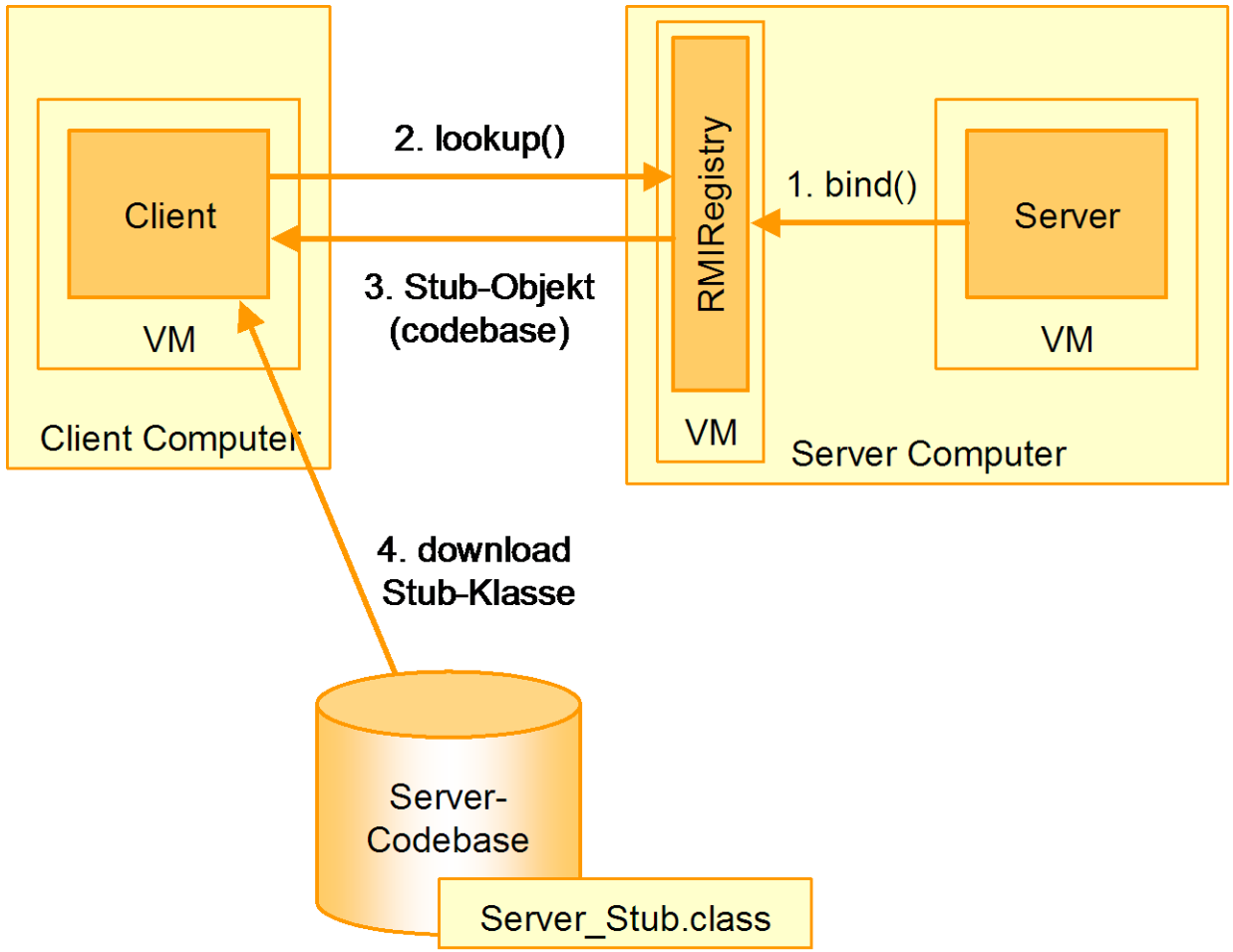
```
server.setClient (this);
```

The client implements the following remote method:

```
public void setString (String str) {
   System.out.println (str);
}
```

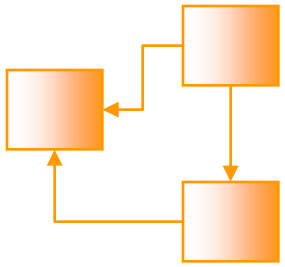
Download of classes





Used classes will be loaded dynamically from the codebase if needed.

