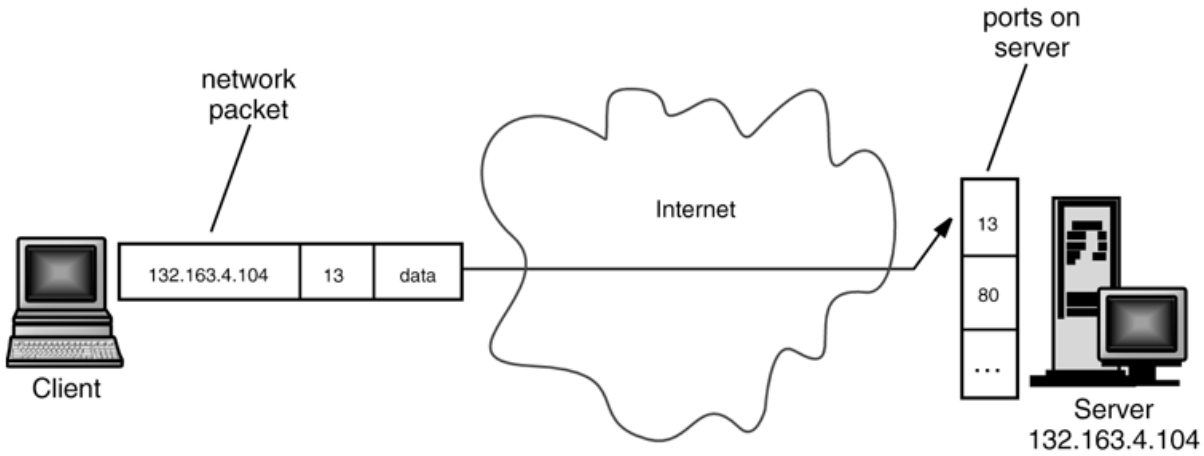
1. Using Sockets

The socket is simply a software construct that represents one endpoint of a connection. They make it possible to create your own applications that can communicate with one another. With stream sockets, a process establishes a connection to another process. While the connection is in place, data flows between the processes in continuous streams. Stream sockets are said to provide a connection-oriented service. The protocol used for transmission is the popular TCP (Transmission Control Protocol).
With datagram sockets, individual packets of information are transmitted. This is not appropriate for everyday programmers, because the protocol used UDP (User Datagram Protocols) is a connectionless service. It does not guarantee that packets arrive in any particular order. With UDP, packets can even be lost or duplicated.



The server software is continuously running on the remote machine, waiting for any network traffic that wants to chat with port 13. When the operating system on the remote computer receives a network package that contains a request to connect to port number 13, it wakes up the listening server process and establishes the connection. The connection stays up until it is terminated by one of the parties.

```
import java.io.*;
import java.net.*;

class SocketTest
{  public static void main(String[] args)
   {  try
      {  Socket t = new Socket("time-A.timefreq.bldrdoc.gov",
            13);
         BufferedReader is = new BufferedReader
            (new InputStreamReader(t.getInputStream()));
         boolean more = true;
         while (more)
         {  String str = is.readLine();
            if (str == null) more = false;
            else
               System.out.println(str);
         }
         if (!t.isClosed())  // not necessary
             t.close();
      }
      catch(IOException e)
```

```
      { System.out.println("Error" + e); }
   }
}

57264 15-08-30 09:24:02 50 0 0 525.9 UTC(NIST) *
```

With sockets, network I/O appears to Java programs to be similar to sequential file I/O. Sockets hide much of the complexity of network programming from the programmer.

With Java's multithreading, we can create multithreaded servers that can manage many simultaneous connections with many clients. This multithreaded-server architecture is precisely what popular network servers use.

1.1. Server

Establishing a simple server in Java requires five steps.

1.1.1. Step 1

Creating a ServerSocket object.

```
ServerSocket server = new ServerSocket( portNumber[, queueLength] );
```

registers an available TCP port number and specifies a maximum number of clients that can wait to connect to the server (i.e., the queue length). The port number is used by clients to locate the server application on the server computer. If the queue is full, the server refuses client connections. The constructor establishes the port where the server waits for connections from clients a process known as binding the server to the port. Each client will ask to connect to the server on this port. Only one application at a time can be bound to a specific port on the server.
Port numbers can be between 0 and 65,535. Most operating systems reserve port numbers below 1024 for system services.
Remarks:

- A port of 0 creates a socket on any free port.
- The maximum queue length for incoming connection indications (a request to connect) is set to 50. If a connection indication arrives when the queue is full, the connection is refused.

1.1.2. Step 2,

The server listens indefinitely (or blocks) for an attempt by a client to connect. To listen for a client connection, the program calls ServerSocket method accept, as in

```
Socket connection = server.accept();
```

which returns a Socket when a connection with a client is established.

1.1.3. Step 3

The creation of the OutputStream and InputStream objects that enable the server to communicate with the client by sending and receiving bytes. The server sends information to the client via an OutputStream and receives information from the client via an InputStream. The server invokes method getOutputStream on the Socket to get a reference to the Socket's

OutputStream and invokes method getInputStream on the Socket to get a reference to the Socket's InputStream.
Whatever the server writes to the ObjectOutputStream is sent via the OutputStream and is available at the client's InputStream.
Whatever the client writes to its OutputStream (with a corresponding ObjectOutputStream) is available via the server's InputStream.
The transmission of the data over the network is seamless and is handled completely by Java.

1.1.4. Step 4

The next step is the processing phase in which the server and the client communicate via the OutputStream and InputStream objects.
Both sides must agree upon a specific protocol. There is no support from the compiler.

1.1.5. Step 5

When the transmission is complete, the server closes the connection by invoking the close method on the streams and on the Socket.

1.2. Client

There are 4 steps that are necessary.

1.2.1. Step 1

It is the Client that initializes the connection. The server address and port must be known. The Socket constructor establishes the connection to the server. For example, the statement

```
Socket connection = new Socket( serverAddress, port );
```

uses the Socket constructor with two arguments the server's address (serverAddress) and the port number. If the connection attempt is successful, this statement returns a Socket. A connection attempt that fails throws an instance of a subclass of IOException, so many programs simply catch IOException. Use localhost for test purposes.

1.2.2. Step 2

The client uses Socket methods getInputStream and getOutputStream to obtain references to the Socket's InputStream and OutputStream. If the server is sending information in the form of actual types, the client should receive the information in the same format. Thus, if the server sends values with an ObjectOutputStream, the client should read those values with an ObjectInputStream.

1.2.3. Step 3

The step is the processing phase in which the client and the server communicate via the InputStream and OutputStream objects.

1.2.4. Step 4,

The client closes the connection when the transmission is complete by invoking the close method on the streams and on the Socket. The client must determine when the server is finished sending information so that it can call close to close the Socket connection. For example, the InputStream method read returns the value 1 when it detects end-of-stream (also called EOFend-of-file). If an ObjectInputStream is used to read information from the server,

an EOFException occurs when the client attempts to read a value from a stream on which end-of-stream is detected.

1.3. Example

An example of a server that is connected to multiply clients.

- Line based communication, the server sends back any line that it receives
- The number of clients that are served simultaneously is limited to a predefined number.

1.3.1. Client

```
public class EchoClient {
	public static void main(String[] argv) {
		EchoClient c = new EchoClient();
		c.converse(argv.length==1?argv[0]:"localhost");
	}
	/** Hold one conversation with the named hosts echo server */
	protected void converse(String hostname) {
		Socket sock = null;
		try {
			sock = new Socket(hostname, 8189);     // echo server.
			BufferedReader stdin = new BufferedReader(
				new InputStreamReader(System.in));
			BufferedReader is = new BufferedReader(
				new InputStreamReader(sock.getInputStream(), "8859_1"));
			PrintWriter os = new PrintWriter(
				new OutputStreamWriter(
					sock.getOutputStream(), "8859_1"), true);

			String line;
			do {
				System.out.print(">> ");
				if ((line = stdin.readLine()) == null)
					break;
				if (line.startsWith("****"))
					break;
			// Do the CRLF ourself since println appends only a \r on
			// platforms where that is the native line ending.
				os.print(line + "\r\n");
				os.flush();
				String reply = is.readLine();
				System.out.print("<< ");
				System.out.println(reply);
			} while (line != null);
		} catch (IOException e) { // handles all input/output errors
			System.err.println(e);
		} finally {                          // cleanup
			try {
				if (sock != null)
					sock.close();
			} catch (IOException ignoreMe) {
				// nothing
			}
		}
	}
}
```