Using the codebase

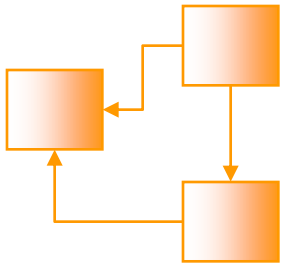


For distributing a rmi application on several Computers, the following topics must be considered:

1. The classes of the server must not be found in the classpath of the rmiregistry. Else the codebase will be ignored by the rmi registry.
2. The codebase must be set at the server.
3. The codebase must contain the protocol for loading the classes.
4. A SecurityManager must be set on server- and on the client side.
5. The rights for the SecurityManager must be set in the policy.

If the rmi application can not load the stub-class from the codebase, this class is automatically created by the framework. Own classes, referenced in the remote-interface can then not be used any longer.

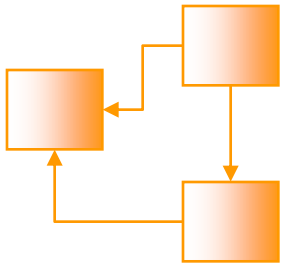
Simple example: Interface



Implementation of the remote interface:

```
import java.rmi.*;

public interface Server extends Remote {
    void service () throws RemoteException;

}
```

Simple example: Server



Implementation of the server:

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

public class RMIServerImpl extends UnicastRemoteObject
                           implements Server {

    public RMIServerImpl () throws RemoteException {
        super ();
        System.out.println ("Try to bind ...");
        Naming.rebind ("rmi:///Server", this);
    }

    public void service () {
        System.out.println ("Client uses service");
    }

    public static void main (String args[]){
        try {
            System.setSecurityManager
                (new RMISecurityManager ());
            RMIServerImpl server = new RMIServerImpl ();
            System.out.println
                ("bound\nWaiting for connection, press s to stop");
            while (System.in.read () != 's');
            Naming.unbind ("rmi:///Server");
            System.exit (0);
        } catch (Exception e) {
            e.printStackTrace ();
        }
    }
}
```

Executing the Server

Start the server:

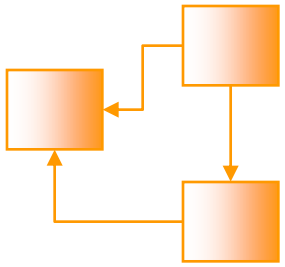
```
java -Djava.rmi.server.codebase=protocol://location
     -Djava.security.policy=policy.all
     ServerImpl
```

Codebase:

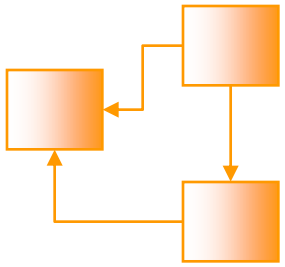
```
file:/c:/rmi/server/
http://hostname/classes
```

Policy:

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



Simple example: Client



Implementation of the client:

```
import java.rmi.*;

public class RMIClient {
    public RMIClient (String host) {
        try {
            System.setSecurityManager
                (new RMISecurityManager ());

            RMIServer server =
                (RMIServer)Naming.lookup
                ("rmi://" + host + "/Server");
            server.service ();
        } catch (Exception e) {
            e.printStackTrace ();
        }
    }

    public static void main (String[] args) {
        try {
           RMIClient client = new RMIClient
                                ("localhost„s);
        } catch (Exception e) {
           e.printStackTrace ();
        }
    }
}
```

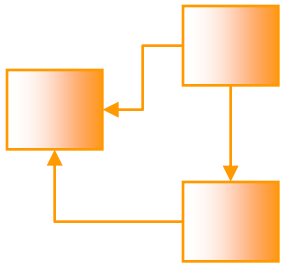
Executing the client

Compiling the client classes:

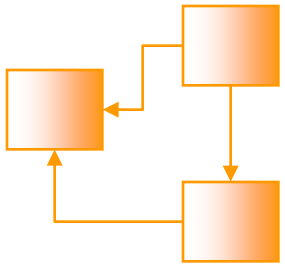
```
javac *.java
```

Startup of the client:

```
java -Djava.security.policy=policy.all RMIClient
```



Files



The classes of the application and the policy file have to be distributed as follows:

Directory of the server (`server`):

```
RMIServer.class
RMIServerImpl.class
[RMIServerImpl_Stub.class]
policy.all
```

Codebase directory (`codebase`):

```
RMIServer.class
[RMIServerImpl_Stub.class]
```

Directory of the client (`client`):

```
RMIClient.class
RMIServer.class
policy.all
```