1.3.2. Server

```
public class EchoServerThreaded2 {
	public static final int ECHOPORT = 8189;

	public static final int NUM_THREADS = 4;
	/** Main method, to start the servers. */
	public static void main(String[] av)
	{
		new EchoServerThreaded2(ECHOPORT, NUM_THREADS);
	}

	/** Constructor */
	public EchoServerThreaded2(int port, int numThreads)
	{
		ServerSocket servSock;

		try {
			servSock = new ServerSocket(port);

		} catch(IOException e) {
			/* Crash the server if IO fails. Something bad has happened */
			throw new RuntimeException("Could not create ServerSocket " +
e);
		}

		// Create a series of threads and start them.
		for (int i=0; i<numThreads; i++) {
			new Handler(servSock, i).start();
		}
	}

	/** A Thread subclass to handle one client conversation. */
	class Handler extends Thread {
		ServerSocket servSock;
		int threadNumber;

		/** Construct a Handler. */
		Handler(ServerSocket s, int i) {
			super();
			servSock = s;
			threadNumber = i;
			setName("Thread " + threadNumber);
		}

		public void run()
		{
			/* Wait for a connection. Synchronized on the ServerSocket
			 * while calling its accept() method.
			 */
			while (true) {
				try {
					System.out.println( getName() + " waiting");

					Socket clientSocket;
					// Wait here for the next connection.
					synchronized(servSock) {
						clientSocket = servSock.accept();
					}
					System.out.println(getName() + " starting, IP=" +
```

```
                                    clientSocket.getInetAddress());
                              BufferedReader is = new BufferedReader(
                                    new
InputStreamReader(clientSocket.getInputStream()));
                              PrintStream os = new PrintStream(
                                    clientSocket.getOutputStream(), true);
                              String line;
                              while ((line = is.readLine()) != null) {
                                    os.print(line + "\r\n");
                                    os.flush();
                              }
                              System.out.println(getName() + " ENDED ");
                              clientSocket.close();
                        } catch (IOException ex) {
                              System.out.println(getName() + ": IO Error on
socket " + ex);
                              return;
                        }
                  }
            }
      }
}
```

1.4. Remarks

- Sockets offer no type control over the stream of data. Server and Client must "understand" each other.
- The much more needed flexibility could be achieved by transmitting serialized objects. To do so we have to serialize objects to a sequence of bytes and restore them from it.

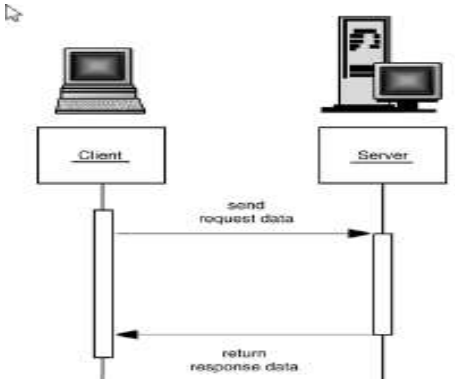
```
public byte[] convertToBytes(MyClass object) throws IOException {
          byte[] res=null;
            try (ByteArrayOutputStream bos = new ByteArrayOutputStream();
               ObjectOutput out = new ObjectOutputStream(bos)) {
              out.writeObject(object);
              res=bos.toByteArray();
              bos.close();
              return res;
          }
      }

public Object convertFromBytes(byte[] bytes) throws IOException,
ClassNotFoundException {
          try (ByteArrayInputStream bis = new ByteArrayInputStream(bytes);
               ObjectInput in = new ObjectInputStream(bis)) {
            Object ob=in.readObject();
            in.close();
             return ob;
          }
      }
```

2. RMI Remote Method Invocation

2.1. Introduction

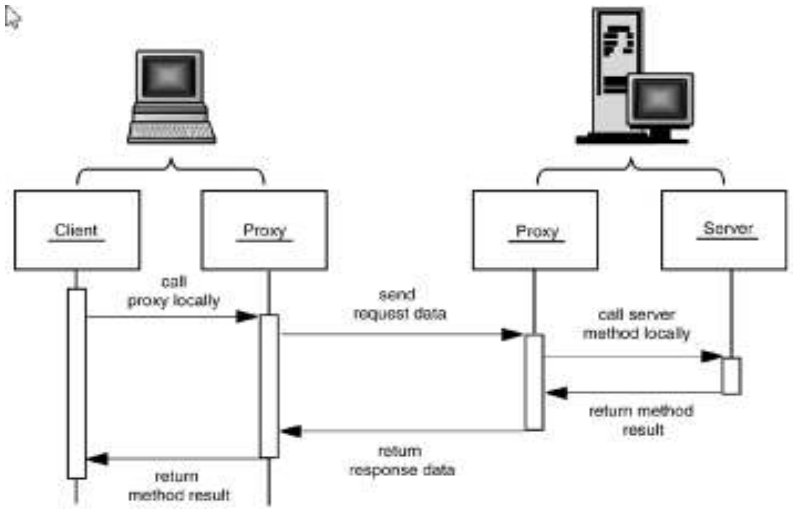
The sockets make it possible to transfer data or even objects from one computer to another over a network. What usually need is not data but processing. Parsing a message is not flexible and does not offer any support from the compiler. The sockets are equivalent of memo fields in the database world.



A data database is far more powerful. It offers a language e.g. the SQL. The capabilities of the language are strictly limited to data base processing.

Sockets can transmit objects. There is no compiler support. The remote objects techniques enable us to USE objects located anywhere. The compiler checks whether we use the remote objects in a syntactical proper way.

The idea is to have a bunch of collaborating objects that can be located anywhere. These objects are supposed to communicate through standard protocols across a network. You'll have an object on the client where the user can fill in a request for data. The client object sends a message to an object on the server that contains the details of the request. The server object gathers the requested information, perhaps by accessing a database or by communicating with additional objects. Once the server object has the answer to the client request, it sends the answer back to the client.
What we want is a mechanism by which the client programmer makes a regular method call, without worrying about sending data across the network or parsing the response.
The solution is to install a proxy for the server object on the client. The client calls the proxy, making a regular method call. The client proxy contacts the server. Similarly, the programmer of the server object doesn't want to fuss with client communication. The solution is to install a second proxy object on the server.



The proxies could communicate using:

- RMI, the Java Remote Method Invocation technology, supports method calls between distributed Java objects.

- CORBA, the Common Object Request Broker Architecture, supports method calls between objects of any programming language. CORBA uses the Internet Inter-ORB Protocol, or IIOP, to communicate between objects.

- SOAP, the Simple Object Access Protocol, is also programming-language neutral. However, SOAP uses an XML-based transmission format.

RMI is Java specific and is easier to understand than CORBA or SOAP. It is also used in the EJB architecture (Enterprise Java Beans).
CORBA and SOAP are completely language neutral. Client and server programs can be written in C, C++, Java, or any other language.

3. Basics of Remote Method Invocation

As the name indicates the Remote Method Invocation mechanism lets you call methods of the remote object and obviously obtain the result calculated by the method.
In RMI terminology, the object whose method makes the remote call is called the client object. The remote object is called the server object. This is valid for a single method call. It is entirely possible that the roles of client and server are reversed as the computation continues.

3.1. Stubs and Parameter Marshalling

When client code wants to invoke a remote method on a remote object, it actually calls an ordinary method on a proxy object called a stub. The stub resides on the client machine, not on the server. The stub packages the parameters used in the remote method into a block of bytes. This packaging uses a device-independent encoding for each parameter.
The process of encoding the parameters is called parameter marshalling. The purpose of parameter marshalling is to convert the parameters into a format suitable for transport from one virtual machine to another.
The stub method on the client builds an information block that consists of:

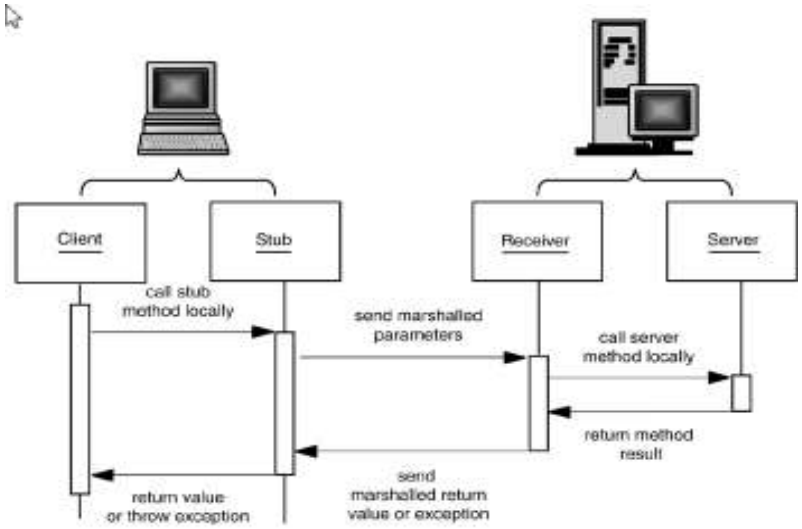
- An identifier of the remote object to be used;
- A description of the method to be called; and

- The marshalled parameters.

The stub sends this information to the server. On the server side, a receiver object performs the following actions for every remote method call:

- It unmarshals the parameters.
- It locates the object to be called.
- It calls the desired method.
- It captures and marshals the return value or exception of the call.
- It sends a package consisting of the marshalled return data back to the stub on the client.

The remote objects look very much like local objects. The syntax for a remote method call is the same as for a local call. This process is completely automatic and, to a large extent, transparent for the programmer.



3.2. Naming conventions

We need many classes, naming conventions make it easier to make out what is the purpose of a class.

No suffix (e.g., `Product`)	A remote interface
`Impl` suffix (e.g., `ProductImpl`)	A server class implementing that interface
`Server` suffix (e.g., `ProductServer`)	A server program that creates server objects
`Client` suffix (e.g., `ProductClient`)	A client program that calls remote methods

3.3. Setting the RMI

Running even the simplest remote object example requires quite a lot of setup much more than does running a standalone program.

- You must run programs on both the server and client computers.
- The necessary object information must be separated into client-side interfaces and server-side implementations.
- A special lookup mechanism allows the client to locate objects on the server.

The client program needs to manipulate server objects, but it doesn't actually have copies of them. The objects themselves reside on the server. The client code must still know what it can do with those objects.

The solution: an interface that is shared between the client and server and so resides simultaneously on both machines. It describes the capabilities of a shared object.

The concept is illustrated by a typical example of a client - server application.

3.3.1. Interfaces and Implementations

The capabilities of objects are expressed in an interface that is shared between the client and server.

```
import java.rmi.*;

public interface Product extends Remote
{  String getDescription()
          throws RemoteException;
}
```

All interfaces for remote objects must extend the Remote interface.
All the methods in those interfaces must also declare that they will throw a RemoteException.
The remote method calls are inherently less reliable than local and it is always possible that a remote call will fail.

The product implementation:

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

public class ProductImpl extends UnicastRemoteObject implements Product
{
        private static final long serialVersionUID = 1L;
        public ProductImpl(String n)
                        throws RemoteException
                        {  name = n;
                        }

        public String getDescription()
                        throws RemoteException
                        {  return "I am a " + name + ". Buy me!";
                        }

        private String name;
}
```

This class is a server for remote methods because it extends UnicastRemoteObject.
The class is a concrete Java platform class that makes objects remotely accessible.

The example is by no way typical as it has only one very simple method. Usually a server class does some heavy-duty work that a client could not carry out locally.

Occasionally, you may not want a server class that extends the UnicastRemoteObject class, perhaps because it already extends another class. In that situation, you need to manually instantiate the server objects and pass them to the static exportObject method. Instead of extending UnicastRemoteObject, call

UnicastRemoteObject.exportObject(this, 0);

in the constructor of the server object. The second parameter is 0 to indicate that any suitable port can be used to listen to client connections.

3.3.2. Locating Server Objects

We have the chicken-and-egg problem.
To access a remote object that exists on the server, the client needs a local stub object. How can the client request such a stub? The most common method is to call a remote method of another server object and get a stub object as a return value.
The Sun RMI library provides a bootstrap registry service to locate the first server object.
A server program registers objects with the bootstrap registry service, and the client retrieves stubs to those objects. You register a server object by giving the bootstrap registry service a reference to the object and a name. The name is a string that is (hopefully) unique.

Class ProductServer:

```
ProductImpl p1 = new ProductImpl("Blackwell Toaster");
Context namingContext = new InitialContext();
namingContext.bind("rmi:toaster", p1);
```

and on the client side, class ProductClient:

```
Product c1 = (Product)Naming.lookup(url + "toaster");
```

RMI URLs start with rmi:// and are followed by a server, an optional port number, another slash, and the name of the remote object, e.g.

```
rmi://localhost:99/central_warehouse
```

By default, the server is localhost and the port number is 1099.
In practice there should be relatively few named server objects registered with the bootstrap service. These should be objects that can locate other objects for you.

For security reasons, an application can bind, unbind, or rebind registry object references only if it runs on the same host as the registry. This prevents hostile clients from changing the registry information. However, any client can look up objects.

3.4. The code

3.4.1. ProductServer

```
public class ProductServer
{
```

```
	public static void main(String args[])
	{
		try
		{
			System.out.println("Constructing server
implementations...");

			ProductImpl p1 = new ProductImpl("Blackwell Toaster");
			ProductImpl p2 = new ProductImpl("ZapXpress Microwave
Oven");

			System.out.println("Binding server implementations to
registry...");
			Context namingContext = new InitialContext();
			namingContext.bind("rmi:toaster", p1);
			namingContext.bind("rmi:microwave", p2);
			System.out.println("Waiting for invocations from
clients...");
		}
		catch (Exception e)
		{
			e.printStackTrace();
		}
	}
}
```

The Context interface represents a naming context, which consists of a set of name-to-object bindings. It contains methods for examining and updating these bindings.

3.4.2. ProductImpl

```
public class ProductImpl extends UnicastRemoteObject implements Product
{
	private static final long serialVersionUID = 1L;
	public ProductImpl(String n)
			throws RemoteException
			{  name = n;
			}

	public String getDescription()
			throws RemoteException
			{  return "I am a " + name + ". Buy me!";
			}

	private String name;
}
```

3.4.3. Product

```
public interface Product extends Remote
{  String getDescription()
		throws RemoteException;
}
```

3.5. Starting the Server

Our server program isn't quite ready to run yet.
It uses the bootstrap RMI registry so the service must be available.
To start the RMI registry under Windows, call

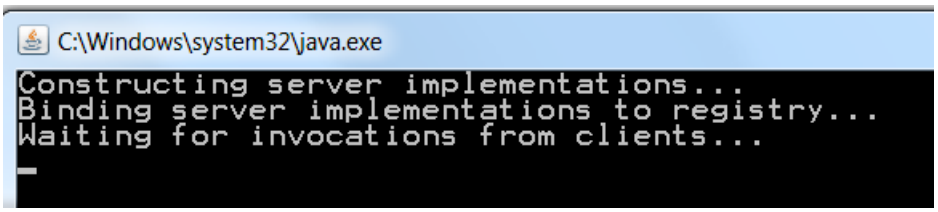
start rmiregistry

at a DOS prompt.

Now you are ready to start the server:

start java ProductServer

start java RMI/ProductServer



Remarks:

- the ProductServer must be on the ClassPath.
- The program exist immediately, its sole purpose is to register the products.
- Do not run the ProductServer class directly from JVC, you have to use the start command just like the threads are created with the call of start not run method.
- A separate thread for handling invocations does not stop although the main function has ended.
- It is necessary to REMOVE the network firewall. Using the RMI with firewall is described e.g. " http://www.netcluesoft.com/rmi-through-a-firewall.html".

3.6. The client Side

Client programs that use RMI should install a security manager to control the activities of the dynamically loaded stubs using the RMISecurityManager. To install use the instruction

System.setSecurityManager(new RMISecurityManager());

The manager uses system property java.security.policy. In the example it is controlled by the file client.policy.

It could be done specifically:

```
grant
{
   permission java.net.SocketPermission
      "192.168.0.10:50596", "connect,resolve";
   permission java.net.SocketPermission
      "127.0.0.1:1099", "connect,resolve";
};
```

or more general:

```
grant
{
     permission java.net.SocketPermission
      "*:1024-65535", "connect,resolve";
};
```

or even more general:

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

This is a regular file and we can use any text editor to create and/or modify it.

The code:

```
public class ProductClient
{
      public static void main(String[] args)
      {
            System.setProperty("java.security.policy", "client.policy");
            System.setSecurityManager(new RMISecurityManager());
            String url = "rmi://localhost/";
            // change to "rmi://yourserver.com/" when server runs on remote
machine  yourserver.com
            try
            {
                  Context namingContext = new InitialContext();
                  Product c1 = (Product) namingContext.lookup(url +
"toaster");
                  Product c2 = (Product) namingContext.lookup(url +
"microwave");
                  System.out.println(c1.getDescription());
                  System.out.println(c2.getDescription());
            }
            catch (Exception e)
            {
                  e.printStackTrace();
            }
      }
}
```

The base folder should contain the file client.policy

We can alternatively specify directly the policy file:

System.setProperty("java.security.policy","file:./clientX.policy");

what is more foolproof.

The context is used to locate the products on the server,
The results:

c:\myJava\LJ2014\bin>java rMI/ProductClient

```
I am a Blackwell Toaster. Buy me!
I am a ZapXpress Microwave Oven. Buy me!
```

It is also possible to specify the system property setting on the command line:

java -Djava.security.policy=client.policy ProductClient

Multiple server objects on the same server can share a port. However, if a remote call is made and the port is busy, another port is opened automatically. Thus, you should expect somewhat fewer ports to be used than there are remote objects on the server.

3.7. Naming context in detail

Is used for accessing the RMI registry. The methods:

- static Object lookup(String name)

returns the object for the given name. Throws a NamingException if the name is not currently bound.

- static void bind(String name, Object obj)

binds name to the object obj. Throws a NameAlreadyBoundException if the object is already bound.

- static void unbind(String name)

unbinds the name. It is legal to unbind a name that doesn't exist.

- static void rebind(String name, Object obj)

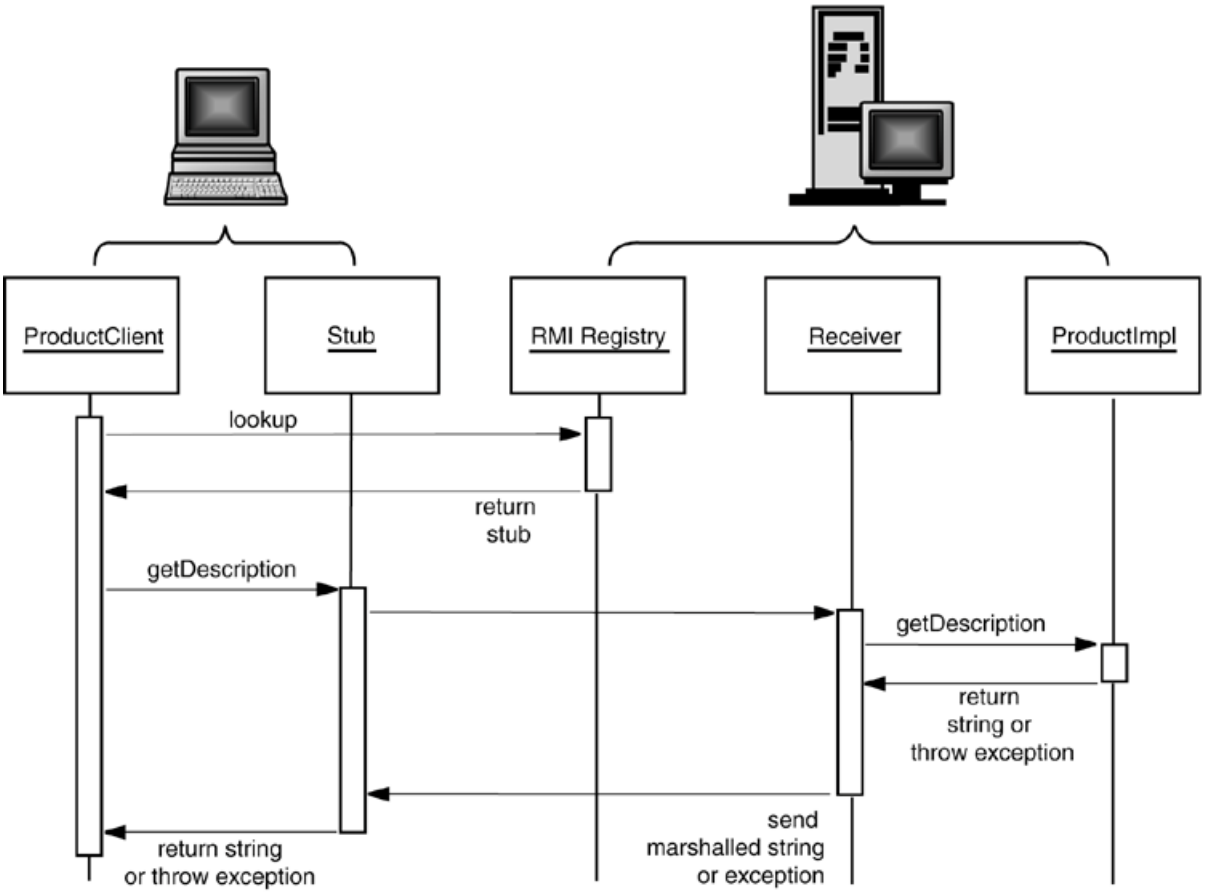
binds name to the object obj. Replaces any existing binding.
NamingEnumeration<NameClassPair> list(String name)

- boolean hasMore()

returns true if this enumeration has more elements.

- T next()

returns the next element of this enumeration.



returns an enumeration listing all matching bound objects. To list all RMI objects, call with "rmi:".

4. Parameter Passing in Remote Methods

Not so simple there are some pitfalls.

4.1. Passing Non Remote Objects

When a remote object is passed from the server to the client, the client receives a stub. Using the stub, it can manipulate the server object by invoking remote methods. The object, however, stays on the server.
If the invoked method returns object that does not implement the Remote interface the client receives a copy of it. This is the case of the above example. The returned object is not connected to the server any more. The RMI mechanism uses the serialization to transport objects.
The principles:

- Non remote objects are copied.
- All of this is automatic and requires no programmer intervention.
- The objects are serialized.

4.2. Passing Remote Objects

Passing remote objects from the server to the client is simple.

- Remote objects are passed across the network as stubs.
- The client receives a stub object, then saves it in an object variable with the same type as the remote interface.
- The client can access the actual object on the server through the variable.
- Only the remote interfaces can be accessed through the stub.
- A remote interface is any interface extending Remote.
- Local (not included in the interface) methods can run only on the virtual machine containing the actual object.
- If a subclass doesn't implement a remote interface but a superclass does, and an object of the subclass is passed to a remote method, only the superclass methods are accessible.

4.3. Remote Objects and the equals and hashCode Methods

There is a problem when trying to compare remote objects. To find out if two remote objects have the same contents, the call to equals would need to contact the servers containing the objects and compare their contents. And that call could fail.

- Equals

The equals method in the class Object is not declared to throw a RemoteException, whereas all methods in a remote interface must throw that exception. Because a subclass method cannot throw more exceptions than the superclass method it replaces, you cannot define an equals method in a remote interface.

- HashCode

The same holds for hashCode.
The equals and hashCode methods on stub objects simply look at the location of the server objects. The equals method deems two stubs equal if they refer to the same server object. Two stubs that refer to different server objects are never equal, even if those objects have identical contents. Similarly, the hash code is computed only from the object identifier.

To summarize, you can use stub objects in sets and hash tables, but you must remember that equality testing and hashing **do not** take into account the contents of the remote objects.

4.4. Cloning Remote Objects

Stubs do not have a clone method, so you cannot clone a remote object by invoking clone on the stub.
There is a very good reason for that. If clone were to make a remote call to tell the server to clone the implementation object, then the clone method would need to throw a RemoteException.
This is not possible as the clone method in the Object superclass promised never to throw any exception other than CloneNotSupportedException.